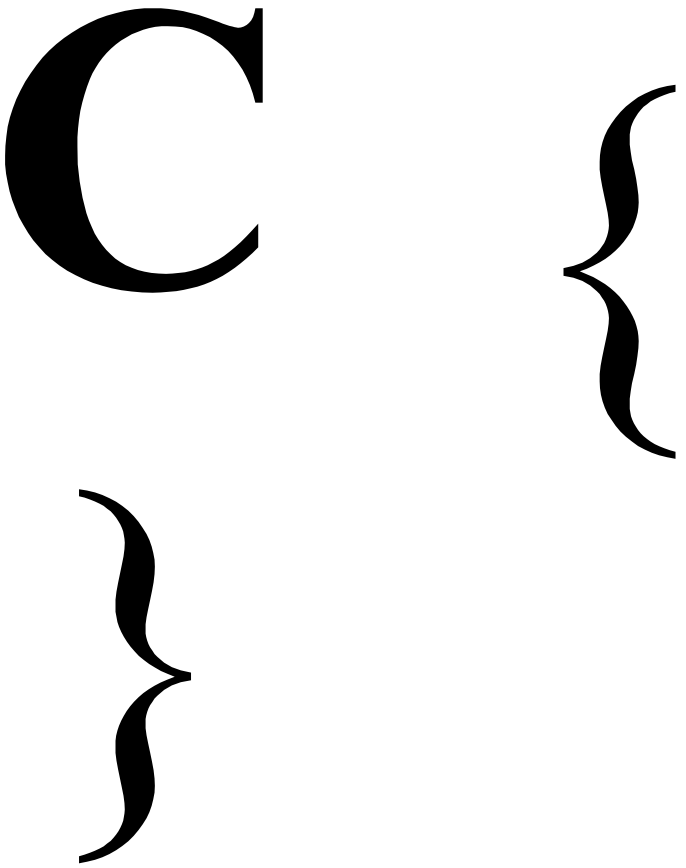
Learning C For Real

By Sergio C. Carbone



www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Table of Contents

Introduction to C 13
- The history of the C programming language. 13
- C compared to other programming languages. 13
- C viewed as a structured language. 13
- C's place in today's development environment. 13
- Common first reactions to C. 14
- Challenges facing C students. 14
- Quick Test. 14

Authoring and Executing a C Program 15
- Coding a C application. 15
- Storing the application source code. 15
- Converting the application source code to object code. 15
- Binding the application object code to form an executable image. 16
- Submitting the application image for processing. 16
- The dangers of not following the order. 16
- Quick Test. 16
- Assignment. 16

Program Layout 18
- The basic syntax and layout of a C program 18
- Program information comment. 18
- External inclusions. 19
- Global declarations. 19
- Function prototypes. 19
- Primary function. 19
- Hidden declarations and prototypes. 19
- Function bodies. 20
- Quick Test. 20

Data Storage in C 21
- A quick look at the way memory works. 21
- The main areas of memory. 21
- How addresses play a role in C programming. 21
- Quick Test. 22

Built in Data Types 23
- The character data type. 23
- The integer data type. 23
- The float data type. 23
- The double data type. 24
- The void data type. 24
- Pointer data types. 24
- Getting the size of data. 25
- Quick Test. 25

Arrays in C 26
- A general description of what an array is. 26
- The notation used to work with arrays. 26

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Null terminated arrays of characters better known as strings. 26
Quick Test. 27
Assignment. 27
Introduction to Functions 28
The reason for functions in C. 28
The main sources for functions. 28
The layout of functions in C. 28
The function prototype. 28
The function arguments. 29
The return value. 29
Quick Test. 30
Standard Input and Output 33
The background on standard I/O. 33
What is meant by the stream concept. 33
A word on operating system redirection. 33
Displaying information with standard output functions. 33
Accepting data using standard input functions. 35
What needs to be done to use standard input and output. 37
Examples. 37
Quick Test. 37
Assignment. 37
Tips for Using Data Storage 39
Initializing your data storage. 39
The impact of function arguments and local variables on performance. 39
The impact of global variables on code complexity and reuse. 39
Quick Test. 40
Assignment. 40
Conditional Execution 41
How true and false works in C. 41
The logical operators in C. 41
Executing conditionally with an if statement. 41
Trapping condition failure with an else statement. 42
General rules regarding if..else statements. 42
Nesting if and if..else statements. 42
Executing conditionally with a switch..case statement. 43
The use of a default case. 44
How the break statement is used with switch..case statements. 44
Altering values conditionally with the ?: operator. 44
Nesting the ?: operator. 45
Boolean short circuiting. 45
Examples. 45
Quick Test. 45
Assignment. 46
Iterative Execution Control 47
An overview of iterative control. 47

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Using a "for" loop. 47
Using a "while" loop. 48
Using a "do..while" loop. 48
Using the break statement with iterative control. 49
Using the continue statement with iterative control. 50
Examples. 50
Quick Test. 50
Assignment. 50
Branch Control 53
A word on why not to read this section. 53
How to implement code labels and goto statements. 53
Examples. 53
Quick Test. 54
Assignment. 54
Handling Basic Mathematics 55
The background on basic mathematics. 55
The mathematical operators. 55
What is meant by prefix and post-fix notation. 56
Calculating information with math functions. 56
What needs to be done to use math functions. 57
The bit level operators. 58
Examples. 58
Quick Test. 58
Assignment. 59
Handling Strings, Memory, and Characters 61
What needs to be remembered when working with strings. 61
Manipulating strings, memory, and characters with functions. 61
What needs to be done to use string, memory, or character functions. 74
Examples. 74
Quick Test. 74
Assignment. 74
Backslash Control Characters 75
The background on backslash control characters. 75
Quick Test. 75
The Conversion Functions 77
The background on the conversion functions. 77
Quick Test. 80
Assignment. 80
The Preprocessor 81
The background on the preprocessor. 81
Including other source files in your compilation. 81
Working with text substitution identifiers. 81
Conditional compilation. 82
Stopping compilation. 83
Examples. 83

Quick Test. 84
Assignment. 84
User Defined Data Types 85
Using structures to group related but dissimilar data. 85
Using unions to view the same memory in different ways. 86
Using enumerations. 87
Creating formal data types. 87
Casting data types so they can fit different roles. 88
Examples. 88
Quick Test. 88
Assignment. 89
Utilizing Dynamic Memory 93
The background on dynamic memory. 93
Allocating and de-allocating dynamic memory with functions. 93
What needs to be remembered when working with memory. 94
A word about virtual memory. 95
Examples. 95
Quick Test. 95
Assignment. 96
Recursive Function Calls 97
An overview of recursive function calls. 97
Examples. 97
Quick Test. 97
An Introduction to Files 99
Three basic file types. 99
File names and paths. 100
How DOS deal with files. 100
Quick Test. 101
Accessing Files with Standard I/O 103
The background on the files and standard I/O. 103
Opening and closing files. 103
Working text files. 105
Working binary files. 106
What needs to be done to use standard input and output. 108
Examples. 108
Quick Test. 108
Assignment. 108
Accessing Files with UNIX Style I/O 109
The background on the UNIX style file I/O. 109
Opening and closing files. 109
Working binary files. 111
What needs to be done to use UNIX style I/O. 112
Examples. 112
Quick Test. 112
Assignment. 113

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Implementing Complex Data Structures Dynamically 115
Leveraging memory with data structures. 115
Pointers to pointers to pointers. 115
Starting with linked lists. 115
Examples. 116
Quick Test. 116
Pointer Arithmetic 117
General overview of pointer arithmetic. 117
Examples. 117
Quick Test. 117
Console Style Screen Output 119
The background on console style I/O. 119
Displaying information with console output functions. 119
Accepting data using console input functions. 119
An overview of other console I/O functions. 120
What needs to be done to use console input and output. 121
Examples. 121
Quick Test. 121
Memory Models 123
The background on memory models. 123
The memory models available. 123
The effect of memory models on a programs size and speed. 124
Quick Test. 124
Passing Information Via the Command Line 125
The background on command line arguments. 125
The syntax for using command line arguments. 125
Error trapping and implementing command line arguments. 125
Examples. 126
Quick Test. 126
Execution of Child Processes 127
The general information on child processes. 127
The functions used to evoke a child process. 127
What needs to be done to use functions that evoke a child process. 128
Examples. 129
Quick Test. 129
Working with DOS 131
Utilizing environmental variables. 131
Accessing the system time. 131
Working with file level functions. 133
Handling directories from C. 133
Other DOS level functions. 135
What needs to be done to use DOS level functions. 136
Examples. 136
Quick Test. 136
Interrupt Processing 139

General overview of interrupt processing. 139
Calling DOS interrupt services. 140
Calling Other interrupt services. 140
Hooking your code onto an interrupt vector..... 141
What needs to be done to use interrupt processing functions. 141
Examples. 141
Quick Test. 141
Libraries and Object Modules 143
Breaking your programs into object modules. 143
Building a reusable library of functions. 144
General rules about organized development. 144
Using non-standard libraries and object modules. 144
Quick Test. 145
Make Files 147
A general overview of dependencies and dependence processors. 147
Examples. 147
Quick Test. 148
Binary Graphics 149
The background on binary graphics. 149
Manipulating video with binary graphics functions. 149
What needs to be done to use binary graphics functions..... 151
Examples. 151
Quick Test. 152
Port Level Processing 153
The background on port level processing..... 153
Manipulating peripheral equipment with port level functions. 153
What needs to be done to use port level I/O..... 153
Examples. 153
Quick Test. 154
A First Look at C++ and Objects 155
The basic differences between C and C++. 155
The concept of classes and objects. 155
Examples. 156
Quick Test. 156
MS-Windows Programming Using C 157
The process of developing a MS-Windows program. 157
The fallout of SDK programming. 157
Examples. 158
Quick Test. 158
The Future of C 159
C vs. C++. 159
C/C++ vs. the "wonder tools". 159
Making the "wonder tools" really work. 159
Quick Test. 159

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Part One - Before Programming Starts

& int .obj

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Introduction to C

The history of the C programming language.
- C was developed in the 1970's.
- Dennis Ritchie is credited as the creator of the C language.
- C was originally based in UNIX on a DEC PDP-11.
- C was derived from a language called B (an offspring of a language call BCPL).
- The only standard (unofficially) C was the UNIX 5 version.
- As the popularity of microcomputers increased so did the usage of C.
- Even with no real standard the applications written were surprisingly compatible.
- To resolve the remaining incompatibly ANSI in 1983 developed the standard for C.
- The ANSI standard allows portability between environments.

C compared to other programming languages.
- C is a middle level language.
- C mixes the power of high level languages with the capabilities of assembler.
- High level languages include COBOL, Pascal, FORTRAN, BASIC, and Ada.
- Although C is a higher level than macro assembly it shares the middle level with it.
- Assembly language is a low level language.
- C is an infinite language and is expanded with the use of functions.

C viewed as a structured language.
- C is considered a structured language.
- C is a structured language because of its modular implementation style.
- The rules of scope (global and local) are applied in C.
- C allows the use of code groupings within functions.
- C does not have a field concept and allows complete user formatting of code.
- C is not a block structured language and does not allow functions within functions.
- C allows branched control by using label and goto statements (not recommended).
- Other structured languages include Pascal, Ada, and Modula-2.
- Non-structured languages include COBOL, FORTRAN, and BASIC.

C's place in today's development environment.
- C was developed by programmers and is a programmers' language.
- Other languages were developed to be understood by non-programmers.
- C has very few restrictions and allows programmers to do what they want.
- C can be a dangerous language and requires a strong command of the language.

- C was used to code OS's, interpreters, editors, compilers, and database engines.
- C allows programs to be broken into smaller more manageable source files.
- C is a very strong language for libraries development.
- C can also be used as an effective system development language.
- C can generate object code that is nearly as efficient as assembly code.
- Many 4GL solutions would not be a solution if it weren't external C calls.

Common first reactions to C.

- I love it, or I hate it.
- I have never needed to be so aware of storage and addresses.
- Some things look very complex and cryptic.
- There are so many ways to do the same thing.
- I can't find a pattern to the usage of addresses.
- The compiler didn't say anything was wrong.
- My program appears to work fine but the output is not correct.

Challenges facing C students.

- C has a very steep learning curve.
- Students must develop a knowledge of how storage and addresses work.
- C requires students to know more about their programming environment.
- Students need to know more about the steps in a high level process.
- C requires students to invest more time in learning external libraries.
- Students will only become effective if they practice often.
- Students that know C need to deal with the lacks of other languages.

Quick Test.

1. Who developed the C language?
2. What are some common uses for C?
3. How is C valuable for 4GL's?
4. Is C more similar to COBOL or Macro Assembly?
5. What organization made the C language more portable?
6. Does C allow for local variables or scope?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Authoring and Executing a C Program

Coding a C application.
- This stage is where the programmer creates the C source code.
- Most C compilers come with a programming environment that has a built in editor.
- Many C programmers prefer more robust external editors such as Brief.
- C is case sensitive; 'x' is not the same as 'X'.
- ANSI C has very few keywords including auto, break, case, char, const, continue default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, and while.
- Common non-ANSI keywords include far, interrupt, near, and pascal.
- Mostly all work in C is completed with the use of function calls.
- All applications must contain an execution starting point or primary function.
- All C applications must contain a primary function called main().
- The word main is not part of the C language but is expected by the link editor.

Storing the application source code.
- This stage saves the source code on disk.
- C source code files are stored as text files, and end with the extension ".C".
- Saving the source code is not always an automatic event in all environments.
- For an application to be reproduced the original source code must be preserved.
- Saving the source code is the connection point for version management.
- Common errors include disk errors, overwriting files, and misplacing files.

Converting the application source code to object code.
- Programming languages fall into two categories: compiled and interpreted.
- Compiled languages are executed as load code and require no translation overhead.
- Interpreted languages are executed as instructions to an interpreting translator.
- The C language is a compiled language.
- C source code must be processed by a compiler; this process is called compilation.
- The output of compilation is an object file.
- Object file contains load code without external function calls and has a file name with the extension of ".OBJ".
- Errors found during compilation or compile time are called compilation errors.
- Compilation errors come in two forms: warnings and errors.
- Warnings are problems found that don't prevent the creation of an object module.
- Errors are problems found that prevent the creation of an object module.
- Errors are normally syntax related, and warnings are normally type related.
- Programs should compile without errors or warnings.

Binding the application object code to form an executable image.
- An object module is not in a form that is recognized by a command interpreter.
- External calls need to be linked with the OBJ's to form an executable image.
- Linking is done by a linkage editor or linker; this process is called linking.
- Output of a linker is called an executable, and has an extension of ".EXE".
- Errors found during linkage are referred to as linkage errors.
- Linkage errors are problems with external references and duplications.

Submitting the application image for processing.
- Executables or EXE are given to the OS's command interpreter.
- The OS loads the EXE from disk and starts it execution.
- Errors found during execution are referred to as run time errors.
- Run time errors are logic problems; what you told C to do was incorrect.
- Run time errors can cause the computer to require rebooting.

The dangers of not following the order.
- Source code can be lost.
- The EXE may be out of date and not reflect the latest changes.

Quick Test.
1. A syntax error will be found during what process?
2. Is 'O' the same as 'o' in C?
3. Is it true that most work in C is done with keywords?
4. How do you generate object modules?
5. Source code is stored in what format?
6. Is it true that you should never have compilation warnings?

Assignment.
- Type in and run the following program.

```
/* Classic hello world program.
   This program is intended to build a level of comfort
   with C compilation.
   by - Sergio C. Carbone
*/
#include <stdio.h>      /* Include standard I/O. */
void main()             /* The start of the program.  The
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
                              main function. */
{  puts("Hello World"); /* Call standard I/O function to
                              print string. */
}                       /* End of the main function. */
```


Program Layout

The basic syntax and layout of a C program

- Comments can be added to the source code.
- "/*" is used to start a comment and "*/" is used to end a comment.
- Comment in C wrap lines from the starting "/*" to the ending "*/".
- Never nest comments.
- All statements end with a ';'.
- Groups are not statements and do not normally end with ';'.
- Groups are started with '{' and are ended with '}'.
- For easy reading try to line up the start and end of a group in one column.
- Scope or local data is declared after the starting '{' of a group.
- The following is a shell that shows the basic layout of a C program.

```
/* Program title information comment
Program name: Demo
...
*/
#include <stdio.h> /* External inclusions */
int x = 0;         /* Global declarations */
void func1();      /* Function prototypes */
void main()        /* Primary function */
{  func1();
  x = 6;
  ...
}
void func2(); /* hidden prototypes */
int y = 0;    /* hidden declarations */
void func1()  /* Function body */
{ func2();
  ...
}
void func2()  /* Function body */
{  y = 5;
  ...
}
```

Program information comment.

- Standard comment syntax "/* ... */".
- Used to identify the program.
- Should appear at the top of the module.
- Should contain a least program description, list of functions, external requirements.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

External inclusions.
- #include compiler directive is used to include a file into the source during compilation.
- Lines compiled in the included file are added to the line count.
- #include <filename> will look in the compiler include path only for the file.
- #include "filename" will look first in the working directory then in the include path.
- Include files called header files normally have the extension of ".h"
- Include files are collections of external function prototypes and declarations.
- You should include the appropriate header files in order to use external functions.

Global declarations.
- Data and types that are intended to be used throughout the module.
- All data and types need to be declared before they are used.

Function prototypes.
- A prototype is a description of a function, its arguments, and return value.
- All functions should have prototypes so the compiler will generate proper call code.
- Prototypes are just a copy of the function header that ends with a semicolon ';'.
- The prototype must match the real function header.
- Prototypes should be placed above main().

Primary function.
- The primary function is the starting place for execution.
- In C this function is called the main function because its name is main().
- All C applications need to have only one main().
- The return values available for the main function are integer or void.
- The integer return value sets the OS error level.
- Two arguments can be passed to the main() for command line arguments.

Hidden declarations and prototypes.
- Like global declarations and prototypes except they are hidden from some code.
- Hidden declarations use a concept called data hiding.
- Data hiding is implemented by placing a declaration after non-privileged functions. Since the non-privileged function is compiled before the declaration is declared the non-privileged functions can't use the declaration. The declaration is hidden from the non-privileged functions.

Function bodies.

- The code statements of your functions.
- In C the function body should be placed below the main().
- Beginner C programmers place the function bodies above the main() like Pascal.
- The function header on the body should match the function prototype.
- The function code statements are placed in a code grouping after the header.
- The code grouping is started with '{' and ended with '}'.
- A shell of a prototype and body follow.

```
void myfunc(); /* prototype */
void myfunc()  /* header */
{ /* start of code group */
  /* code statements go here */
} /* end of code group */
```

Quick Test.

1. Should you nest comments in C?
2. How do you include prototypes from external functions?
3. What do function bodies contain?
4. Where should prototypes be placed?
5. If global data is placed after some functions that data is ____ to the functions.
6. What is the primary function in C?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Data Storage in C

A quick look at the way memory works.

- Memory to a program is like space to matter.
- Things occupy space, types do not.
- Many things in a program occupy memory including code, constants, and data.
- The amount of memory taken up is the size of the thing stored.
- All things in a program have a location referred to as an address.
- Nothing can occupy more than one location at one time.
- If things are copied, then they are two different things and are in separate locations.
- A variable or function name is just a label that references an address.
- Many people think of memory in a flat model.
- A flat memory model looks at the computers memory as one long piece.
- Memory on a PC with DOS is not a flat model; it is a segmented model.
- A segmented model divides the computers memory into segments making a matrix.
- A segment is then divided into bytes or offsets.
- The biggest size of a segment is 64K (1024 byte * 64).
- Segmentation is a great limitation on DOS programmers.

The main areas of memory.

- A program uses segments as areas to hold four different types of data.
- The program code is placed in an area called the code area or code segment.
- The global data is placed in an area called the data area or data segment.
- The stacked or temporary data is placed in the stack area or stack segment.
- The remaining memory is used as an area called the heap.
- The code and data areas will stay fixed from the result of the linkage editor.
- The stack and the heap will constantly change level and content.
- The way a program arranges these areas depends on the memory model it uses.
- The layout for a simple one segment model follows.

```
[The stack area] The stack grows down.
[The heap area ] The Heap grows up.
[The data area ] The data area is fixed.
[The code area ] The code area is fixed.
```

How addresses play a role in C programming.

- All data has an address and contains some value. For some data that value is an address. This type of data is called a pointer. Just remember that a pointer is just another piece of data; it too has an address and a value.
- For any data you need only remember two simple rules.
  - To change a thing you must know were it is (its address) not its value.
  - To print or compare a thing you must know its value not its address.

Quick Test.

1. Which areas of memory change in level and content at run time?
2. Is the memory on a PC with DOS flat or segmented?
3. What are the areas in memory?
4. When do you need the address of something?
5. How big can a segment be?
6. Which occupies memory: things, types, or both?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Built in Data Types

The character data type.
- Although it can be used for mathematical implementations, it is normally used for storing integer style numbers that represent ASCII characters.
- Its size is one byte or eight bits.
- The range for a char is -128 to +127, and an unsigned char is 0 to 255.
- Storing ASCII characters needs a range of 0 to 255 so unsigned char should be used.
- The syntax for allocating a char follows.

```
char x = 0; /* this allocates a single char called x
               and sets its value to 0 */
unsigned char z = 0; /* this allocates an unsigned
                        char called z and sets it to
                        0 */
```

The integer data type.
- The integer is the counting type. It can be used in mathematical implementations, and is also used as a Boolean type.
- It is available in two sizes short and long.
- It is two bytes or sixteen bits as an int, and four bytes or thirty-two bits as a long int.
- The range for an int is -32768 to +32767, and an unsigned int is 0 to 65535.
- The range for a long int is -2147483648 to +2147483647, and an unsigned long int is 0 to 4294967295.
- The syntax for allocating an int follows.

```
int x = 0; /* this allocates an int called x and
              sets it to 0 */
unsigned int y = 0; /* this allocates an unsigned
                       int called y and sets it to 0
                    */
long int z = 0; /* this allocates a long int called
                   z and sets it to 0 */
unsigned long int a = 0; /*this allocates an
                           unsigned long int
                           called a; sets it to 0*/
```

The float data type.
- The float stores floating point numbers and can be used in mathematical implementations, but it is better to use double.
- It is four bytes or thirty-two bits.
- The range for a float is 3.4E-38 to 3.4E+38.
- The syntax for allocating a float follows.

```
float x = 0.0; /* this allocates a float called x
```



```
                                and sets it to 0 */
```

The double data type.

- The double stores floating point numbers and is used in mathematical implementations.
- It is eight bytes or sixty-four bits.
- The range for a double is 1.7E-308 to 1.7E+308.
- The syntax for allocating a double follows.

```
    double x = 0.0; /* this allocates a double called x
                       and sets it to 0 */
```

The void data type.

- Void is used to state that a function has no return value or arguments.
- You cannot allocate a variable of type void.

Pointer data types.

- Pointers come in two sizes: near (two byte in size) used to point to data in the current segment, and far (four bytes in size) used to point to data in a different segment.
- All data of any type has an address, and a pointer to that address type can be made by combining the type name with the '*'.
- A void pointer is a generic pointer.
- The '*' tells the compiler that you want storage for a pointer to point to that type.
- The following example declares a pointer to each type and sets each pointer to null.

```
    char* x = NULL;
    unsigned char* y = NULL;
    int* z = NULL;
    unsigned int* a = NULL;
    long int* b = NULL;
    unsigned long int* c = NULL;
    float* d = NULL;
    double* e = NULL;
    void* f = NULL;
```

- The following example declares a far pointer to each type and sets each pointer to null.

```
    char far* x = NULL;
    unsigned char far* y = NULL;
    int far* z = NULL;
    unsigned int far* a = NULL;
    long int far* b = NULL;
    unsigned long int far* c = NULL;
    float far* d = NULL;
    double far* e = NULL;
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
     void far* f = NULL;
```

- NULL has the value of hex 0.
- To get the address of a piece of data, use the '&'.
- To get the value of the data the pointer points at, use the '*' to deference it.

```
int x = 0; /* make an integer called x and set it to 0 */
int* x_ptr = NULL; /* make an integer pointer called
                      x_ptr and set it to NULL */
x_ptr = &x;  /* assign x_ptr to the address of x (use the
                & to get the address) */
*x_ptr = 50; /* assign the value of what x_ptr points at
                to 50.  This changes the value of x to be
                50 (use the * to deference x_ptr and deal
                with the integer value) */
```

- Remember pointers are data. They just happen to store an address as a value.

Getting the size of data.

- To get the size of any data, use the sizeof keyword.
- sizeof returns the size of the data passed as an integer.
- The following is an example of the sizeof keyword.

```
     int x = 0; /* An integer to hold the return value of
                   sizeof */
     x = sizeof(float); /* sizeof used with the type name
                           float returns 4 */
     x = sizeof(x); /* sizeof used with the variable x
                       returns 2 because x is an integer
                       (note the wrapping comment) */
```

Quick Test.

1. What is the difference between a near and far pointer?
2. What is better to use in calculations: float or double?
3. To store the ASCII character set, which data type should be used?
4. What does the '&' do?
5. What is the value of NULL?
6. What kind of data does a void pointer point to?

Arrays in C

A general description of what an array is.
- An array is a sequential block of type similar and related data.
- Arrays can be made of any data type.
- Arrays are fixed in size; this size is set at compile time.
- Arrays can have multiple dimensions.
- When working with multiple dimension arrays, storage becomes a concern.

The notation used to work with arrays.
- To create an array, or reference a subscript, the "[]" are used.
- The format is "data_type variable_name[first dimension size] = initialization list;".
- Some examples of declaring arrays follow.

```
int x[5];
int y[4] = {3,66,54,77};
int z[3][5] = {{1,2,3,4,5},{11,22,33,44,55},
               {111,222,333,444,555}};
/* x is an array of five integers not set to
   anything.  y is an array of four integers set to
   values.  z is a double dimensioned array set to
   values. */
```

- Arrays start at zero, so an array of 3 integers is subscript 0 to subscript 2.

```
int x[3];
x[0] = 1;
x[1] = 2;
x[2] = 3;
```

- Without a subscript, the array returns the starting address of the array.
- Sizeof can be use to get the number of elements in a dimension.

```
int x[5];
int nos_elements;
nos_elements = sizeof (x) / sizeof(x[0]);
```

Null terminated arrays of characters better known as strings.
- C does not support strings like other languages.
- Strings are logical implementations of character arrays that end with NULL.
- NULL tells string functions when they have reached the end of the string.
- When allocating a string, you must remember to leave one space for the NULL.

```
char x[51]; /* x is 51 bytes and can hold 50
               characters and a NULL. */
char y[5][21]; /* y is 105 bytes and can hold 5 20
                  character NULL terminated strings
               */
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
     char z[11] = "ABCDEFG"; /* z is 11 bytes and can
                                hold a 10 character NULL
                                terminated string.  It
                                has been set to "ABCDEFG"
                             */
     char a[3][11] = {"Red","Green","Blue"};
     /* a is 3 10 character NULL terminated strings.
        It has been set to a[0] == "Red",
        a[1] == "Green", a[2] == "Blue"  */
```

- C does no run time space checking and will always write the NULL even if it overwrites other data in memory.
- To assign, copy, compare, or do anything with strings, they must be handled byte by byte. There are ANSI string functions available to make this easier.

Quick Test.

1. What happens if you don't supply a subscript?
2. What are the subscripts for a 5 element array?
3. What is a string?
4. How many characters does a 10 element char array hold?
5. What happens if you copy a string bigger than the receiving array?
6. What will we use more of as we increase the dimensions of an array?

Assignment.

- Type in and run the following program.

```
/* Type sizes program.
   This program is intended to build a level of comfort
   with C compilation.
   by - Sergio C. Carbone
*/
#include <stdio.h> /* Include standard I/O. */
void main()        /* The start of the program.  The
                      main function. */
{ printf("char %i\n",sizeof(char));
  /* Standard I/O function printf prints sizeof char.*/

  printf("int %i\n",sizeof(int));
  /* Standard I/O function printf prints sizeof int.*/

  printf("long %i\n",sizeof(long));
  /* Standard I/O function printf prints sizeof long.*/

  printf("float %i\n",sizeof(float));
  /* Standard I/O function printf prints sizeof float.*/

  printf("double %i\n",sizeof(double));
  /* Standard I/O function printf prints sizeof double.*/
} /* End of the main function. */
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Introduction to Functions

The reason for functions in C.
- All programming activity takes place in functions.
- Even the program starting point main() is a function.
- C has a very small keyword set that can not perform complex tasks.
- All complex code is done through the use of added functions.
- Functions are used to extend the capabilities of C.
- Functions are used to organize groups of logically related code.
- Modular programs are much easier to write and maintain.

The main sources for functions.
- The first place to look is the standard library. This comes with the environment.
- Mostly all program specific function will be written in-house.
- Very complex functions are sometimes given to library development groups.
- Library development groups are normally in-house teams.
- Often functions that serve a generic role are purchased from software vendors

The layout of functions in C.
- A function has four main areas: the name, return type, arguments, and code group.

```
return-type name(arguments)
{ code group
}
```

- The function's code group has three areas: declarations, code statements, and return.

```
int test_function()
{ int x = 0; /* local declaration */
  x = x + 1; /* code statement */
  return x;  /* return */
}
```

- The code grouping is started with '{' and ended with '}'.
- Local declarations and arguments are scope data, and their values are valid only until program control leaves their scope.
- Local declarations and arguments are only available to the function.
- Functions should have only one exit or return point.
- Functions should be relatively short and focused.

The function prototype.
- All functions should have a prototype.
- The function header on the body should match the function prototype.
- If a prototype is incorrect, the compiler will not create the call properly.
- If no prototype is given, the compiler assumes the function returns an int.
- A shell of a prototype and body follow.


```
void myfunc(); /* prototype */
void myfunc()  /* header */
{ /* start of code group */
  /* code statements go here */
} /* end of code group */
```

The function arguments.

- C can pass data values to functions.
- C has no true pass by reference capability.
- Pass by reference can be simulated by passing an address, and using that address to change the value of what it points at.
- All data passed to a function uses stack space.
- Arguments are declared and placed within the parentheses.
- Arguments are separated by commas.
- If no arguments are to be passed, place a void within the parentheses.
- An example of arguments follows.

```
void func(int x, int* a_ptr) /* passed an int and an
                                int pointer */
{ x = 0;      /* x is a copy of the int s; this
                 assignment has no effect on s*/
  *a_ptr = 7; /* Using the address passed as a_ptr
                 we can change the value of t */
}
void main()
{ int s = 1;  /* int s and t */
  int t = 1;
  func(s,&t); /* pass the value of s and the address
                 of t */
}
```

- The syntax of arguments is just like other data.
- Don't panic, the function tells you exactly what it wants to be passed.
- Just give it the types of data it needs.

The return value.

- The return value is the data the function returns to the calling body.
- The return value is used to give an outcome of the function call.
- Return values are often used to return error status codes.
- A function can have a return value of void, meaning it has no return value.
- A function can have only one return value.
- To use a return value, set the return type to the type of data to be returned.
- At the end of the function, use the return statement with the data to be returned.
- In void functions, return is not given a value, and causes an immediate return.
- An example of return values follows.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
int func(int x) /* passed an int */
{ return x * 2; /* we return x * 2 */
}
void main()
{ int t = 1;
  t = func(t); /* pass the value of t and store the
                  return in t */
}
```

Quick Test.

1. What is a common use for a return value?
2. What is used to separate arguments?
3. Does a function need to return a value?
4. How would a function return an int value?
5. Can addresses be passed to functions?
6. Why are functions needed in C?

++ if

for

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Standard Input and Output

The background on standard I/O.
- Standard I/O is supplied with all ANSI C environments.
- The functions in standard I/O are meant to work on all platforms.
- Standard I/O is too limited for most PC style systems.
- Even with its limits standard I/O provides a good basic functionality.
- Most professional systems use third party libraries for I/O.
- Standard I/O provides a basis for portable interfaces.
- Standard I/O is most commonly used for debug diagnostics and error messages.
- Standard I/O helps many beginning students learn the rules of function arguments.

What is meant by the stream concept.
- A stream is a logical, device independent way of handling many peripheral devices.
- Most peripheral devices are different and would require special programming techniques.
- Streams mask this difference from the programmer and allows the programmer to handle most peripheral devices in the same way.
- Streams are buffered interfaces that are thought of as a series of characters read one at a time.
- PCs are based on a stream architecture.

A word on operating system redirection.
- The functions in this section use the standard input and standard output streams.
- Standard input would normally come from a keyboard on a PC.
- Standard output would normally write to the screen on a PC.
- The OS can redirect standard input and standard output to come from almost any peripheral.
- The user can tell the OS to redirect standard input by using '<'.
- The user can tell the OS to redirect standard output by using '>'.
- OS redirection is a useful capability when dealing with batch processes.

Displaying information with standard output functions.
- There are a wide variety of functions used to write information in standard I/O.
- All the output functions in this section write to standard output.
- int putchar(int c);
  - putchar(c) is a macro to write c to standard output.

  - On success, putchar returns c. On error, putchar returns EOF.
  - A short example of putchar follows:

```
char x = 'A'; /* Make a char called x. */
putchar(x);   /* Use putchar to print it on the
                 screen. */
```

- int puts(const char *s);
  - Puts writes the string s to standard output and translates the NULL to a newline.
  - On success, puts returns a newline "\n" value. Otherwise, it returns EOF.
  - A short example of puts follows:

```
char x[5] = 'ABCD'; /* Make a string called x. */
puts(x);            /* Use puts to print it on the
                       screen. */
```

- int printf(const char *format[, argument, ...]);
  - Printf writes a series of arguments formatted as specified by format to standard output.
  - The format string is a series of codes and text.
  - The codes for the format string start with a '%' and end with a character.
  - The format codes used in prinf follow:

    %d Signed decimal integer.
    %i Signed decimal integer.
    %o Octal integer.
    %u Unsigned decimal integer.
    %x Hexadecimal int (with a, b, c, d, e, f).
    %X Hexadecimal int (with A, B, C, D, E, F).
    %f Decimal point format.
    %e Scientific notation.
    %g use the shorter of %f or %e.
    %E Same as e, but with E for exponent.
    %G Same as g, but with E for exponent if %e format is shorter.
    %c Single character.
    %s String pointer.
    %% Prints the % character.
    %p Prints the input argument as a pointer.
    %n Stores number of chars written so far in the location pointed to by the argument.

  - The format codes also have modifiers that refine the output.
  - *% [flags] [width] [.prec] [F|N|h|l|L] type_char* is the format used for modifiers.
  - Flags modify the output by changing justification, numeric signs, decimal points, trailing zeros, octal and hex prefixes.
  - The available flags follow:

    '-' Left-justifies, and pads on the right with blanks. If not used, it right-justifies, and pads on the left with zeros or blanks.
    '+' Always begins with a plus or minus sign.
    ' ' Only negative values have sign.
    # Conversion using an alternate form.
      With c, s, d, i, or u there is no effect.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

With o, o is prepended to arg.
With x, or X 0x (or 0X) is prepended to arg.
With e, E, or f always has a decimal point.
With g, or G it is same as e, or E, except that trailing zeros are not removed.

- Width modifies the minimum characters to print, padding with blanks or zeros.
- Precision modifies the number of decimal places to print for float's and double's. For int's and strings it modifies the maximum number of characters to print.
  - Input size modifies default size of next input argument:
    N= near pointer
    F = far pointer
    h = short int
    l = long
    L = long double
  - There must be enough arguments to match each entry in the format string. If there are not, the program may crash. Extra arguments are ignored.
  - On success, printf returns the number of bytes written. On error, it returns EOF.
- A short example of printf follows:

```
char x[5] = 'ABCD'; /* Make a string called x. */
printf("The string is %s.\n",x); /* Use printf to
                                    print it on the
                                    screen. */
```

Accepting data using standard input functions.

- There are a wide variety of functions used to input information in standard I/O.
- Before calling any standard I/O input function, it is best to flush the input stream.
- Use fflush(stdin) to flush the standard input stream.
- int fflush(FILE *stream);
  - Flushes a stream.
  - If output, fflush writes the output for stream to the file.
  - If input, fflush clears state codes from the input stream.
  - fflush returns 0 on success. It returns EOF if any errors were detected.
  - A short example of fflush follows:

```
fflush(stdin); /* Flush the input stream. */
```

- All the input functions in this section read from standard input.
- int getchar(void);
  - Getchar is a macro that returns the next character from standard input.
  - On success, getchar returns the character. On error, it returns EOF.
  - A short example of getchar follows:

```
char x =0; /* char called x. */
puts("enter a char."); /* print a prompt. */
fflush(stdin);    /* Flush the input stream. */
do                /* Start of loop */
{ x = getchar();  /* Use getchar to input a char. */
} while (x==EOF); /* Loop until x is not EOF */
```

- char *gets(char *s);

- Gets reads a string of characters from standard input, and puts it into s.
- Gets allows strings to contain spaces, and tabs. Gets returns when it encounters a new line; everything up to the new line is copied into s.
- The new line is replaced by a NULL in s.
- On success, gets returns s. On error, it returns NULL.
- A short example of gets follows:

```
char x[256]; /* String called x. */
puts("enter your full name."); /* print a prompt. */
fflush(stdin);    /* Flush the input stream. */
gets(x);          /* Use gets to input a string
                     from stdin. */
```

- int scanf(const char *format[, address, ...]);
  - Scanf reads a series of inputs in to arguments formatted as specified by format from standard input.
  - The format string is a series of codes and text.
  - The codes for the format string start with a '%' and end with a character.
  - The format codes used in prinf follow:

%d	Pointer to int (int *arg).
%D	Pointer to long (long *arg).
%e,E	Pointer to float (float *arg).
%f	Pointer to float (float *arg).
%g,G	Pointer to float (float *arg).
%o	Octal integer - Pointer to int (int *arg).
%O	Octal integer - Pointer to long (long *arg).
%i	Decimal, octal, or hex - Pointer to int (int *arg).
%I	Decimal, octal, or hex - Pointer to long (long *arg).
%u	Pointer to unsigned int (unsigned int *arg).
%U	Pointer to unsigned long (unsigned long *arg).
%x	Hexadecimal integer - Pointer to int (int *arg).
%X	Hexadecimal integer - Pointer to int (int *arg).
%s	String - Pointer to array of chars (char arg[]).
%c	Pointer to char (char *arg).
%%	No conversion done; the % is stored.
%n	The number of characters read successfully up to %n.
%p	Hexadecimal form - Pointer to an object (far* or near*).

  - The format codes also have modifiers that refine the input.
  - *% [*] [width] [F|N] [h|l|L] type_char* is the format used for modifiers.
    [*] - Assignment-suppression character. Stops assignment of the next input field.
    [width] - Width specifier. Specifies maximum number of characters to read.
    [F|N] - Pointer size modifier. N=near pointer, F=far pointer.
    [h|l|L] - If int h=short int, and l=long int. If float l=double, and L=long double.
  - Scanf often leads to unexpected results if you diverge from the expected pattern.
  - Gets is better to use than scanf.
  - There must be enough arguments to match each entry in the format string. If there are not, the program may crash. Extra arguments are ignored.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- On success, scanf returns the number of input fields successfully stored. On error, scanf returns 0 or EOF.
- A short example of scanf follows:

```
char x[256]; /* String called x. */
puts("enter your first name."); /* print prompt. */
fflush(stdin); /* Flush the input stream. */
scanf("%s",x); /* Use scanf to input a string
                  from stdin. */
```

What needs to be done to use standard input and output.

- Just include stdio.h using a #include <stdio.h> at the top of the program.
- Link with the standard library when building the .EXE.

Examples.

- See example disk for program STDIOEXM.C

Quick Test.

1. How does scanf differ from gets?
2. What does fflush do?
3. What does %s mean in a printf?
4. What is standard I/O commonly used for?
5. Does scanf have any problems?
6. Which function reads spaces and tabs gets or scanf?

Assignment.

Write a program that:

- Declares variables of five different data types int, float, long, double, and string.
- Reads data from the keyboard for each variable.
- Prints the value entered for each variable to the screen.
- Prints the size of each variable to the screen.

NOTE:

- All data should be global data.
- The program should contain four functions.

main() - should call other functions.
inputdata() - should input values.
outputvalues() - should output values.
outputsizes() - should output sizes.

- This program should be no more than two pages.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Tips for Using Data Storage

Initializing your data storage.

- Eighty percent of run time bugs are caused by non-initialized data storage.
- Always initialize data storage, even if it is being set directly after it is declared.
- Remember programs change, and what works today may not work tomorrow.
- Try to use realistic values.
- Numeric data should at least be initialized to zero.
- Strings should at least be initialized to NULL strings.
- Pointers should at least be initialized to NULL.
- Pay close attention to pointers because they cause very complicated bugs.
- Some small examples of initializing storage follow:

```
int x = 0;         /* x is set to 0. */
char n[51] = "";   /* n is set to a NULL string. */
double* z = NULL; /* z is set to NULL. */
```

The impact of function arguments and local variables on performance.

- Function arguments and local variables are stacked data.
- Pushing this information on the stack does take a small amount of processing time.
- The fewer the arguments and local variables a function has, the faster it executes.
- If a function has many arguments, it may be better to package their addresses in a structure, and pass the address of the structure.
- Passing data is important for good design and easy maintenance, but do not pass arguments just for the sake of passing.

The impact of global variables on code complexity and reuse.

- Global variables are data segment variables.
- The more information you place on the data segment the larger the application is.
- Data segment space is limited in some memory models.
- A function that uses global variables is tied to that program, and can not be used elsewhere without considerable coding.
- Using a large number of global variables make a program difficult to understand.
- The use of global variables makes a program difficult to debug, because it is difficult to determine what functions use and modify the global variables.
- Do not over use global variables.

Quick Test.

1. Which runs more quickly, a program with global data, or arguments?
2. Why use arguments and local variables?
3. Which is smaller, a program with global data, or arguments?
4. Why not make all variables global variables?
5. Why should variables be initialized?
6. What are some guidelines for initializing variables?

Assignment.

Write a program that:

- Declares variables of five different data types int, float, long, double, and string.
- Reads data from the keyboard for each variable.
- Prints the value entered for each variable to the screen.
- Prints the size of each variable to the screen.

NOTE:

- All data should be local to main() and passed where needed.
- The program should contain four functions.
  main() - should call other functions.
  inputdata() - should input values.
  outputvalues() - should output values.
  outputsizes() - should output sizes.
- This program should be no more than two pages.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Conditional Execution

How true and false works in C.

- There is no formal Boolean data type in C.
- Integers are used to handle Boolean data.
- The result of any expression or function return can be tested for true or false.
- A comparison is not needed to test for true or false.
- In C, false is zero.
- True is any value other than zero.

The logical operators in C.

- The operators that are used for logical or relational expressions are:

()	Grouping precedence.
!	Not.
>	Greater than.
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.
==	Equal to.
!=	Not equal to.
&&	And.
||	Or.

Executing conditionally with an if statement.

- Conditional execution is most simply handled with an if statement.
- There are two formats for an if statement in C.
  - The single statement conclusion format.

```
if (expression)
  conclusion;
```

  - The multiple statement conclusion group format.

```
if (expression)
{ conclusion;
  ...
}
```

- "Then" is not used in C if statements.
- The expression is evaluated, and if it is "true" the conclusion is executed.

Trapping condition failure with an else statement.

- There are times when code needs to be executed only when an expression is false.
- An else statement can be added to an if statement to trap false expressions.
- There are two formats for an else statement in C.
  - The single statement conclusion format.

```
if (expression)
  conclusion;
else
  conclusion;
```

  - The multiple statement conclusion group format.

```
if (expression)
  conclusion;
else
{ conclusion;
  ...
}
```

- The expression is evaluated, and if it is "false" the else conclusion is executed.

General rules regarding if..else statements.

- The else is matched with the last if in its current scope.
- If statements do not need to have an else statement.
- Else statements must be immediately preceded by an if statement.
- The if conclusion and else conclusion will never both be executed.

Nesting if and if..else statements.

- Nesting is used when it is necessary to place an if or if..else statements as part of a conclusion.
- Nesting increases code complexity, but it is often required.
- Large multiple level nested if or if..else statements should be avoided by the use of function calls.
- Nesting causes problems with the matching of if..else statements. These problems can be overcome by using the multiple statement conclusion format.
- Some formats for nested if and if..else statements follow.

```
if (expression)   /* Simple nested if */
  if (expression) /* Matching if */
    conclusion;
  else            /* Matching else */
    conclusion;

if (expression)   /* Matching if */
{ if (expression) /* multiple statement conclusion*/
    conclusion;
  ...
}
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
else              /* Matching else */
  conclusion;
if (expression)   /* Match A */
  if (expression) /* Match B */
    conclusion;
  else            /* Match B */
    conclusion;
else              /* Match A */
  if (expression) /* Match C */
    conclusion;
  else            /* Match C */
    conclusion;

if (expression)
{ if (expression)
    conclusion;
}
else
  if (expression)   /* Else with if */
    if (expression) /* Simple nested if */
      conclusion;

if (expression)   /* Matching if */
{ if (expression)
    conclusion;
}
else              /* Matching else */
{ if (expression) /* multiple statement conclusion*/
    conclusion;
  ...
}
```

Executing conditionally with a switch..case statement.

- Switch..case statements can take the place of many if..else statements that test only for equality.
- Switch..case statements are cleaner and easier to read than nested if..else statements.
- The format for switch..case statements follows:

```
switch (variable)
{ case constant 1 :
    statements;
    ...
    break;
  case constant 2 :
    statements;
    ...
    break;
  case ...
  default:
    statements;
}
```


- The variable is compared to each constant.
- If a match is found, execution starts with the statements for that case, and continues until a break statement is processed.
- If a break is not processed, execution continues even into the next case.
- If no match is found, execution starts with the default case.
- Variables can be defined directly after the opening group symbol of a switch..case statement.
- A case does not need to have statements for it. Execution will start with the next statement found, and continue until a break is encountered.
- The switch variable must be convertible to an integer.
- The case values must be literals or constants that can be converted into an integer.
- The case values must be unique.
- Strings can not be used in switch..case statements.

The use of a default case.

- The default case is not required.
- If no matches are found, execution starts at the default case.
- If no matches are found, and there is no default case, execution continues after the switch..case statement.
- If a match is found before the default case, and no break statement is encountered, execution continues into the default case.

How the break statement is used with switch..case statements.

- The break statement is not required.
- The break statement forces execution to jump to the first statement after the switch..case statement.
- In most implementations, the break statement is used with each case.
- The break statement is not required in the very last case in a switch..case statement.
- The break statement can be conditionally executed, but this will produce confusing C code.

Altering values conditionally with the ?: operator.

- The ?: allows one of two values to be return based on a condition.
- The format of the ?: operator follows:

```
condition ? true expression : false expression;
```

- If the condition evaluates "true", the true expression is evaluated.
- If the condition evaluates "false", the false expression is evaluated.
- Unlike an if..else statement, the ?: operator can by placed within an expression.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Nesting the ?: operator.

- The ?: operator can be nested, but the syntax is very difficult to understand.
- Some short examples of the ?: operator follow:

```
int x = 0;
int y = 5;
int z = 10;
printf("%i", x ? y : z);              /* prints 10 */
printf("%i", y < z ? y : z);          /* prints 5 */
printf("%i", z ? x : y);              /* prints 0 */
printf("%i", x ? y : z > y ? z : x); /* prints 10 */
```

- The examples above show how nested ?: operators can be difficult to follow, and that example had only one nested ?:. The following only has two nested ?:.

```
printf("%i",x ? y < x ? y : z : z > y ? z : x);
/* ouch! */
```

Boolean short circuiting.

- Boolean short circuiting is a way of evaluating expressions with the fewest tests.
- Evaluation stops as soon as the expression is proven "true" or "false".
- "And" expressions need to find only one portion "false" to stop evaluation.
- "Or" expressions need to find only one portion "true" to stop evaluation.
- In the follow examples that right expressions would not be evaluated.

```
int x = 0;
int y = 5;
int z = 10;
if (x < 1 || z > y) ...
if ( x && z > y) ...
```

- If an expression is a function call or meaningful code, the function call or code would not be executed.
- Boolean short circuiting can lead to unexpected program execution.

Examples.

- See example disk for program CONDIEXM.C

Quick Test.

1. Does a switch..case need to have a default case?
2. What is the difference between && and ||?
3. How are else statements matched with if statements?
4. What does this return 5 > 6 ? 0 : 6 < 2 ? 3 : 5?

5. What does the break statement do for a switch..case?
6. How can Boolean short circuiting effect execution?

Assignment.

Write a program that:

- Inputs a person's full name and age.
- Prints the full name and what age group the person is in.
- The text for the age groups should be child, teen, or adult.

Note:

- The program should have all variables local to main.
- The program should have three functions:
  main() - calls inputdata and printdata.
  inputdata() - inputs user name and age.
  printdata() - tests age and prints result.

* Try the ?: operator.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Iterative Execution Control

An overview of iterative control.

- Looping is used to repeat a sequence of program statements multiple times.
- All loops in C have a condition expression that must evaluate to true for the loop to continue.
- The condition expression can be made from almost any combination of code. It must evaluate to a zero or non-zero value.
- In most loops the condition is tested before the first execution of the program statements takes place.
- In some loops the first execution of program statements takes place before the condition is tested.
- There are three styles of loops in C: a "for" loop, a "while" loop, and a "do..while" loop.
- The number of iterations a loop makes can be predetermined with a counter, or the loop can be controlled by some variable condition such as being at end of file.

Using a "for" loop.

- A "for" loop is thought of as the most controlled loop.
- The C "for" loop is similar to Pascal, BASIC, or many other languages.
- A "for" loop is commonly thought of as a counting loop.
- The basic syntax of a "for" loop has four main components: the initialization statement, the condition expression, the program statement or grouping, and the increment statement.

```
for (initialization; condition; increment)
  program statement;
for (initialization; condition; increment)
{  grouping
}
```

  - The initialization statement is normally used to set the starting value of the loop counter. It is only executed once and occurs before the loop starts.
  - The condition expression is evaluated at the start of every iteration. If it evaluates to true, the program statements are executed. If it evaluates to false, the loop is ended.
  - The program statement or grouping is the code that will be executed each iteration.
  - The increment statement is executed at the end of every loop and is normally used to increment the loop counter.
  - Normally the initialization statement, condition expression, and increment statement all work with a loop counter, but they do not need to be related in any way.

  - Although it is not recommended, the initialization, condition, or increment can be empty as long as the ';' is used as a place holder.
- A "for" loop does not need to have any program statements.
- An example of a "for" loop follows:

```
void func()
{ int x = 0;
  for (x = 0; x < 5; x = x + 1)
    printf("x = %d\n",x);
}
```

Using a "while" loop.

- A "while" loop is thought of as the simplest loop.
- The C "while" loop is similar to Pascal, BASIC, or many other languages.
- A "while" loop is not commonly thought of as a counting loop.
- The basic syntax of a "while" loop has two main components: the condition expression, and the program statement or grouping.

```
while (condition)
  program statement;
while (condition)
{  grouping
}
```

  - The condition expression is evaluated at the start of every iteration. If it evaluates to true, the program statements are executed. If it evaluates to false the loop is ended.
  - The program statement or grouping is the code that will be executed each iteration.
  - The program statement or grouping must have some affect on the condition or the loop will be infinite.
- A "while" loop does not need to have any program statements.
- An example of a "while" loop follows:

```
void func()
{ int age = 150;
  while (age >= 150)
  { puts("Input age:");
    fflush(stdin);
    scanf("%d",&age);
  }
}
```

Using a "do..while" loop.

- A "do..while" loop is thought of as the least controlled loop.
- A "do..while" loop will always execute at least once, and is not commonly thought of as a counting loop.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- The basic syntax of a "do..while" loop has two main components: the condition expression, and the program statement or grouping.

```
do
 program statement;
while (condition);
do
{  grouping
} while (condition);
```

  - The condition expression is evaluated at the end of every iteration. If it evaluates to true, the program statements are executed again. If it evaluates to false, the loop is ended.
  - The program statement or grouping is the code that will be executed each iteration.
  - The program statement or grouping must have some affect on the condition or the loop will be infinite.
- An example of a "do..while" loop follows:

```
void func()
{ int age = 0;
  do
  { puts("Input age:");
    fflush(stdin);
    scanf("%d",&age);
  } while (age >= 150)
}
```

Using the break statement with iterative control.

- The break statement can be used to stop a loop.
- The break statement forces execution to jump to the first statement after the loop.
- In most implementations, the break statement can be avoided.
- The break statement can be conditionally executed, but this will produce confusing C code.
- An example of the break statement follows:

```
void func()
{ int age = 150;
  for (;;)
  { puts("Input age:");
    fflush(stdin);
    scanf("%d",&age);
    if (age <= 150)
      break;
  }
}
```


Using the continue statement with iterative control.

- The continue statement can be used to skip to the "bottom" of a loop.
- In a "while" or "do..while" loop the continue statement causes execution to go directly to the condition.
- In a "for" loop the continue statement causes execution to go directly to the increment statement.
- In most implementations, the continue statement can be avoided.
- The continue statement can be conditionally executed, but this will produce confusing C code.
- An example of the continue statement follows:

```
void func()
{ int age = 150;
  do
  { puts("Input age:");
    fflush(stdin);
    scanf("%d",&age);
    if (age <= 150)
      continue;
    puts("Age cannot be greater then 150.");
  } while (age > 150);
}
```

Examples.

- See example disk for program LOOPSEXM.C

Quick Test.

1. Which loop executes at least once?
2. What is the difference between break and continue?
3. Does a "for" loop need to have an initialization statement?
4. Which loop is commonly used as a counting loop?
5. What does the condition evaluate to for the loop to end?
6. Can a "while" loop not have any program statements?

Assignment.

Write a program that:

- Inputs information on five people (name and age).
- Prints a report on the five people (the name and what age group the person is in).
- The text for the age groups should be child, teen, or adult.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- Prints the average age of the five people.
- The program should have two arrays local to main (one for names and one for ages).
- The program should pass the arrays to two functions (one to input the information on five people and one the print a report).
- The input information function should have a loop and call a function to input information about one person.
- The report function should have a loop and call a function to print output for one person. It should also print the average age.
- The program should test for realistic information.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Branch Control

A word on why not to read this section.

- The use of branch control, better known as the goto statement, has gotten a bad name in the programming world.
- The goto statement causes execution control to move directly to the location specified by the goto statement. Control does not return.
- The goto statement has a tendency to make programs difficult to understand.
- The goto statement makes programs more difficult to debug and maintain.
- There are no programming problems that require the goto statement.
- If you use the goto statement, other programmers may develop a poor impression of your capabilities.

How to implement code labels and goto statements.

- To implement a goto statement you must first have declared a code label.
- A code label is a passive statement that attaches an identifier to a location in a function.
- To implement a code label, just make up an identifier name, and place it in the function followed by a ':'.

```
void func()
{ int x = 0;
  startoffunc:  /* this the label startoffunc */
  ...
}
```

- To implement a goto statement that will go to that label, simply place goto followed by the label name.

```
void func()
{ int age = 0;
  startoffunc:  /* this the label startoffunc */
  puts("Input age:");
  fflush(stdin);
  scanf("%d",&x);
  if (x > 150)
    goto startoffunc; /* this goto forms a loop */
  ...
}
```

- The label must be in the same function.

Examples.

- See example disk for program GOTOSEXM.C

Quick Test.

1. What do you need in order to use a goto statement?
2. What does a goto statement do to execution control?
3. How does a goto statement impact code readability?
4. Are goto statements required?
5. What type of statement is a label?
6. Can the use of goto statement affect your programming career?

Assignment.

Write a program that:

- Inputs information on five people (name and age).
- Prints a report on the five people (the name and what age group the person is in).
- The text for the age groups should be child, teen, or adult.
- Prints the average age of the five people.
- The program should have two arrays local to main (one for names and one for ages).
- The program should pass the arrays to two functions (one to input the information on five people and one the print a report).
- The input information function should have a loop and call a function to input information about one person.
- The report function should have a loop and call a function to print output for one person. It should also print the average age.
- The program should test for realistic information.
- Do not use "for", "while", or "do..while" to loop. All loops must be made from goto statements.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Handling Basic Mathematics

The background on basic mathematics.

- Mathematics in C can be handled in three ways:
  - Using mathematical operators.
  - Using mathematical functions.
  - Using bit level operators.
- Equations that contain different types of values will go through a process known as type promotion.
- Type promotion is a process that converts a lower level type such as an int, to a higher level type such as a double.
- Types can only be promoted to higher levels; types are never demoted.
- In the following example x and y are used as integers and have a sum of three. The value three would be promoted to a double, and then multiplied by z.

```
int x = 1;
int y = 2;
double z = 3.5;
z = z * (x + y); /* z == 10.5 */
```

The mathematical operators.

- C has a wide variety of mathematical operators.
  - ++ The increment operator increases the value of a variable. Typically it will increment by one, but it will increment by type size when dealing with addresses. This operator alters the variable directly and is not equivalent to "x + 1".
  - -- The decrement operator decreases the value of a variable. Typically it will decrement by one, but it will decrement by type size when dealing with addresses. This operator alters the variable directly and is not equivalent to "x-1".
  - - The negation operator reverses the sign of a value, and does not alter the variable.
  - * Standard multiplication.
  - / Standard division.
  - % The modulo division operator is used to get the remainder from the division of two integers "1 == 10 % 3". Modulo division can only be used with integers.
  - - Standard subtraction.
  - + Standard addition.
  - += The assignment addition operator adds the value of the right operand to the left operand and stores the new value in the left operand.
  - -= The assignment subtraction operator subtracts the value of the right operand from the left operand and stores the new value in the left operand.
  - *= The assignment multiplication operator multiplies the value of the right operand with the left operand and stores the new value in the left operand.

/= The assignment division operator divides the left operand with the value of the right operand and stores the new value in the left operand.

- Some short examples of operator usage follow:

```
int a = 1;
int b = 5;
double c = 4.5;
a++;
--b;
++b;
a += b;
c /= b % (a + 2);
```

What is meant by prefix and post-fix notation.

- Both the increment operator and the decrement operator can be used in prefix or post-fix form.
- Prefix notation is implemented when the operator is placed before the operand.
- Before the operands value is used in the equation, the operand is modified.
- A short example of prefix notation follows:

```
int z = 0;
int g = 0;
g = ++z;
printf("g: %i  z: %i\n",g,z);
```

- In the preceding example one would be printed for both g and z.
- Post-fix notation is implemented when the operator is place after the operand.
- Only after the operands value is used in the equation, is the operand modified.
- A short example of post-fix notation follows:

```
int z = 0;
int g = 0;
g = z++;
printf("g: %i  z: %i\n",g,z);
```

- In the preceding example zero is printed for g, and one is printed for z.

Calculating information with math functions.

- Most C programming environments come with mathematical functions.
- The mathematical functions are broken into four specialties including trigonometric, hyperbolic, logarithmic, and general.
- If a mathematical function is passed an argument that is not appropriate for the functions, it returns a function specific error value and sets errno to EDOM.
- If a mathematical function results in an overflow, it returns HUGE_VAL and sets errno to ERANGE.
- If a mathematical function results in an under-flow, it returns a zero and sets errno to ERANGE.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- The availability of mathematical functions vary from environment to environment, but the following is a list of functions that are normally available:
  - To get the absolute value of a number, use abs().
    ```
    double abs(double arg);
    ```
  - To get the arc cosine of a radian (-1 to 1), use acos().
    ```
    double acos(double arg);
    ```
  - To get the arc sine of a radian (-1 to 1), use asin().
    ```
    double asin(double arg);
    ```
  - To get the arc tangent of a radian (-1 to 1), use atan().
    ```
    double atan(double arg);
    ```
  - To get the ceiling for a number (the ceiling of 2.5 is 3), use ceil().
    ```
    double ceil(double arg);
    ```
  - To get the cosine of a radian, use cos().
    ```
    double cos(double arg);
    ```
  - To get the value of the natural logarithm e raised to a number, use exp().
    ```
    double exp(double arg);
    ```
  - To get the floor for a number (the floor of 2.5 is 2), use floor().
    ```
    double floor(double arg);
    ```
  - To get the remainder of a division (the remainder of 4/3 is 1), use fmod();
    ```
    double fmod(double x, double z);
    ```
  - To get the length of the hypotenuse on a right triangle using the lengths of the other sides, use hypot().
    ```
    double hypot(double x, double y);
    ```
  - To get the natural logarithm of a number, use log().
    ```
    double log(double num);
    ```
  - To get the base ten logarithm of a number, use log10().
    ```
    double log10(double num);
    ```
  - To break a double into its integer and fractional components, use modf().
    ```
    double modf(double num, int* i);
    ```
  - To get the value of a number raised to a exponential power, use pow().
    ```
    double pow(double num, double exp);
    ```
  - To get the sine of a radian, use sin().
    ```
    double sin(double arg);
    ```
  - To get the square root of a number, use sqrt().
    ```
    double sqrt(double num);
    ```
  - To get the tangent of a radian, use tan().
    ```
    double tan(double arg);
    ```

What needs to be done to use math functions.

- Just include math.h using a #include <math.h> at the top of the program.
- Link with the standard library when building the .EXE.

The bit level operators.

- Bytes are made from bits, and bits are set to one for true or zero for false.
- Sometimes it is necessary to deal with the bits of a number.
- In many languages bit level control is not possible.
- C has a wide variety of bit-wise operators to allow good control.

& Bit level and. Both bits must be set to one for it to return a one.

```
char x = 1; /* 00000001 */
char y = 3; /* 00000011 */
x = x & y; /* x is set to 1 (00000001) */
```

| Bit level or. One or both bit must be set to one for it to return a one.

```
char x = 1;
char y = 3;
x = x | y; /* x is set to 3 (00000011) */
```

^ Bit level exclusive or. Only one bit can be set to one for it to return a one.

```
char x = 1;
char y = 3;
x = x ^ y; /* x is set to 2 (00000010) */
```

~ Bit level not. This operator reverses the setting of the bits. Zero's become one's and one's become zero's.

```
char x = 1;
char y = 3;
x = ~y; /* x is set to 252 (11111100) */
```

>> Shift right. This operator shifts the bits of a byte right.

```
char x = 1;
char y = 3;
x = y >> 1; /* x is set to 1 (00000001) */
```

<< Shift left. This operator shifts the bits of a byte left.

```
char x = 1;
char y = 3;
x = y << 3; /* x is set to 24 (00011000) */
```

Examples.

- See example disk for program MATHMEXM.C

Quick Test.

1. What function is used to separate the decimal and integer portion of a number?
2. What is the difference between % and fmod()?
3. What is the result of 7 ^ 3?
4. Where are the math function prototypes found?
5. What is the result of (8 / 2) == (8 >> 1)?
6. If x equals 5 what is the result of x++ * 5?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Assignment.

Write a program that:

- Processes a simple mortgage application.
- Inputs a persons full name, social security number, full address, total yearly salary, total monthly debt payment, target loan amount, interest rate, and load duration.
- Calculates a mortgage payment based on the target loan amount, interest rate, and loan duration.
- Calculates the person's current debt ratio, and the debt ratio with the mortgage.
- Qualifies or rejects the person based on standard 28% mortgage, 36% total debt ratio.
- Prints a simple mortgage application summary.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Handling Strings, Memory, and Characters

What needs to be remembered when working with strings.

- The key to successful string processing can be summarized into two statements.
  - Remember that C does no size or bounds checking, and it is your responsibility to allocate sufficient storage for the operation.
  - Consider whether the string being processed is an individual copy of a string or just a pointer to the original.
- Strings are normally handled with string functions.
- String functions can be written in-house, supplied with the environment, or purchased from a library vendor.
- All string functions deal with strings one byte at a time in an iterative style, and they use the NULL terminator to determine where to stop. If no NULL terminator is found, the string function will continue to process until it finds a NULL terminator, or the program enters an unpredictable state.
- It is permitted to place more than one NULL terminator in a string.
- Placing multiple NULL terminators in a string can be a convenience when parsing a string, or performing other complex manipulations such as implementing an array of variable length strings.
- Strings with multiple NULL terminators require special care in handling.
- A string function will stop processing when it reaches the first NULL terminator, so to copy a string with multiple NULL terminators, block oriented memory functions need to be used.
- To reach the characters after the first NULL terminator, the address of the character after the NULL terminator would be passed to the string function.
  - The address of the character after the first NULL can be gotten by using a string function that returns the length of the string, and taking the address of the character at string subscript length plus one.
  - The string function will process the portion of the string as normal, and will stop with the next NULL terminator found.

Manipulating strings, memory, and characters with functions.

- Before you author a function to handle strings in some way, you may find that what you need is already included in the compiler's environment.
- There is a large variety of functions that will manipulate, convert, analyze strings, memory, and characters.
- The functions are normally present from one environment to another, but some times there are small differences with the functions.
- The are some cases where careful attention to near and far data it required.
- Some of the available functions follow.

- int isalnum(int c);
  - Isalnum() tests for an alphanumeric value.
  - The character to be tested is passed as the integer c.
  - If c is alphanumeric, isalnum() returns "true". Otherwise isalnum() returns "false".
  - Isalnum() is found in the header file ctype.h.
  - A short example of isalnum() follows:

```
char x = '6';
if (isalnum(x))
  puts("x is alphanumeric.");
else
  puts("x is not alphanumeric.");
```

- int isalpha(int c);
  - Isalpha() tests for an alphabetic value.
  - The character to be tested is passed as the integer c.
  - If c is alphabetic, isalpha() returns "true". Otherwise isalpha() returns "false".
  - Isalpha() is found in the header file ctype.h.
  - A short example of isalpha() follows:

```
char x = 'A';
if (isalpha(x))
  puts("x is alphabetic.");
else
  puts("x is not alphabetic.");
```

- int isascii(int c);
  - Isascii() tests for an ASCII value (0 to 127).
  - The character to be tested is passed as the integer c.
  - If c is ASCII, isascii() returns "true". Otherwise isascii() returns "false".
  - Isascii() is found in the header file ctype.h.
  - A short example of isascii() follows:

```
char x = 5;
if (isascii(x))
  puts("x is ASCII.");
else
  puts("x is not ASCII.");
```

- int iscntrl(int c);
  - Iscntrl() tests for a control value (0 to 31, and 127).
  - The character to be tested is passed as the integer c.
  - If c is a control value, iscntrl() returns "true". Otherwise iscntrl() returns "false".
  - Iscntrl() is found in the header file ctype.h.
  - A short example of iscntrl() follows:

```
char x = 127;
if (iscntrl(x))
  puts("x is a control character.");
else
  puts("x is not a control character.");
```

- int isdigit(int c);
  - Isdigit() tests for a digit value ('0' through '9').
  - The character to be tested is passed as the integer c.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - If c is a digit, isdigit() returns "true". Otherwise isdigit() returns "false".
  - Isdigit() is found in the header file ctype.h.
  - A short example of isdigit() follows:

```
char x = '8';
if (isdigit(x))
  puts("x is a digit.");
else
  puts("x is not a digit.");
```

- int isgraph(int c);
  - Isgraph() tests for a non-space graphic value (33 to 126).
  - The character to be tested is passed as the integer c.
  - If c is a non-space graphic, isgraph() returns "true". Otherwise isgraph() returns "false".
  - Isgraph() is found in the header file ctype.h.
  - A short example of isgraph() follows:

```
char x = 'T';
if (isgraph(x))
  puts("x is a non-space graphic.");
else
  puts("x is not a non-space graphic.");
```

- int islower(int c);
  - Islower() tests for a lowercase value.
  - The character to be tested is passed as the integer c.
  - If c is a lowercase, islower() returns "true". Otherwise islower() returns "false".
  - Islower() is found in the header file ctype.h.
  - A short example of islower() follows:

```
char x = 'g';
if (islower(x))
  puts("x is lowercase.");
else
  puts("x is not lowercase.");
```

- int isprint(int c);
  - Isprint() tests for a printable value (32 to 126).
  - The character to be tested is passed as the integer c.
  - If c is a printable value, isprint() returns "true". Otherwise isprint() returns "false".
  - Isprint() is found in the header file ctype.h.
  - A short example of isprint() follows:

```
char x = '$';
if (isprint(x))
  puts("x is a printable value.");
else
  puts("x is not a printable value.");
```

- int ispunct(int c);
  - Ispunct() tests for a punctuation character.
  - The character to be tested is passed as the integer c.

  - If c is a punctuation character, ispunct() returns "true". Otherwise ispunct() returns "false".
  - Ispunct() is found in the header file ctype.h.
  - A short example of ispunct() follows:

```
char x = '!';
if (ispunct(x))
  puts("x is a punctuation character.");
else
  puts("x is not a punctuation character.");
```

- int isspace(int c);
  - Isspace() tests for a spacing character (space, tab, or new line).
  - The character to be tested is passed as the integer c.
  - If c is a spacing, character isspace() returns "true". Otherwise isspace() returns "false".
  - Isspace() is found in the header file ctype.h.
  - A short example of isspace() follows:

```
char x = ' ';
if (isspace(x))
  puts("x is a spacing character.");
else
  puts("x is not a spacing character.");
```

- int isupper(int c);
  - Isupper() tests for a uppercase character.
  - The character to be tested is passed as the integer c.
  - If c is a uppercase character, isupper() returns "true". Otherwise isupper() returns "false".
  - Isupper() is found in the header file ctype.h.
  - A short example of isupper() follows:

```
char x = 'Z';
if (isupper(x))
  puts("x is a uppercase character.");
else
  puts("x is not a uppercase character.");
```

- int isxdigit(int c);
  - Isxdigit() tests for a hexadecimal digit value ('0' through '9').
  - The character to be tested is passed as the integer c.
  - If c is a hexadecimal digit, isxdigit() returns "true". Otherwise isxdigit() returns "false".
  - Isxdigit() is found in the header file ctype.h.
  - A short example of isxdigit() follows:

```
char x = 'f';
if (isxdigit(x))
  puts("x is a hexadecimal digit.");
else
  puts("x is not a hexadecimal digit.");
```

- void* memccpy(void *dest, const void *src, int c, int n);
- void far* _fmemccpy(void far *dest, const void far *src, int c, int n)
  - Memccpy() is used for near operations, and _fmemccpy is used for far operations.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - Memccpy() copies the memory pointed to by src into the location pointed to by dest. Memccpy() copies byte by byte until the character c is copied, or a total of n characters are copied.
  - Memccpy() is not a string style function, and does not recognize or use NULL.
  - The destination address to be filled is passed as dest.
  - The source address to be copied is passed as src.
  - The character that stops the copy process is passed as c.
  - The maximum number of characters to copy is passed as n.
  - The return value of memccpy() is the address in dest directly after the c character. If c was not copied, the return value is NULL.
  - Memccpy() is found in the header file mem.h and string.h.
  - A short example of memccpy() follows:

```
char x[11] = "ABCD";
char y[11] = "";
if (memccpy(y,x,NULL,10))
  puts("Null was found.");
else
  puts("The max. of 10 was copied.");
```

- void *memchr(const void *s, int c, int n);
- void far * far _fmemchr(const void far *s, int c, int n);
  - Memchr() is used for near operations, and _fmemchr is used for far operations.
  - Memchr() searches the memory pointed to by s for the first occurrence of c is found, or a total of n characters are searched.
  - Memchr() is not a string style function, and does not recognize or use NULL.
  - The address to be searched is passed as s.
  - The character that the search is for is passed as c.
  - The maximum number of characters to search is passed as n.
  - The return value of memchr() is the address in s of the first occurrence of c. If c was not found, the return value is NULL.
  - Memchr() is found in the header file mem.h and string.h.
  - A short example of memchr() follows:

```
char x[11] = "ABCD";
char y = 'C';
if (memchr(x,y,10))
  puts("y was found.");
else
  puts("The max. of 10 was searched.");
```

- int memcmp(const void *s1, const void *s2, int n);
- int far _fmemcmp(const void far *s1, const void far *s2, int n);
  - Memcmp() is used for near operations, and _fmemcmp is used for far operations.
  - Memcmp() compares the memory pointed to by s1 with the memory pointed to by s2 for n bytes.
  - Memcmp() is not a string style function, and does not recognize or use NULL.
  - The address of the memory to be compared is passed as s1 and s2.
  - The maximum number of characters compared is passed as n.
  - The return value of memcmp() is zero if s1 is the same as s2, less than zero if s1 is less than s2, and greater than zero if s1 is greater than s2.

  - Memcmp() is found in the header file mem.h and string.h.
  - A short example of memcmp() follows:

```
char x[11] = "ABCD";
char y[11] = "ABCD";
if (!memcmp(x,y,10))
  puts("y and x are the same.");
else
  puts("y and x are not the same.");
```

- void *memcpy(void *dest, const void *src, int n);
- void far *far _fmemcpy(void far *dest, const void far *src, int n);
  - Memcpy() is used for near operations, and _fmemcpy is used for far operations.
  - Memcpy() copies n bytes of the memory pointed to by src into the memory pointed to by dest.
  - Memcpy() is not a string style function, and does not recognize or use NULL.
  - Memcpy copies byte by byte, so if src and dest overlap the results are invalid.
  - The destination address to be filled is passed as dest.
  - The source address to be copied is passed as src.
  - The maximum number of characters to copy is passed as n.
  - The return value of memcpy() is dest.
  - Memcpy() is found in the header file mem.h and string.h.
  - A short example of memcpy() follows:

```
char x[11] = "ABCD";
char y[11] = "";
memcpy(y,x,10);
```

- void *memmove(void *dest, const void *src, int n);
- void far *far _fmemmove(void far *dest, const void far *src, int n);
  - Memmove() is used for near operations, and _fmemmove is used for far operations.
  - Memmove() copies n bytes of the memory pointed to by src into the memory pointed to by dest.
  - Memmove() is not a string style function, and does not recognize or use NULL.
  - Memmove copies as a block, so if src and dest overlap, the results are correct.
  - The destination address to be filled is passed as dest.
  - The source address to be copied is passed as src.
  - The maximum number of characters to copy is passed as n.
  - The return value of memmove() is dest.
  - Memmove() is found in the header file mem.h and string.h.
  - A short example of memmove() follows:

```
char x[11] = "ABCD";
char y[11] = "";
memmove(y,x,10);
```

- void *memset(void *s, int c, size_t n);
- void far * far _fmemset (void far *s, int c, size_t n);
  - Memset() is used for near operations, and _fmemset is used for far operations.
  - Memset() sets n bytes of the memory pointed to s to the value of c.
  - Memset() is not a string style function, and does not recognize or use NULL.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - The address to be filled is passed as s.
  - The value to fill s with is passed as c.
  - The maximum number of characters to fill is passed as n.
  - The return value of memset() is s.
  - Memset() is found in the header file mem.h and string.h.
  - A short example of memset() follows:

```
char x[11] = "ABCD";
memset(x,NULL,sizeof(x));
```

- int sprintf(char *s, const char *fmt[, args ...]);
  - Sprintf() formats the arguments as specified by fmt, and copies the formatted string into the string pointed to by s.
  - Sprintf() is much like printf(), but instead of printing to the standard output device is places its result into a string.
  - The address of the destination string is passed as s.
  - The address of the format string is passed as fmt.
  - The arguments to be formatted are passed after fmt.
  - The return value of sprintf() is the size of the formatted string.
  - Sprintf() is found in the header file stdio.h.
  - A short example of sprintf() follows:

```
char x[40] = "";
sprintf(x,"I am %d years old.\n", 28);
```

- char* strcat(char *s1, const char *s2);
- char far* _fstrcat(char far *s1, const char far *s2);
  - Strcat() is used for near operations, and _fstrcat is used for far operations.
  - Strcat() concatenates the string pointed to by s2 on to the end of the string pointed to by s1.
  - Strcat() is a string style function, and the strings must be NULL terminated.
  - The address of the destination string is passed as s1.
  - The address of the source string is passed as s2.
  - The return value of strcat() is s1.
  - Strcat() is found in the header file string.h.
  - A short example of strcat() follows:

```
char x[21] = "ABCD";
char y[11] = "EFGH";
strcat(x,y);
```

- char *strchr(const char *s, char c);
- char far * far _fstrchr(const char far *s, char c);
  - Strchr() is used for near operations, and _fstrchr is used for far operations.
  - Strchr() searches the string pointed to by s for the first occurrence of c.
  - Strchr() is a string style function, and the strings must be NULL terminated.
  - The address to be searched is passed as s.
  - The character that the search is for is passed as c.
  - The return value of strchr() is the address in s of the first occurrence of c. If c was not found, the return value is NULL.
  - Strchr() is found in the header file string.h.
  - A short example of strchr() follows:


```
char x[11] = "ABCD";
char y = 'C';
if (strchr(x,y))
  puts("y was found.");
```

- int strcmp(const char *s1, const char *s2);
- int far _fstrcmp(const char far *s1, const char far *s2);
  - Strcmp() is used for near operations, and _fstrcmp is used for far operations.
  - Strcmp() compares the string pointed to by s1 with the string pointed to by s2.
  - Strcmp() is a string style function, and the strings must be NULL terminated.
  - The address of the strings to be compared is passed as s1 and s2.
  - The return value of strcmp() is zero if s1 is the same as s2, less than zero if s1 is less than s2, and greater than zero if s1 is greater than s2.
  - Strcmp() is found in the header file string.h.
  - A short example of strcmp() follows:

```
char x[11] = "ABCD";
char y[11] = "ABCD";
if (!strcmp(x,y))
  puts("y and x are the same.");
else
  puts("y and x are not the same.");
```

- char *strcpy(char *dest, const char *src);
- char far *far _fstrcpy(char far *dest, const char far *src);
  - Strcpy() is used for near operations, and _fstrcpy is used for far operations.
  - Strcpy() copies the string pointed to by src into the string pointed to by dest.
  - Strcpy() is a string style function, and the strings must be NULL terminated.
  - The destination string is passed as dest.
  - The source string is passed as src.
  - The return value of strcpy() is dest.
  - Strcpy() is found in the header file string.h.
  - A short example of strcpy() follows:

```
char x[11] = "ABCD";
char y[11] = "";
strcpy(y,x);
```

- int strcspn(char *s1, const char *s2);
- int _fstrcspn(char far *s1, const char far *s1);
  - Strcspn() is used for near operations, and _fstrcspn is used for far operations.
  - Strcspn() searches the string pointed to by s1 for the first occurrence of any character in the string pointed to by s2.
  - Strcspn() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be searched is passed as s1.
  - The address of the string containing the characters to search for is passed as s2.
  - The return value of strcspn() is an array index within the string pointed to by s1 of the first occurrence of any character listed in the string pointed to by s2. If no characters are found, the length of the string pointed to by s1 is returned.
  - Strcspn() is found in the header file string.h.
  - A short example of strcspn() follows:

```
char x[15] = "This is a test";
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
char y[4] = "abc";
printf("%d\n",strcspn(x,y)); /* prints 8 */
```

- int stricmp(const char *s1, const char *s2);
- int far _fstricmp(const char far *s1, const char far *s2);
  - Stricmp() is used for near operations, and _fstricmp is used for far operations.
  - Stricmp() compares the string pointed to by s1 with the string pointed to by s2 without regard for case.
  - Stricmp() is a string style function, and the strings must be NULL terminated.
  - The address of the strings to be compared is passed as s1 and s2.
  - The return value of stricmp() is zero if s1 is the same as s2, less than zero if s1 is less than s2, and greater than zero if s1 is greater than s2.
  - Stricmp() is found in the header file string.h.
  - A short example of stricmp() follows:

```
char x[11] = "ABCD";
char y[11] = "abcd";
if (!stricmp(x,y))
  puts("y and x are the same.");
else
  puts("y and x are not the same.");
```

- size_t strlen(char *s);
- size_t _fstrlen(char far *s);
  - Strlen() is used for near operations, and _fstrlen is used for far operations.
  - Strlen() counts the characters in the string pointed to by s until a NULL is found.
  - Strlen() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be counted is passed as s.
  - The return value of strlen() is the length of the string pointed to by s. The NULL terminator is not counted.
  - Strlen() is found in the header file string.h.
  - A short example of strlen() follows:

```
char x[14] = "ABCDEF";
printf("%d\n",strlen(x)); /* prints 6 */
```

- char *strlwr(char *s);
- char far *far _fstrlwr(char far *s);
  - Strlwr() is used for near operations, and _fstrlwr is used for far operations.
  - Strlwr() changes all the characters in the string pointed to by s to lowercase.
  - Strlwr() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be converted is passed as s.
  - The return value of strlwr() is s.
  - Strlwr() is found in the header file string.h.
  - A short example of strlwr() follows:

```
char x[14] = "ABCDEF";
strlwr(x);
```

- char* strncat(char *s1, const char *s2, int n);
- char far* _fstrncat(char far *s1, const char far *s2, int n);
  - Strncat() is used for near operations, and _fstrncat is used for far operations.

  - Strncat() concatenates the first n characters of the string pointed to by s2 onto the end of the string pointed to by s1.
  - Strncat() is a string style function, and the strings must be NULL terminated.
  - The address of the destination string is passed as s1.
  - The address of the source string is passed as s2.
  - The number of characters to concatenate is passed as n.
  - The return value of strncat() is s1.
  - Strncat() is found in the header file string.h.
  - A short example of strncat() follows:

```
char x[21] = "ABCD";
char y[11] = "EFGHIJK";
strncat(x,y,4); /* x is "ABCDEFGH" */
```

- int strncmp(const char *s1, const char *s2, int n);
- int far _fstrncmp(const char far *s1, const char far *s2, int n);
  - Strncmp() is used for near operations, and _fstrncmp is used for far operations.
  - Strncmp() compares first n characters of the string pointed to by s1 with the string pointed to by s2.
  - Strncmp() is a string style function, and the strings must be NULL terminated.
  - The address of the strings to be compared is passed as s1 and s2.
  - The number of characters to be compared is passed as n.
  - The return value of strncmp() is zero if s1 is the same as s2, less than zero if s1 is less than s2, and greater than zero if s1 is greater than s2.
  - Strncmp() is found in the header file string.h.
  - A short example of strncmp() follows:

```
char x[11] = "ABCD";
char y[11] = "ABCDEFG";
if (!strncmp(x,y,4))
  puts("y and x are the same."); /*This will print*/
else
  puts("y and x are not the same.");
```

- char *strncpy(char *dest, const char *src, int n);
- char far *far _fstrncpy(char far *dest, const char far *src, int n);
  - Strncpy() is used for near operations, and _fstrncpy is used for far operations.
  - Strncpy() copies the first n characters of the string pointed to by src into the string pointed to by dest.
  - Strncpy() is a string style function, and the strings must be NULL terminated.
  - The destination string is passed as dest.
  - The source string is passed as src.
  - The number of characters to copy is passed as n.
  - The return value of strncpy() is dest.
  - Strncpy() is found in the header file string.h.
  - A short example of strncpy() follows:

```
char x[11] = "ABCDEFGH";
char y[11] = "";
strncpy(y,x,4); /* y is "ABCD" */
```

- char *strnset(char *s, int c, int n);
- char far * far _fstrnset (char far *s, int c, int n);

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - Strnset() is used for near operations, and _fstrnset is used for far operations.
  - Strnset() sets n bytes of the string pointed to s to the value of c.
  - Strnset() is a string style function, and the strings must be NULL terminated.
  - The address to be filled is passed as s.
  - The value to fill s with is passed as c.
  - The maximum number of characters to fill is passed as n.
  - The return value of strnset() is s.
  - Strnset() is found in the header file string.h.
  - A short example of strnset() follows:

```
char x[11] = "ABCD";
strnset(x,NULL,sizeof(x));
```

- char *strpbrk(char *s1, const char *s2);
- char far *far _fstrpbrk(char far *s1, const char far *s1);
  - Strpbrk() is used for near operations, and _fstrpbrk is used for far operations.
  - Strpbrk() searches the string pointed to by s1 for the first occurrence of any character in the string pointed to by s2.
  - Strpbrk() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be searched is passed as s1.
  - The address of the string containing the characters to search for is passed as s2.
  - The return value of strpbrk() is a pointer within the string pointed to by s1 of the first occurrence of any character listed in the string pointed to by s2. If no characters are found, NULL is returned.
  - Strpbrk() is found in the header file string.h.
  - A short example of strpbrk() follows:

```
char x[15] = "This is a test";
char y[4] = "abc";
puts(strpbrk(x,y)); /* prints "a test" */
```

- char *strrchr(const char *s, char c);
- char far * far _fstrrchr(const char far *s, char c);
  - Strrchr() is used for near operations, and _fstrrchr is used for far operations.
  - Strrchr() searches the string pointed to by s for the last occurrence of c.
  - Strrchr() is a string style function, and the strings must be NULL terminated.
  - The address to be searched is passed as s.
  - The character that the search is for is passed as c.
  - The return value of strrchr() is the address in s of the last occurrence of c. If c was not found, the return value is NULL.
  - Strrchr() is found in the header file string.h.
  - A short example of strrchr() follows:

```
char x[30] = "can the cat run?";
char y = 'c';
puts(strrchr(x,y)); /* prints "cat run?" */
```

- char* strrev(char *s);
- char far *far _fstrrev(char far *s);
  - Strrev() is used for near operations, and _fstrrev is used for far operations.
  - Strrev() reverses all the characters in the string pointed to by s.
  - Strrev() is a string style function, and the strings must be NULL terminated.

  - The address of the string to be reversed is passed as s.
  - The return value of strrev() is s.
  - Strrev() is found in the header file string.h.
  - A short example of strrev() follows:

```
char x[14] = "ABCDEF";
puts(strrev(x)); /* prints "FEDCBA" */
```

- char *strset(char *s, int c, int n);
- char far * far _fstrset (char far *s, int c, int n);
  - Strnset() is used for near operations, and _fstrnset is used for far operations.
  - Strnset() sets the string pointed to s to the value of c.
  - Strnset() is a string style function, and the strings must be NULL terminated.
  - The address to be filled is passed as s.
  - The value to fill s with is passed as c.
  - The return value of strnset() is s.
  - Strnset() is found in the header file string.h.
  - A short example of strnset() follows:

```
char x[11] = "ABCD";
strnset(x,' ');
```

- int strspn(char *s1, const char *s2);
- int _fstrspn(char far *s1, const char far *s1);
  - Strspn() is used for near operations, and _fstrspn is used for far operations.
  - Strspn() searches the string pointed to by s1 for the first occurrence of any character not in the string pointed to by s2.
  - Strspn() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be searched is passed as s1.
  - The address of the string containing the filter characters is passed as s2.
  - The return value of strspn() is array index within the string pointed to by s1 of the first occurrence of any character not listed in the string pointed to by s2. If no bad characters are found, the length of the string pointed to by s1 is returned.
  - Strspn() is found in the header file string.h.
  - A short example of strspn() follows:

```
char x[15] = "This is a test";
char y[4] = " hisT";
printf("%d\n",strspn(x,y)); /* prints 7 */
```

- char* strstr(char *s1, char *s2);
- char far* _fstrstr(char far *s1, char far *s1);
  - Strstr() is used for near operations, and _fstrstr is used for far operations.
  - Strstr() searches the string pointed to by s1 for the first occurrence of the string pointed to by s2.
  - Strstr() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be searched is passed as s1.
  - The address of the string to be searched for is passed as s2.
  - The return value of strstr() is address within the string pointed to by s1 of the first occurrence of the string pointed to by s2. If no occurrence of the string pointed to by s2 is found, NULL is returned.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - Strstr() is found in the header file string.h.
  - A short example of strstr() follows:

```
char x[15] = "This is a test";
char y[4] = "is";
puts(strstr(x,y)); /* prints "is a test" */
```

- char* strtok(char *s1, char *s2);
- char far* _fstrtok(char far *s1, char far *s1);
  - Strtok() is used for near operations, and _fstrtok is used for far operations.
  - Strtok() searches the string pointed to by s1 for the first occurrence of any character in the string pointed to by s2. If s1 is NULL, strtok() starts searching after the address of the last occurrence found. When an occurrence is found, a null is placed at the address of the find.
  - Strtok() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be searched is passed as s1.
  - The address of the string of characters to be searched for is passed as s2.
  - The return value of strtok() is address of the sub-string within the string pointed to by s1 before the latest find. If at the end of the string pointed to by s1, NULL is returned.
  - Strtok() is found in the header file string.h.
  - A short example of strtok() follows:

```
char x[15] = "This is a test";
char* y = strtok(x," ");
while (y)
{ puts(y);
  y = strtok(NULL," ");
}
```

- char *strupr(char *s);
- char far *far _fstrupr(char far *s);
  - Strupr() is used for near operations, and _fstrupr is used for far operations.
  - Strupr() changes all the characters in the string pointed to by s to uppercase.
  - Strupr() is a string style function, and the strings must be NULL terminated.
  - The address of the string to be converted is passed as s.
  - The return value of strupr() is s.
  - Strupr() is found in the header file string.h.
  - A short example of strupr() follows:

```
char x[14] = "abcdef";
strupr(x);
```

- int tolower(int c);
  - Tolower() returns the lowercase of character c, but does not change c.
  - The character to lowercase is passed as c.
  - The return value of tolower() is c in lowercase.
  - Tolower() is found in the header file ctype.h.
  - A short example of tolower() follows:

```
char x = 'A';
x = tolower(x);
```

- int toupper(int c);
  - Toupper() returns the uppercase of character c, but does not change c.

- The character to uppercase is passed as c.
- The return value of toupper() is c in uppercase.
- Toupper() is found in the header file ctype.h.
- A short example of toupper() follows:

```
char x = 'A';
x = toupper(x);
```

What needs to be done to use string, memory, or character functions.

- Just include ctype.h, mem.h or string.h using a #include <ctype.h>, #include <mem.h>, or #include <string.h> at the top of the program.
- Link with the standard library when building the .EXE.

Examples.

- See example disk for program STRNGEXM.C

Quick Test.

1. What function is used to compare two strings without case?
2. What is a problem with storage when concatenating strings?
3. What is the difference between memcpy and memmove?
4. What function would be used to find the index of '!', '?', or ',' within a string?
5. How can a character be tested to determine if it is a digit?
6. What function concatenates a specified number of characters onto a string?

Assignment.

Write a program that:

- Reports simple paragraph information.
- Allows the user to enter a paragraph of up to 100 - 127 seven character lines, and stores them in an array of strings.
- Calculates how many words, punctuation characters, spaces, lines, uppercase letters, and lowercase letters were entered.
- Prints a simple summary.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Backslash Control Characters

The background on backslash control characters.

- The backslash gives special meaning within a single or double quoted literal.
- A backslash is combined with code character and placed within a quoted literal.
- The backslash and code combination acts as a control character that is replaced with its respective meaning during compilation.
- Even though the backslash code combination appears as two or more characters within the editing environment, it is replaced with only one character during compilation.
- Only quoted littorals that are interpreted by the compiler are replaced.
- Strings or values set at runtime, and preprocessor directives, are not affected.
- A list of backslash control characters follow:

  \b Backspace
  \n New line
  \r Carriage return
  \t Horizontal tab
  \" Double quotation marks
  \' Single quotation marks
  \0 Null
  \\ Backslash
  \v Vertical tab (printer only)
  \aAlert (beep)
  \x Hexadecimal constant

- Some examples of backslash control characters follow:

```
puts("Backspace                >\b<");
puts("New line                 >\n<");
puts("Carriage return          >\r<");
puts("Horizontal tab           >\t<");
puts("Double quotation marks   >\"<");
puts("Single quotation marks   >\'<");
puts("Null                     >\0<");
puts("Backslash                >\\<");
puts("Vertical tab             >\v<");
puts("Alert                    >\a<");
puts("Hexadecimal constant     >\x20<");
```

Quick Test.

1. Does the backslash control character have an effect in a preprocessor statement?
2. Where is a backslash control character valid?
3. How many characters replace the backslash control character?
4. When does the replacement of the backslash control character take place?
5. What backslash control character would cause audio output?
6. Do single or double quotes make a difference with backslash control characters?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

The Conversion Functions

The background on the conversion functions.

- Conversion functions are used to convert from numeric information to string information, and from string information to numeric information.
- The conversion functions are found in the include file stdlib.h.
- To use the conversion functions, simply include stdlib.h by placing #include <stdlib.h> at the top of your program.
- A list and description of the conversion functions follow.
- int atoi(const char *s);
  - Atoi() converts a string to an integer.
  - A pointer to the string to be converted is passed as s.
  - Atoi() ignores preceding white spaces (" 123") including tabs.
  - A sign can precede the number ("-123"), and will set the sign of the converted value.
  - The conversion ends with the first non-convertible character encountered, or at the end of the string.
  - Atoi() will convert the string to a number even if it results in an overflow.
  - If an overflow occurs, the results are unpredictable.
  - A short example of atoi() follows:

```
int x = atoi("-5678");
```

- double atof(const char *s);
- long double _atold(const char *s);
  - Atof() converts a string to a double, and _atold() converts a string to a long double.
  - A pointer to the string to be converted is passed as s.
  - Atof() ignores preceding white spaces (" 123") including tabs.
  - A sign can precede the number ("-123"), and will set the sign of the converted value.
  - A decimal can be included in the string.
  - The string can use the e or E notation.
  - Atof() will convert a string with +INF and -INF to mean plus and minus infinity, also +NAN and -NAN can be used for not-a-number.
  - The conversion ends with the first non-convertible character encountered, or at the end of the string.
  - Atof() will convert the string to a number even if it results in an overflow.
  - If an overflow occurs, the results will be plus or minus HUGE_VAL for atof(), and plus or minus _LHUGE_VAL for _atold().
  - A short example of atof() follows:

```
double x = atof("45.56");
```

- long atol(const char *s);
  - Atol() converts a string to a long integer.
  - A pointer to the string to be converted is passed as s.
  - Atol() ignores preceding white spaces (" 123") including tabs.

  - A sign can precede the number ("-123"), and will set the sign of the converted value.
  - The conversion ends with the first non-convertible character encountered, or at the end of the string.
  - Atol() will convert the string to a number even if it results in an overflow.
  - If an overflow occurs, the results are unpredictable.
  - A short example of atol() follows:

```
long int x = atol("69998");
```

- char *itoa(int value, char *string, int radix);
  - Itoa() converts an integer into a NULL terminated string.
  - The number to be converted is passed as value.
  - A pointer to the string to receive the converted value passed as string.
  - The base of the number is passed as radix, and must be between 2 and 36, inclusive.
  - The size of the string used to receive the converted value must be large enough or memory will be overwritten.
  - The return value is string.
  - A short example of itoa() follows:

```
char x[51] = "";
itoa(32,x,16); /* x is set to "20" */
```

- char *ecvt(double value, int ndig, int *dec, int *sign);
  - Ecvt() converts a double into a NULL terminated string.
  - The number to be converted is passed as value.
  - The number of significant digits to be converted is passed as ndig.
  - A pointer to the static converted string is returned.
  - There is no sign or decimal point in the converted string.
  - The location of where the decimal point should be in the converted string is assigned to the integer pointed to by dec.
  - The sign of the converted value is assigned to the integer pointed to by sign (0 == positive, and 1 == negative).
  - The return value must be used to copy the converted string, because each call to ecvt() overwrites the last converted string.
  - A short example of ecvt() follows:

```
char* x = NULL;
int dec = 0;
int sign = 0;
x = ecvt(56.77,4,&dec, &sign);
```

- char *fcvt(double value, int ndig, int *dec, int *sign);
  - Fcvt() converts a double into a NULL terminated string.
  - The number to be converted is passed as value.
  - The number of significant digits to be converted is passed as ndig.
  - A pointer to the static converted string is returned.
  - There is no sign or decimal point in the converted string.
  - The location of where the decimal point should be in the converted string is assigned to the integer pointed to by dec.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - The sign of the converted value is assigned to the integer pointed to by sign (0 == positive, and 1 == negative).
  - The return value must be used to copy the converted string, because each call to fcvt() overwrites the last converted string.
  - The converted string is in FORTRAN format.
  - A short example of fcvt() follows:

```
char* x = NULL;
int dec = 0;
int sign = 0;
x = fcvt(656.67,5,&dec, &sign);
```

- char *gcvt(double value, int ndig, char *buf);
  - Gcvt() converts a double into a NULL terminated string.
  - The number to be converted is passed as value.
  - The number of significant digits to be converted is passed as ndig.
  - A pointer to the string that gcvt() will place the converted value is passed as buf.
  - The return value is buf.
  - A short example of gcvt() follows:

```
char x[51] = "";
gcvt(6757.75,6,x);
```

- char *ltoa(long value, char *string, int radix);
  - Ltoa() converts a long integer into a NULL terminated string.
  - The number to be converted is passed as value.
  - A pointer to the string to receive the converted value passed as string.
  - The base of the number is passed as radix, and must be between 2 and 36, inclusive.
  - The size of the string used to receive the converted value must be large enough or memory will be overwritten.
  - The return value is string.
  - A short example of ltoa() follows:

```
char x[51] = "";
ltoa(70000,x,10); /* x is set to "70000" */
```

- double strtod(const char *s, char **endptr);
- long double _strtold(const char *s, char **endptr);
  - Strtod() converts a string to a double, and _strtold() converts a string to a long double.
  - A pointer to the string to be converted is passed as s.
  - Strtod() ignores preceding white spaces (" 123") including tabs.
  - A sign can precede the number ("-123"), and will set the sign of the converted value.
  - A decimal can be included in the string.
  - The string can use the e or E notation.
  - Strtod() will convert a string with +INF and -INF to mean plus and minus infinity, also +NAN and -NAN can be used for not-a-number.
  - The conversion ends with the first non-convertible character encountered, or at the end of the string.

  - Endptr is assigned the address of the non-convertible character that stopped the conversion.
  - A NULL can be passed for endptr. This will cause strtod() to ignore endptr.
  - Strtod() will convert the string to a number even if it results in an overflow.
  - If an overflow occurs the results will be plus or minus HUGE_VAL for strtod(), and plus or minus _LHUGE_VAL for _atold().
  - A short example of strtod() follows:

```
double x = strtod("678.56",NULL);
```

- char *ultoa(unsigned long value, char *string, int radix);
  - Ultoa() converts an unsigned long integer into a NULL terminated string.
  - The number to be converted is passed as value.
  - A pointer to the string to receive the converted value passed as string.
  - The base of the number is passed as radix, and must be between 2 and 36, inclusive.
  - The size of the string used to receive the converted value must be large enough or memory will be overwritten.
  - The return value is string.
  - A short example of ultoa() follows:

```
char x[51] = "";
ultoa(123000,x,10); /* x is set to "123000" */
```

Quick Test.

1. Which function is used to convert a string value to an integer number?
2. When does a conversion from a string to a numeric stop?
3. Which function is used to convert a double number to a string with a decimal?
4. Do any conversion functions check for storage overflow?
5. Where are the prototypes for the conversion functions found?
6. Do any conversion functions provide a way of telling where a conversion stopped?

Assignment.

Write a program that:

- Declares variables of five different data types int, float, long, double, and string.
- Reads data from the keyboard for each variable using gets().
- Converts each value to its data type using the conversion functions.
- Prints the value entered for each variable to the screen.
- Prints the size of each variable to the screen.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

The Preprocessor

The background on the preprocessor.
- The preprocessor is part of the compiler.
- The preprocessor scans for preprocessor directives.
- All preprocessor directives start with a #.
- The directives form a small language set of their own.
- The directives are only interpreted during compilation; not at execution.
- The directives do not end with a ';'.

Including other source files in your compilation.
- To have the preprocessor include an external file for compilation, the #include directive is used.
- The format of the #include directive follows:

```
#include <file name>
#include "file name"
```

- The use of less than and greater than symbols tells the preprocessor that it will find the file in the environment's include path.
- The use of the double quote symbols tells the preprocessor that is should check the current directory or the path name of the file before searching the environment's include path.
- An example of #include follows:

```
#include <stdio.h>
#include "c:\mylib\myio.h"
```

Working with text substitution identifiers.
- The #define preprocessor directive is used to define a substitution macro.
- Substitution macros are normally typed in uppercase.
- Macros are faster than function calls because there is no function call overhead.
- Macros are less space efficient than function calls because the macro code is repeated everywhere it is used.
- The format of a #define follow:

```
#define MACRONAME[(arguments)] text value [\
  more text value]
```

- After the macro has been defined, it can be used anywhere in the program.
- During compilation the compiler will replace each instance of the macro with its defined substitution value.
- Arguments can be given to the macro, but there are no types assigned or checked.
- A substitution macro is not a function; it is purely a text substitution, and unexpected results can occur if complex arguments are given to a macro.
- A short example of macro substitution follows:


```
#define PRMPTPRNT(a,b) printf("%s %s,a,b)
void func()
{ char x[51] = "Sue";
  PRMPTPRNT("Hello",x);
}
```

- The macro defined by #define does not need to have a substitution value.
- The format for a macro with no substitution value follows:

```
#define MACRONAME
```

- Without a substitution value, the macro would be used in tests that check for its existence.
- The check for its existence can be used to control the compilation of the program.
- A short example of a macro without a substitution value follows:

```
#define DEBUG_ON
void func()
{ ...
#ifdef DEBUG_ON
puts("Debugging is on");
#endif
  ...
}
```

- If a #define is issued for an existing macro and the substitution string is different, a warning will occur.
- To have a defined macro undefined, use the #undef preprocessor directive.
- The format of the #undef directive follows:

```
#undef MACRONAME
```

- No error occurs if the macro was not defined before the #undef directive.
- A short example of #undef follows:

```
#undef DEBUG_ON
```

Conditional compilation.

- Preprocessor directives can be used to control which program statements get compiled and which statements don't.
- To control the compilation of a program, the conditional directives are used.
- A condition directive sequence requires at least two preprocessor directives. One directive tests the compile time condition and the second directive marks the end of the condition sequence.
- Remember preprocessor directive are compile time only; consequently, they have no iterative capability, and they can only work with literal or macro values.
- The #endif is used to mark the end of a condition sequence, and is required in all conditional sequences.
- The #if directive is used as the start of a condition sequence, and works like a standard if statement in C.
  - The #if can only test compile time values.
  - The #if must have a corresponding #endif, and groupings are not used.
  - A short example of a #if directive follows:

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
#if (MAX_VALUE < 55)
C program statements.
...
#endif
```

- The #else directive works like a standard else statement in C.

```
#if (MAX_VALUE < 55)
C program statements.
...
#else
C program statements.
...
#endif
```

- The #elif combines the else statement with an immediate if.

```
#if (MAX_VALUE < 55)
C program statements.
...
#elif (MIN_VALUE > 2)
C program statements.
...
#endif
```

- The #ifdef is a condition statement that tests for the existence of a macro.

```
#ifdef DEBUG_ON
C program statements.
...
#endif
```

- The #ifndef is a condition statement the test for the in-existence of a macro.

```
#ifndef DEBUG_ON
C program statements.
...
#endif
```

Stopping compilation.

- Preprocessor directives give great control over compilation including the ability to stop the compilation process if needed.
- The #error directive causes compilation to stop, and an error message supplied with the #error directive to be displayed.
- The format of the #error directive follows:

```
#error message
```

- The #error directive can be used to stop compilation if a conditional finds a problem.
- The #error directive can help during program development. It can allow the finished code at the top of a module to compile, and stop compilation before the unfinished code is reached.
- An example of the #error directive follows:

```
#error Got passed func1!
```

Examples.

- See example disk for program PRPRCEXM.C

Quick Test.

1. Can int x be used with a #if?
2. What happens if an attempt to redefine an existing macro occurs?
3. What other directive is required for a #ifdef?
4. Do preprocessor statements require a ';'?
5. What is the difference between double quotes and "<>" in a #include?
6. What two impacts do macros have on a program?

Assignment.

Write a program that:

- Reports simple paragraph information.
- Allows the user to enter a paragraph of up to 100 - 127 seven character lines, and stores them in an array of strings.
- Calculates how many words, punctuation characters, spaces, lines, uppercase letters, and lowercase letters were entered.
- Prints a simple summary.
- Will compile in debugging mode that prints debug messages, or in no debug mode.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

User Defined Data Types

Using structures to group related but dissimilar data.

- Structures are used to construct records in C.
- The format used to declare a structure follows:

```
struct structure-name
{ structure items.
  ...
} [instance-name];
```

- Any data type can be used as an element of a structure including other user defined types.
- Whenever a structure is referenced, the struct keyword must be used.
- Instance-name does not need to be specified in the structure declaration. If it is not specified, an instance can be created later in the program.
- To create new instances of a structure, just use the following format:
  struct structure-name instance-name;
- A pointer to a structure can be made by using the '*' in the declaration.
  struct structure-name* pointer-name;
- It is important to remember that the structure definition does not take up storage space.
- Only when an instance of the structure is made, is memory allocated.
- A short example of a structure follows:

```
struct emp_record
 { char fname[51];
  char lname[51];
  char ssnum[12];
  double salary;
};
struct emp_record emp1;
```

- A structure can be used to implement bit level data.
  - Bit level data gives an easy way of breaking an integer into groups of bits, and is an easy alternative to bit level operators.
  - The bits are often used as flags.
  - When even one bit is allocated, the minimum space allocated is that of an integer. Any further bit level data will use the remaining portion until the integer is full. Once the integer is full, a new integer will be allocated.
  - Bits can be allocated in two styles signed/int or unsigned.
    - Signed type bit fields must be a length of two to sixteen, because one bit will be used to hold the sign.
    - Unsigned data can be a length of 1 to sixteen.
  - Bit fields can not be made into an array, and you can not take the address of a bit field.
  - The format of a bit field follows:

```
type field-name : length;
```

- An example of a structure with bit level data follows:


```
struct emp_record
 { char fname[51];
  char lname[51];
  char ssnum[12];
  double salary;
  unsigned sex : 1;
  unsigned exempt : 1;
};
```

- The elements of a structure can be accessed using dot notation.

```
struct emp_record emp1;
emp1.salary = 0;
```

- Using pointers to structures requires the use of "*." or "->" notation.

```
struct emp_record emp1;
struct emp_record* emp_ptr = &emp1;
(*emp_ptr).salary = 0;
emp_ptr->salary = 0;
```

- The arrow notation is easier to understand.

Using unions to view the same memory in different ways.

- Unions are used to provide different views of the same memory in C.
- The format used to declare a union follows:

```
union union-name
{ union items.
  ...
} [instance-name];
```

- Any data type can be used as an element of a union including other user defined types.
- Whenever a union is referenced, the union keyword must be used.
- To create new instances of a union, just use the following format:

```
union union-name instance-name;
```

- A pointer to a union can be made by using the '*' in the declaration.

```
union union-name* pointer-name;
```

- It is important to remember that the union definition does not take up storage space.
- Only when an instance of the union is made, is memory allocated.
- A short example of a union follows:

```
union emp_record
{ struct payrec payroll;
  struct emprec employee;
};
union emp_record emp1;
```

- The elements of a union can be accessed using dot notation.

```
union emp_record emp1;
emp1.emprec.salary = 0;
```

- Using pointers to unions requires the use of "*." or "->" notation.

```
union emp_record* emp_ptr = &emp1;
emp_ptr->emprec.salary = 0;
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Using enumerations.

- Enumerations are a way of specifying all the valid values that an integer can be set to.
- The values are represented by symbols.
- The symbol values are listed on declaration, and normally start at zero.
- In the following example first == 0, jan == 1, and last == 13.

```
enum months {first, jan, feb, mar, apr, may, jun,
             jul, aug, sep, oct, nov, dec, last};
```

- The list elements can be assigned a starting value by using the '='.
- The '=' can be used as many times as needed in an enumeration.
- In the following example jan == 1, and dec == 12.

```
enum months {jan=1, feb, mar, apr, may, jun,
             jul, aug, sep, oct, nov, dec};
```

- In the following example US coins are enumerated.

```
enum coinage {pen = 1, nic = 5, dim = 10,
              qua = 25, hal = 50, dol = 100};
```

- The declaration of an enumeration does not take up memory. Only instances take up memory.
- To make an instance of a enumerated type, use the following format:

```
enum enum-name instance;
```

- An example of enumeration follows:

```
enum months themonths;
themonths = jan;
```

- Remember that the symbol values are only integer values, and they have no text correlation.

Creating formal data types.

- To name a structure, union, enumeration, array, or any type as a formal C datatype, use the typedef keyword.
- The format of the typedef keyword follows:
  - For structures, unions, enumerations:

```
typedef struct/union/enum
{ elements
   ...
} type-name;
```

  - For arrays:

```
typedef type type-name[array size];
```

  - For other types:

```
typedef type type-name;
```

- The typedef keyword tells the compiler that the type declaration is to be used as any other type declaration in C.
- Struct, enum, or union do not need to be supplied after a declaration has been type defined.
- A type defined structure can not contain a pointer of its own type.
- Some examples of type definitions follow:


```
typedef char namestr[51];

typedef struct
{ namestr fname;
  namestr lname;
  char ssnum[12];
  double salary;
} emp_record;

emp_record emp1;
```

Casting data types so they can fit different roles.

- Type casting is used to have the compiler treat a variable as a different type.
- A variable can only be cast to a type of equal or greater size.
- Any variable can be type cast to act as a different type, but some type casts have no useful outcome.
- Type casting it often used to eliminate warning messages.
- To implement a type cast just place the desired type, within parenthesis, before the data to be cast.
- A short example of type casting follows:

```
char x = 0;
int y = 0;
y = (int)x;
```

Examples.

- See example disk for program USERTEXM.C

Quick Test.

1. What is the minimum memory allocated for a bit field?
2. Can a type definition contain a pointer to its own type?
3. Does a structure definition take up memory?
4. What notation is used to reference the elements of a structure?
5. What notation is used to reference the elements of a structure when accessed with a pointer?
6. When is a type cast invalid?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Assignment.

Write a program that:

- Processes simple mortgage applications.
- Asks the user how many applications they want to enter (10 max).
- Inputs a persons full name, social security number, full address, total yearly salary, and total monthly debt payment into a separate person record for each application.
- Inputs a target loan amount, interest rate, and load duration into a separate application record for each application.
- Calculates a mortgage payment based on the target loan amount, interest rate, and loan duration.
- Calculates the person's current debt ratio, and the debt ratio with the mortgage.
- Qualifies or rejects the person based on standard 28% mortgage, 36% total debt ratio.
- Prints a simple mortgage applications report. The report should include a summary for each application, number of approved applications, and number of rejected applications.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Utilizing Dynamic Memory

The background on dynamic memory.

- The C language supports the use of dynamic memory.
- In most applications, dynamic memory is heavily relied on.
- Dynamic memory is not like global or local data.
  - Global and local data are allocated on the data segment or stack segment.
  - Dynamic memory is allocated on the heap.
  - The size of global and local data is determined at compile time.
  - The size of dynamic data is determined at run time.
  - The size of global and local data is limited by the remaining space of the data or stack segment.
  - The size of global data is limited by the remaining space of the heap, and the heap is generally much larger than either the data or stack segment.
  - Dynamic memory is not labeled with a variable name, and can only be accessed by a pointer.
- The heap is just a large block of memory. It gets logically allocated into portions to the program, and it is the program's responsibility to manage the memory.
  - The program must call for allocation of the space desired, and test to be sure it was given the space.
  - The program needs to store the pointer to the dynamic memory returned by the allocation.
  - The program needs to initialize the memory.
  - The program must call for the release of the memory when it is through using the memory.
  - The program must never use the pointer to the memory after the memory has been released.

Allocating and de-allocating dynamic memory with functions.

- void *malloc(size_t size);
- void far *farmalloc(unsigned long size);
  - Malloc() and farmalloc() are used to allocate dynamic memory.
  - Malloc() allocates up to 64K from the near heap. If the memory model being used does not allow for a near heap, malloc() uses the heap, but still only allocates up to 64K.
  - Farmalloc() allocates dynamic memory from the heap, and has no significant allocation limits.
  - The number of bytes required is passed as size.
  - Malloc() and farmalloc() return a pointer to the allocated memory. If the memory could not be allocated or size was zero, NULL is returned.
- void *calloc(size_t nitems, size_t size);

- void far *farcalloc(unsigned long nitems, unsigned long size);
  - Calloc() and farcalloc() are used to allocate dynamic memory.
  - Calloc() allocates up to 64K from the near heap. If the memory model being used does not allow for a near heap, calloc() uses the heap, but still only allocates up to 64K.
  - Farcalloc() allocates dynamic memory from the heap, and has no significant allocation limits.
  - The number of items is passed as nitems.
  - The number of bytes per item is passed as size.
  - Calloc() and farcalloc() allocate nitem * size bytes of sequential memory.
  - Calloc() and farcalloc() return a pointer to the allocated memory. If the memory could not be allocated or size was zero, NULL is returned.
- void *realloc(void *block, size_t size);
- void far *farrealloc(void far *block, unsigned long size);
  - Realloc() and farrealloc are used to alter the size of a block of dynamic memory.
  - A pointer to the original memory is passed as block.
  - The new size is passed as size.
  - If block is NULL realloc() and farrealloc() act just like malloc() or far malloc().
  - If block is not NULL realloc() and farrealloc() reduces or enlarges the amount of dynamic memory to the amount of size.
  - When enlarging the amount of memory the contents of the original memory may be copied to a new larger location.
  - If size is zero, the block is freed.
  - The return value is the address of the reallocated block of memory. If the reallocation fails or the size is zero, the return value is NULL.
- void free(void *block);
- void farfree(void far * block);
  - Free() and farfree() release memory back to the heap.
  - Free() frees memory to the near heap that was allocated with calloc(), malloc(), or realloc(). If the memory model being used does not allow for a near heap, the heap is used.
  - Farfree() frees memory to the heap that was allocated with farcalloc(), farmalloc(), or farrealloc().
  - A pointer to the memory to be freed is passed as block.
- To use any dynamic memory functions, just include stdlib.h in the program by placing a #include <stdlib.h> at the top of the file, then link with the standard library.

What needs to be remembered when working with memory.

- A pointer must be allocated on the data or stack segment to point to the dynamic memory.
- The dynamic memory must be allocated, and its address must be stored in the pointer.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- The dynamic memory must be initialized and used.
- If the pointer used to store the address is ever lost, the memory is spilled and can not be retrieved.
- Allocated dynamic memory is available to any function in the program that can access the pointer used to store the address.
- Once the memory is no longer required, free it.
- The heap can become very fragmented, and this could lead to out of memory errors. This happens when the amount of free memory is greater than the requested amount, but there are no sequential blocks of memory large enough to hold the requested amount.

A word about virtual memory.

- Virtual memory is capable of using storage space from almost any area of the computer.
- Virtual memory normally encompasses DOS memory, EMS (expanded memory), XMS (extended memory), and disk swap files.
- To use virtual memory, a 3rd party library called a virtual memory manager needs to by implemented.
- The virtual memory manager will be used exclusively when interacting with memory.
- Virtual memory managers can provide many different services including virtual heap defragmentation.
- The standard functions in C do not deal with virtual memory.

Examples.

- See example disk for program DYNAMEXM.C.

Quick Test.

1. Which function can be used to allocate and free memory?
2. Which functions could be used to allocate more than 64K?
3. What are two differences between dynamic and global memory?
4. What are the steps needed to be followed when using dynamic memory?
5. Can the standard C functions deal with EMS?
6. How does heap fragmentation affect memory allocation?

Assignment.

Write a program that:

- Processes simple mortgage applications.
- Asks the user how many applications they want to enter (10 max.).
- Inputs a persons full name, social security number, full address, total yearly salary, and total monthly debt payment into a separate dynamically created person record for each application.
- Inputs a target loan amount, interest rate, and load duration into a separate dynamically created application record for each application.
- Calculates a mortgage payment based on the target loan amount, interest rate, and loan duration.
- Calculates the person's current debt ratio, and the debt ratio with the mortgage.
- Qualifies or rejects the person based on standard 28% mortgage, 36% total debt ratio.
- Prints a simple mortgage applications report. The report should include a summary for each application, number of approved applications, and number of rejected applications.

Recursive Function Calls

An overview of recursive function calls.

- A recursive function call is a call to the same function that is currently processing.
- When a recursive function call is made, a new separate copy of all function data is placed on the stack, and the function code starts at its beginning with the new data.
- The process of placing new copies of the function data on the stack is known as stack winding.
- When the recursed version of the function returns, the separate copy of all function data is removed from the stack, and execution control continues with the line following the recursion call in the original function.
- The process of removing the new copy of function data from the stack is known as stack unwinding.
- It is important to remember that the recursed execution is completely separate and distinct from the original, and with each recurse finite stack space is used.
- A recursive function must have a condition test that stops the recursive cycle, or the program will enter an infinite recursive loop state.
- An infinite recursive loop state will lead to stack overflow, and program termination.
- Recursive function calls are difficult to write, debug, and maintain.
- There are no program circumstances that require recursive function calls, and all recursive calls can be implemented in an iterative form.
- Unless significant coding can be saved, and circumstances are completely controlled, do not use recursive function calls.

Examples.

- See example disk for program RECUREXM.C.

Quick Test.

1. If a local variable called x is created and set to zero in a function, and during the function's execution x is later set to five, and a recursive call to the function is made, what is the starting value of x in the recursed version?
2. How is a recursive function call made?
3. What is stack winding?
4. What is stack unwinding?
5. What is a danger associated with recursion?
6. Is recursion ever required?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

An Introduction to Files

Three basic file types.

- There are three basic file types or implementations: text files, flat binary files, and complex binary files.
- A look at text files follows:
  - Text files are mostly used to store simple text documents, status logs, or beginner level programs.
  - Text files contain only printable ASCII characters.
  - Text files are organized into lines of variable length, and each line ends with a carriage return and line feed.
  - Numeric data is stored as ASCII strings.
  - End of file is determined by an end of file character.
  - Because all information is stored as text, it is cumbersome to implement fixed record length text files.
  - The file is read one character at a time and is logically parsed into components.
  - The numeric data is converted to true hexadecimal form, and then is stored in memory.
  - Because the file is read character by character, and extensive translation is performed, text styles file are very slow to process.
  - Text files are typically accessed in a read only, write only, or append mode.
  - Because the lines of the file vary in length, text files are very seldom used for read- write access.
  - Normally, a text file is rewritten from scratch if it needs to be changed.
- A look at flat binary files follows:
  - Flat binary files, although much better at storing data than text files, are used in low complexity applications to hold a variety of record style data.
  - Flat binary files will store information in hexadecimal format just as it appears in memory.
  - Flat binary files have no carriage return, line feed, or end of file characters.
  - Because all the information is stored as it would be in memory, the size of each record in a flat binary file is the same size throughout the file.
  - Flat binary files make implementation of read-write access very simple.
  - Flat binary files are not translated byte by byte, so they are much faster to read than text files.
  - Flat binary files are used with any access.
- A look at complex binary files follows:
  - Complex binary files are normally implemented in complex applications, where flexible high performance retrieval methods are required.
  - The files are the basis for all modern databases.
  - Complex binary files have many of the same characteristics as flat binary files.
  - A complex binary file is made from an inter-supporting mixture for different record types.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- Normally the different records come together to form a data structure such as a linked list or binary tree.
- The file will typically have overhead records that manages and even describes the format of the file.
- The program accessing the file needs to be written to handle the complex format.
- The complexity of the file will often require the use of specialized function libraries to be implemented by the program.
- As the popularity of a particular file format grows it may gain industry support as a de facto open architecture standard.

File names and paths.

- What is commonly referred to as a file name is made of several parts.

```
[drive letter]:[path]\[stub].[extension]
```

  - The combination of the stub and the extension is called a file name.
  - The combination of the file name and the path is called a relative path name.
  - The combination of the relative path name and the drive letter is called the full path name.
  - Other notation can be used in paths
    - "..\" is used to refer to the parent directory.
    - ".\" is used to refer to the current directory.
- Some examples of path names follow:

```
c:\bc\bin\myfile.c
\bc\bin\myfile.c
..\bin\myfile.c
.\myfile.c
c:myfile.c
```

How DOS deal with files.

- For purposes of speed and organization, the smallest amount of disk space that can be allocated is a cluster.
- DOS allocates space one cluster at a time.
- Regardless of the size of the file, DOS allocates the size of the file rounded up to the nearest cluster.
- Even if a file is only one byte long, it still takes up one full cluster of disk space.
- The size of a cluster varies from drive to drive.
- The minimum size of a cluster is 2K. The current maximum size of a cluster is 16K.
- On a 2K cluster drive, a one byte file will use the same space as a 2K file.
- If a program crashes or ends without closing a file it is writing to, the clusters written will not be marked as part of the file. The cluster will be in a non-allocated, and non-free state, and will be labeled lost.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- Always close files as soon as possible.
- Every directory entry takes up one cluster.

Quick Test.

1. What is a typical use for a text file?
2. How are end of file characters used in binary files?
3. What types of files are used for read-write access?
4. How does a cluster become lost?
5. How much space does a five byte file really take up?
6. What type of file structure does dBase use?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Accessing Files with Standard I/O

The background on the files and standard I/O.

- Like standard input and output, standard file I/O is stream based.
- The lack of stream based I/O is that it does not support more modern access concepts, such as file sharing.
- Stream I/O does support text files, and flat binary files.
- Although stream I/O cannot perform all functionality required, it can be used to implement some complex binary files.
- To work with stream file I/O, a pointer to the type defined structure FILE must be declared.

```
FILE* input_file = NULL;
```

- The file pointer is assigned to a newly opened file, and is used as an argument to all I/O function calls.

```
input_file = fopen("MYFILE.TXT","rt");
```

- With stream I/O, a relocation must occur when switching between reading and writing.

Opening and closing files.

- FILE *fopen(const char *filename, const char *mode);
  - Fopen() is used to open a file for stream I/O.
  - The path name of the file is passed as filename.
  - The method of accessing the file is passed as mode.
  - The string that mode points to can have the following codes:
    - File access codes.

      r Reading (file must exist).
      w Writing (file is created).
      a Appending (if file does not exist, it is created).
      r+ Reading and writing (file must exist).
      w+ Writing and reading (file is created).
      a+ Appending and reading (if file does not exist, it is created).
    - File type codes.

      t Text file.
      b Binary file.
    - The codes would be used in strings by specifying access and type.
    - Some examples of mode strings follow:

      "rt" Read text file
      "wb" Write binary file
      "w+b" Write and read binary file
      "w+t" Write and read text file.
      "ab" Append binary file.

  - Fopen() returns a pointer to the FILE structure used to access the file. If the file could not be opened, NULL is returned.
  - An example of fopen() follows:

```
FILE* input_file = fopen("C:\\Names.dat","r+b");
```

- FILE *freopen(const char *filename, const char *mode, FILE *stream);
  - freopen is used to change the file or the mode referenced by a file pointer.
  - The current file is always closed before it or a new file is opened.
  - The file to be opened is passed as filename.
  - The method of accessing the file is passed as mode.
  - The current file pointer is passed as stream.
  - Freopen() returns stream. If the file could not be opened, NULL is returned.
  - An example of freopen() follows:

```
myfptr = freopen("MYFILE.TXT", "w+b", myfptr);
```

- int fclose(FILE *stream);
  - Fclose() is used to close an opened file.
  - The pointer of the file to be closed is passed as stream.
  - Fclose() returns zero on success. If any errors occurred, EOF is returned.
  - An example of fclose() follows:

```
FILE* myfptr = fopen("MYFILE.TXT", "w+b");
...
fclose(myfptr);
```

- int feof(FILE *stream);
  - Feof() is used to determine if the stream is at end of file.
  - The file pointer to be tested is passed as stream.
  - Feof() returns "true" if the stream is at the end of file, and "false" if it is not.
  - An example of feof() follows:

```
FILE* myfptr = fopen("MYFILE.TXT", "w+b");
...
if (feof(myfptr))
{ puts("at end of file");
}
```

- int ferror(FILE *stream);
  - Ferror() is used to test a file for any I/O errors.
  - The file pointer to be tested is passed as stream.
  - Ferror() returns "true" if the file has an error, and "false" if it does not.
  - An example of ferror() follows:

```
FILE* myfptr = fopen("MYFILE.TXT", "w+b");
...
if (ferror(myfptr))
{ puts("File I/O error...");
}
```

- void clearerr(FILE *stream);
  - Clearerr() is used to reset any I/O errors.
  - The file pointer to be reset is passed as stream.
  - An example of clearerr() follows:

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

```
FILE* myfptr = fopen("MYFILE.TXT", "w+b");
...
if (ferror(myfptr))
{ puts("File I/O error...");
  clearerr(myfptr);
}
```

Working text files.

- There are a variety of functions that are typically used to access text files.
- char *fgets(char *s, int n, FILE *stream);
  - Fgets() is used to read a string from a file.
  - The string used to hold the data read from the file is passed as s.
  - The maximum number of characters to read into s, including the NULL, is passed as n.
  - The file to read is passed as stream.
  - Fgets() stops reading when it has reached the end of line, or when it has read n - 1 characters.
  - If fgets() is stopped by a new line character, it retains the new line character in s.
  - Fgets() NULL terminates s.
  - Fgets() returns s if it is successful, and NULL if it is not.
  - An example of fgets() follows:

```
FILE* myfile = fopen("MYFILE.TXT","rt");
char s[256] = "";
if (myfile)
  fgets(s,256,myfile);
```

- int fputs(const char *s, FILE *stream);
  - Fputs() is used to write a string to a file.
  - The string used to hold the data to write to the file is passed as s.
  - The file to write is passed as stream.
  - Fputs writes only the string to the file.
  - The NULL terminator is not written.
  - Fputs() returns a non-negative value if it is successful, and EOF if it is not.
  - An example of fputs() follows:

```
FILE* myfile = fopen("MYFILE.TXT","wt");
char s[256] = "Hello";
if (myfile)
  fputs(s,myfile);
```

- int fscanf(FILE *stream, const char *format, [address...]);
  - Fscanf() is used to read a series of data from a file.
  - The file to be read is passed as stream.
  - The format string and address arguments are used just like in scanf().
  - Fscanf() returns the number of fields read if it is successful, and NULL if it is not.
  - An example of fscanf() follows:


```
FILE* myfile = fopen("MYFILE.TXT","rt");
char s[256] = "";
if (myfile)
  fscanf(myfile,"%s",s);
```

- int fprintf(FILE *stream, const char *format[, argument, ...]);
  - Fprintf() is used to write a series of data to a file.
  - The file to be written is passed as stream.
  - The format string and address arguments are used just like in printf().
  - Fprintf() returns the number of bytes written if it is successful, and EOF if it is not.
  - An example of fprintf() follows:

```
FILE* myfile = fopen("MYFILE.TXT","wt");
char s[256] = "Hello";
if (myfile)
  fprintf(myfile,"%s\n",s);
```

- int fgetc(FILE *stream);
  - Fgetc() is used to read a character from a file.
  - The file to be read is passed as stream.
  - Fgetc() returns the character read if it is successful, and EOF if it is not.
  - An example of fgetc() follows:

```
FILE* myfile = fopen("MYFILE.TXT","rt");
char s = 0;
if (myfile)
  s = fgetc(myfile);
```

- int fputc(int c FILE *stream);
  - Fputc() is used to write a character to a file.
  - The file to be written is passed as stream.
  - The character to be written is passed as c.
  - Fputc() returns the character written if it is successful, and EOF if it is not.
  - An example of fputc() follows:

```
FILE* myfile = fopen("MYFILE.TXT","wt");
char s = 'A';
if (myfile)
  fputc(s,myfile);
```

- void rewind(FILE *stream);
  - Rewind() to used to reset a file, and place the file location pointer to the beginning of the file.
  - The file to be reset is passed as stream.
  - An example of rewind() follows:

```
FILE* myfile = fopen("MYFILE.TXT","rt");
...
rewind(myfile);
```

Working binary files.

- There are only a few functions necessary to work with binary files.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- size_t fread(void *ptr, size_t size, ,size_t n, FILE *stream);
  - Fread is used to read from a file.
  - The buffer that will hold the data read is passed as ptr.
  - The size of the item to read is passed as size.
  - The number of items to read is passed as n.
  - The file to read is passed as stream.
  - Fread() reads n * size bytes, and the buffer must be large enough to hold the data.
  - Fread() returns the number of items read if it was successful, or EOF or less than n if not.
  - An example of fread() follows:

```
FILE* myfile = fopen("MYFILE.TXT","rb");
char s[256] = "";
if (myfile)
  fread(s,sizeof(s),1,myfile);
```

- size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream);
  - Fwrite is used to write to a file.
  - The buffer that will hold the data to write is passed as ptr.
  - The size of the item to write is passed as size.
  - The number of items to write is passed as n.
  - The file to write is passed as stream.
  - Fwrite() writes n * size bytes, and the buffer must be large enough to hold the data.
  - Fwrite() returns the number of items written if it was successful, or less than n if not.
  - An example of fwrite() follows:

```
FILE* myfile = fopen("MYFILE.TXT","wb");
char s[256] = "Hello";
if (myfile)
  fwrite(s,sizeof(s),1,myfile);
```

- long int ftell(FILE *stream);
  - Ftell() is used to get the current file location pointer.
  - The file to use is passed as stream.
  - The location is the number of bytes from the start of the file.
  - Ftell() returns the location if it is successful, and -1 if not.
  - An example of ftell() follows:

```
FILE* myfile = fopen("MYFILE.TXT","r+b");
printf("at %ld",ftell(myfile));
```

- int fseek(FILE *stream, long offset, int whence);
  - Fseek() is used to set the file location pointer.
  - The file to use is passed as stream.
  - The offset, in bytes, to go to is passed as offset.
  - The starting point of the offset is passed as whence.
  - Whence can have the following values:

SEEK_SET	The beginning of the file.
SEEK_CUR	The current spot in the file.
SEEK_END	The end of the file.

  - Fseek() returns zero if successful, and non-zero if it is not.

- An example of fseek() follows:

```
FILE* myfile = fopen("MYFILE.TXT","r+b");
fseek(myfile,0,SEEK_SET);
```

What needs to be done to use standard input and output.

- To use any stream I/O functions, just include stdio.h in the program by placing a #include <stdio.h> at the top of the file, then link with the standard library.

Examples.

- See example disk for program SFILEEXM.C.

Quick Test.

1. What type of file should be written with fprintf()?
2. What mode is "r+b"?
3. How can an error be reset?
4. What does SEEK_CUR mean?
5. Can a read be made directly after a write?
6. What does a mode of "w+b" do to an existing file?

Assignment.

Write a program that:

- Processes simple mortgage applications.
- Asks the user how many applications they want to enter (10 max.).
- Inputs a persons full name, social security number, full address, total yearly salary, and total monthly debt payment into a dynamically created person record and write each application from the record to a file.
- Inputs a target loan amount, interest rate, and loan duration into a dynamically created application record and write each application from the record to a file.
- Calculates a mortgage payment based on the target loan amount, interest rate, and loan duration.
- Calculates the person's current debt ratio, and the debt ratio with the mortgage.
- Qualifies or rejects the person based on standard 28% mortgage, 36% total debt ratio.
- Prints a simple mortgage applications report. The report should include a summary for each application, number of approved applications, and number of rejected applications.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Accessing Files with UNIX Style I/O

The background on the UNIX style file I/O.

- UNIX style I/O is based on the operating system.
- The benefit of UNIX style I/O is that it allows for all modern access concepts, such as file sharing.
- UNIX style I/O is primarily focused on binary files.
- UNIX style I/O can be used to implement any complex file structure.
- To work with UNIX style I/O, an integer must be declared.

```
int input_file = 0;
```

- The integer is used to store the handle that is assigned by the operating system.
- The handle is passed as an argument to all I/O function calls.

```
input_file = open("MYFILE.TXT",O_BINARY|O_RDWR);
```

Opening and closing files.

- int open(const char *path, int access [, unsigned mode]);
  - Open() is used to open a file.
  - The path name of the file is passed as path.
  - The access method is passed as access
  - The access is specified as an integer that is derived by the use of a bit level or.
    - The following is a list of access constants:

      O_RDONLY Reading only.
      O_WRONLY Writing only.
      O_RDWR Reading and writing.
      O_APPEND Appending.
      O_CREAT Create file only if it does not exist.
      O_TRUNC Truncate the file if is exists.
      O_BINARY Open in binary mode.
      O_TEXT Open in text mode.
  - If the file is to be created, one or both of the following flags must be passed as mode. They set the starting file attributes.

    S_IWRITE Permission to write.
    S_IREAD Permission to read.
  - Open() returns a non-negative integer file handle if it was successful, or a -1 if there was an error.
  - An example of open() follows:

```
int input_file = open("MYFILE.TXT",
                      O_BINARY|O_CREAT|O_WRONLY,
                      S_IWRITE|S_IREAD);
```

- int sopen(char *path, int access, int shflag[, int mode]);
  - Sopen() is used to open a file with file sharing.
  - Sopen() functions like open() with the exception of a share flag that is passed before mode.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

  - The share flag is passed as shflag, and is an integer value.
  - Values for the share flag follow:

SH_DENYRW	Deny read and write.
SH_DENYWR	Deny write.
SH_DENYRD	Deny read.
SH_DENYNONE	Allows read and write.

  - Sopen() returns a non-negative integer file handle if it was successful, or a -1 if there was an error.
  - Share must be loaded to use sopen().
  - An example of sopen() follows:

```
int input_file = sopen("MYFILE.TXT",
                       O_BINARY|O_RDONLY,
                       SH_DENYWR);
```

- int close(int handle);
  - Close() is used to close an opened file.
  - The handle of the file to be closed is passed as handle.
  - Close() returns zero on success. If any errors occurred, -1 is returned.
  - An example of close() follows:

```
int* myfile = open("MYFILE.TXT", O_BINARY|O_RDONLY);
...
close(myfile);
```

- int eof(int handle);
  - Eof() is used to determine if the file is at end of file.
  - The file handle to be tested is passed as handle.
  - Eof() returns 1 if the file is at the end of file, 0 if it is not, and -1 if there was an error.
  - An example of eof() follows:

```
int* myfile = open("MYFILE.TXT", O_BINARY|O_RDONLY);
...
if (eof(myfile)>0)
{ puts("at end of file");
}
```

- int chsize(int handle, long size);
  - Chsize() is used to adjust the size of a file.
  - The file to be adjusted is passed as handle.
  - The new size of the file, in bytes, is passed as size.
  - Chsize() returns zero if successful, and -1 if not.
  - An example of chsize() follows:

```
int* myfile = open("MYFILE.TXT", O_BINARY|O_WRONLY);
chsize(myfile,0);
```

- long filelength(int handle);
  - Filelength() is used to get the size of a file in bytes.
  - The file to be tested is passed as handle.
  - Filelength() returns the size of the file if successful, and -1 if not.
  - An example of filelength() follows:

```
int* myfile = open("MYFILE.TXT", O_BINARY|O_RDONLY);
```



```
printf("size=%ld\n",filelength(myfile));
```

Working binary files.

- int read(int handle, void *buf, unsigned len);
  - Read is used to read from a file.
  - The file to read is passed as handle.
  - The buffer that will hold the data read is passed as buf.
  - The size of the item to read is passed as len.
  - Read() reads n bytes, and the buffer must be large enough to hold the data.
  - Read() returns the number of bytes read if it was successful, zero if EOF, or -1 if error.
  - An example of read() follows:

```
int myfile = open("MYFILE.TXT",O_BINARY|O_RDONLY);
char s[256] = "";
if (myfile != -1)
  read(myfile,s,sizeof(s));
```

- int write(int handle, void *buf, unsigned len);
  - Write is used to write to a file.
  - The file to write is passed as handle.
  - The buffer that will hold the data to write is passed as buf.
  - The size of the item to write is passed as len.
  - Write() writes n bytes, and the buffer must be large enough to hold the data.
  - Write() returns the number of bytes written if it was successful, or -1 if there was an error.
  - An example of write() follows:

```
int myfile = open("MYFILE.TXT",O_BINARY|O_WRONLY);
char s[256] = "Hello";
if (myfile != -1)
  write(myfile,s,sizeof(s));
```

- long tell(int handle);
  - Tell() is used to get the current file location pointer.
  - The file to use is passed as handle.
  - The location is the number of bytes from the start of the file.
  - Tell() returns the location if it is successful, and -1 if not.
  - An example of tell() follows:

```
int myfile = open("MYFILE.TXT",O_BINARY|O_RDONLY);
printf("at %ld",tell(myfile));
```

- long lseek(int handle, long offset, int fromwhere);
  - Lseek() is used to set the file location pointer.
  - The file to use is passed as stream.
  - The offset, in bytes, to go to is passed as offset.
  - The starting point of the offset is passed as fromwhere.
  - Fromwhere can have the same values as whence in fseek().
  - Lseek() returns zero if successful, and -1 if it is not.
  - An example of lseek() follows:

```
int myfile = open("MYFILE.TXT",O_BINARY|O_RDONLY);
lseek(myfile,0,SEEK_SET);
```


www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- int lock(int handle, long offset, long length);
  - Lock() is used to lock any part of a file when using file sharing.
  - A lock will block any other program from reading or writing to the file.
  - The file to be locked is passed as handle.
  - The starting byte of the lock is passed as offset.
  - The number of bytes to lock is passed as length.
  - All locks must be removed before the file is closed.
  - The entire file can be locked.
  - Lock() returns 0 if successful, and -1 if not.
  - An example of lock() follows:

```
int myfile = open("MYFILE.TXT",O_BINARY|O_RDONLY);
lock(myfile,0,15);
```

- int unlock(int handle, long offset, long length);
  - Unlock is used to remove a lock that was previously placed on the file with lock().
  - The file to be unlocked is passed as handle.
  - The starting byte of the unlock is passed as offset.
  - The number of bytes to unlock is passed as length.
  - Unlock() returns 0 if successful, and -1 if not.
  - An example of unlock() follows:

```
int myfile = open("MYFILE.TXT",O_BINARY|O_RDONLY);
if (!lock(myfile,0,15))
  unlock(myfile,0,15);
```

What needs to be done to use UNIX style I/O.

- To use any UNIX I/O functions, just include io.h, sys\stat.h, and fcntl.h in the program by placing a #include for each at the top of the file, then link with the standard library.

Examples.

- See example disk for program UFILEEXM.C.

Quick Test.

1. How does UNIX I/O differ from stream I/O?
2. When does S_IWRITE or S_IREAD need to be used?
3. Which function is used to open a file for sharing?
4. What does SEEK_END mean?
5. What needs to be done before a file is closed if lock was used?
6. How can a file's size be changed?

Assignment.

Write a program that:

- Processes simple mortgage applications from more than one workstation.
- The workstations should share the same file, and locking should be used.
- Asks the user how many applications they want to enter (10 max.).
- Inputs a persons full name, social security number, full address, total yearly salary, and total monthly debt payment into a dynamically created person record, and write each application from the record to a file.
- Inputs a target loan amount, interest rate, and loan duration into a dynamically created application record, and write each application from the record to a file.
- Calculates a mortgage payment based on the target loan amount, interest rate, and loan duration.
- Calculates the person's current debt ratio, and the debt ratio with the mortgage.
- Qualifies or rejects the person based on standard 28% mortgage, 36% total debt ratio.
- Prints a simple mortgage applications report. The report should include a summary for each application, number of approved applications, and number of rejected applications.
- Keep it simple.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Implementing Complex Data Structures Dynamically

Leveraging memory with data structures.

- Storing information as large structures in memory is problematic.
- Large chucks of memory are often difficult to allocate, and in some programming situations, can be cumbersome to handle.
- Large structures can loose focus, and degrade program modularity.
- Often structures can be broken into smaller more focused structures that only require small allocations.
- Smaller allocations suffer less from the affects of heap fragmentation.
- Smaller allocations fit more easily into memory, and can be allocated and freed as needed.

Pointers to pointers to pointers.

- A problem when working with many smaller structures is organizing them in memory.
- The addresses of the smaller structures are normally combined into a structure that is used to manage them as a package.
- The package structures can often contain other data.
- It is not uncommon to package the addresses of the packages.
- This packaging technique acts as a composition; each package is composed of data and pointers that point to smaller in independed packages.
- The rule when dealing with a composition is a component can not be used until all sub-components are in place.
- Compositions must be built from the sub-components up, and must be destroyed from the sub-components up.

Starting with linked lists.

- Many times the data needs to be organized into a listing or table.
- Although arrays can be dynamically implemented, arrays suffer large storage requirements, are fixed in size, and are poor implementations for searching.
- Linked lists are a better implementations for lists or tables.
- Linked lists are composed of small structures that form a list by pointing to the next element.
- Linked lists are not fixed in size, and can grow to fill all available memory.
- Like any other dynamic implementation, a pointer to the start of the list needs to be stored or reached from the data or stack segment.
- The list is built and destroyed one element at a time.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- A linked list requires a minimum of a first and current pointer as well as a pointer to the next element included in the element's structure.
- Often linked lists are sorted, and are used for primitive indexes.
- Linked lists can be linked in any number of orders.
- Although linked lists are an improvement from a storage and organizational aspect, other data structures, such as binary trees, are better for searching.

Examples.

- See example disk for program LISTSEXM.C.

Quick Test.

1. What impact do large structures have on storage?
2. Does the order of construction matter when dealing with compositions?
3. How should a composition be destroyed?
4. What are the benefits of smaller structures?
5. What would be an implementation for linked lists?
6. What pointers are required for a simple linked list?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Pointer Arithmetic

General overview of pointer arithmetic.

- Pointers can be altered using basic increments and decrements.
- Pointers will increase or decrease in value by the size of the data they point to.
- Pointer arithmetic can be used to move through arrays, or any memory structure.
- Only basic increments and decrements are valid with pointers, and no calculations are valid.
- Pointers can be incremented by numbers greater than one, but the pointer's value change will be the number times the size of the pointer's data.
- Pointers can be compared only for equality.
- Before a pointer's value is changed be sure its initial value is preserved.
- Void pointers must be type cast to a real pointer type before they can altered.
- When a type cast is used on a pointer, the pointer behaves as the cast type.

Examples.

- See example disk for program PNTRSEXM.C.

Quick Test.

1. If a integer pointer is incremented by two, what does the pointer value change by?
2. What is a basic use for pointer arithmetic?
3. Can multiplication be done with pointers?
4. What needs to be done when working with void pointers?
5. Can pointers be used with less than and greater than operators?
6. How does type casting play a role in pointer arithmetic?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Console Style Screen Output

The background on console style I/O.
- A computer display can handle many other output techniques than stream style I/O.
- Console style I/O allows for more advanced formatting and output.
- With console I/O windows can be made, colors can be set, and text can be manipulated.
- The availability of console I/O varies from environment to environment.
- Console I/O is based on a graph like X-Y system.
- All action take place in a window.
- The default window is the full screen.
- Console I/O is not affected by O/S redirection.
- Flushing has no effect with console I/O.
- Stream I/O calls do not work with console settings.
- Console I/O calls should never be mixed with stream I/O calls.

Displaying information with console output functions.
- int cprintf(const char *format[, argument, ...]);
  - Cprintf() prints formatted output to the screen.
  - Cprintf() is used as printf() is, but works with console settings.
- int cputs(const char *str);
  - Cputs() prints a string to the screen.
  - Cputs() is used as puts() is, but works with console settings.

Accepting data using console input functions.
- char *cgets(char *str);
  - Cgets() gets a string from the keyboard.
  - Cgets() is used as gets() is, but works with console settings.
- int cscanf(char *format[, address, ...]);
  - Cscanf() is used to get formatted input from the keyboard.
  - Cscanf() is used as scanf() is, but works with console settings.
- int getch(void);
  - Getch() gets a character from the keyboard and does not echo it to the screen.
  - The character inputted is returned by getch().
- int getche(void);
  - Getche() gets a character from the keyboard and echoes it to the screen.
  - The character inputted is returned by getche().

An overview of other console I/O functions.

- void clreol(void);
  - Clreol() clears to the end of the current line.
- void clrscr(void);
  - Clrscr() clears the current windows.
- void delline(void);
  - Delline() deletes the current line from the current window.
  - Lines below the deleted line move up.
- int gettext(int left, int top, int right, int bottom, void *destin);
  - Gettext() retrieves a block of text from the current window.
  - The top left corner of the block is specified by left and top.
  - The bottom right corner of the block is specified by right and bottom.
  - The buffer that will hold the block is passed as destin, and must be large enough to hold the block. Each character will need two bytes.
- void gotoxy(int x, int y);
  - Gotoxy() sets the current row and column for console actions.
  - The column is passed as x, and the row is passed as y.
- void insline(void);
  - Insline() inserts a blank line into the current window at the current row.
  - All lines below the new line are moved down.
- int movetext(int left, int top, int right, int bottom, int destleft, int desttop);
  - Movetext() moves a block of text to a new location within the current window.
  - The top left corner of the block is specified by left and top.
  - The bottom right corner of the block is specified by right and bottom.
  - The new top left corner is specified by destleft and desttop.
- int puttext(int left, int top, int right, int bottom, void *source);
  - Puttext() places a block of text into the current window.
  - The top left corner of the block is specified by left and top.
  - The bottom right corner of the block is specified by right and bottom.
  - The buffer that contains the block is passed as destin, and must be large enough to hold the block. Each character will need two bytes.
- void textbackground(int newcolor);
  - Textbackground() sets a new background color.
  - The new color is passed as new color.
  - The value for a background color must be within 0 and 7.
- void textcolor(int newcolor);
  - Textcolor() sets a new foreground color.
  - The new color is passed as new color.
- Available colors include:

BLACK	0	BLUE	1
GREEN	2	CYAN	3
RED	4	MAGENTA	5
BROWN	6	LIGHTGRAY	7
DARKGRAY	8	LIGHTBLUE	9

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

LIGHTGREEN	10	LIGHTCYAN	11
LIGHTRED	12	LIGHTMAGENTA	13
YELLOW	14	WHITE	15
BLINK	128		

- void window(int left, int top, int right, int bottom);
  - Windows changes the size of the active window.
  - The size of the windows is set by left, top, right, and bottom.

What needs to be done to use console input and output.

- To use any console I/O functions, just include conio.h in the program by placing a #include <conio.h> at the top of the file, then link with the standard library.

Examples.

- See example disk for program CONIOEXM.C.

Quick Test.

1. Can printf() and cprintf() be used at the same time?
2. How can a block of text be moved on the screen?
3. If no window is specified, what size window is available?
4. What is the difference between getch() and getche()?
5. How many bytes of storage are needed to get a 10x5 block of text?
6. Can LIGHTRED be used for a background color?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Memory Models

The background on memory models.

- Memory models are a compiler option.
- Memory models control the way a program occupies memory.
- The model a program compiles with must match the model of the libraries the program is trying to use.
- If the program memory model does not match the library's model, linkage errors will occur.
- The model that a program uses will determine how the program will perform, and how much memory the program will occupy.
- There are six models available, and each model has its purpose.
- The models include: tiny, small, medium, compact, large, and huge.

The memory models available.

- Tiny is the smallest memory model.
  - All code, data, stack, and local heap fit into one segment.
  - Tiny programs can be made into a .COM format by using /t in linkage.
  - All pointers are near.
  - In today's environment this model is rarely used.
- Small is the most common training model, and can be used for many DOS applications.
  - The program code is placed in its own segment.
  - The data, stack, and local heap are placed as a d-group in a common segment.
  - The program has access to the far heap.
  - Small is good for programs with low data requirements, and moderate code requirements.
- Medium is a model that allows large amounts of program code.
  - The program code is placed in one or more segments, and can occupy up to one megabyte.
  - The data, stack, and local heap are placed as a d-group in a common segment.
  - The program has access to the far heap.
  - All calls are far.
  - Medium is good for programs with low data requirements and large code requirements.
- Compact is a good all purpose model for programs heavy on data and code.
  - The program code is placed in its own segment.
  - There is no d-group or local heap.
  - The data is placed in its own segment.
  - The stack is placed in its own segment.
  - The program has access to the far heap.

  - Compact is good for programs with moderate amounts of data and code.
- Large is the most common model used in professional applications.
  - The program code is placed in one or more segments, and can occupy up to one megabyte.
  - The program data is placed in one or more segments, and can occupy up to one megabyte.
  - The stack is placed in its own segment.
  - The program has access to the far heap.
  - All calls are far, and all data is far.
  - No non-dynamic data can occupy more than 64K.
  - Large is good for programs that have large amounts of data and code.
- Huge is rarely required, and is only used if there are unusual data requirements.
  - The program code is placed in one or more segments, and can occupy up to one megabyte.
  - The program data is placed in one or more segments, and can occupy up to one megabyte.
  - The stack is placed in its own segment.
  - The program has access to the far heap.
  - All calls are far, and all data is far.
  - Non-dynamic data can occupy up to one megabyte.
  - Huge is good for programs that have very large pieces of storage.

The effect of memory models on a programs size and speed.

- As a rule the larger the model the larger and slower the application will be.
- Models that have all data, and code in single segments will have the fastest execution time.
- Models that allow multiple code segments will have higher call overhead and slower calls.
- Models that allow multiple data segments will have higher reference overhead and slower references.

Quick Test.

1. Which memory model would likely be the slowest executing?
2. If a program had 63K code, 43K data, and 33K stack, which model is best?
3. How do models affect the use of libraries?
4. What are memory models a function of?
5. When is medium used versus compact?
6. Which model is faster, large or compact?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Passing Information Via the Command Line

The background on command line arguments.

- Information can be passed into an application from the command line as arguments.
- Command line arguments can be used for anything from controlling the processing of an application, to providing basic information such as file paths.
- Command line arguments are listed after the program name on the command line.
- Each argument is separated by a space, so to pass an argument that contains a space, enclose the argument in double quotes.

```
xcopy C:\datafile A:\
ts "Sam Smith"
```

The syntax for using command line arguments.

- To implement command line arguments, add two arguments to the main() of the program.
- The arguments are:
  - int argc
    - Argc is an integer that contains the total number of arguments passed to the application.
    - The program's name is counted, so the number will never be less than one.
  - char* argv[]
    - Argv is an array of character pointers that point to each argument.
    - The arguments are stored in the array in the order they were placed on the command line, and start with the program name as the first argument.
    - The arguments are text strings, and need to be converted to other types if required.
    - Before modifying the arguments they should be copied to a separate working location.
  - Argc and Argv are local to main, but there are global copies available from dos.h as _argc and _argv.
- A short example of a main() that is accepting arguments follows:

```
void main(int argc, char* argv[])
{ int i = 0;
  for (i = 0; i < argc; i++)
    puts(argv[i]);
}
```

Error trapping and implementing command line arguments.

- Detecting invalid command line arguments can be done in two ways.
- Argc can be tested to verify that the proper number of arguments were passed.
- The values can be compared with what the program expects.

- A good way of implementing command line arguments is with the use of flags.
- Flags are pre-defined code sequences that the user attaches to the command line argument.
- Normally flags start with a '-' or a '/' character followed by a symbol and value.

```
-imyfile.txt
-ddebug
/help
/menu
```

- The program would need to interpret the flags and carry out the required processing.
- Any unrecognized flags could be ignored by the program or an error message could be printed.
- Flags give a program the ability to positively identify what information a user is passing.
- Flags allow the order of the arguments to vary, and give the user a feeling of control.
- If a program detects an error with a command line argument it should print a usage message.

```
void main(int argc, char* argv[])
{ if (argc == 1)
    puts("Usage: ...");
}
```

- All programs should identify the basic symbols for help "/help", "-help" , "/?", and "-?". When help is requested the program should respond with basic instructions.
- The DOS allows for a command line of only 127 characters; consequently, large strings of arguments need to be placed in a configuration file, and the program is passed the name of the configuration file to be read.
- The reading of the configuration file is a manual process. Like any other file access in C, the program is responsible for reading it.

Examples.

- See example disk for program CARGSEXM.C.

Quick Test.

1. Which argument contains string values?
2. How can a program determine its own file name?
3. When does a configuration file need to be implemented?
4. What is a normal indication that a user requires help?
5. If a number is passed to a program what needs to be done before it is used?
6. How can a function other than main() access the command line arguments?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Execution of Child Processes

The general information on child processes.

- Child executable processes can be started in two basic ways.
  - Spawning is normally used to place the parent process on hold while the child executes.
  - Spawning causes the parent program to free all available memory from its far heap to allow maximum space for the child.
  - Executing is used to permanently remove the parent process from memory, and replace it with the new process. The parent process is removed at the child's execution point.
- Regardless of the manner started, there must be sufficient memory available to house the new program.
- The child program must be found by path name or environmental path search, or the call will fail.

The functions used to evoke a child process.

- The functions used to evoke a child process are generally found in process.h.
- int execl(char *path, char *arg0 *arg1, ..., *argn, NULL);
  - Execl() is used to execute a program from within a program.
  - The program to be executed is passed as path.
  - The arguments are passed individually as arg0, arg1, ..., arg*n*.
  - At least one argument must be passed to the new program, arg0.
  - Arg0 is always the new program's name.
  - A NULL must be passed after the last argument.
  - Execl() returns a -1 on error, and if successful it does not return.
  - Execl() performs no searching for a file, so path must be set to an appropriate value.
- int execlp(char *path, char *arg0,*arg1, ..., *argn, NULL);
  - Execlp() works the same as execl(), but searches the environmental path for the executable.
- int execv(char *path, char *argv[]);
  - Execv() is used to execute a program from within a program.
  - The program to be executed is passed as path.
  - The arguments are passed as an array of character pointers.
  - At least one argument must be passed to the new program, argv[0].
  - Argv[0] is always the new program's name.
  - A NULL must be placed as the last pointer in the array.
  - Execv() returns a -1 on error, and if successful it does not return.
  - Execv() performs no searching for a file, so path must be set to an appropriate value.
- int execvp(char *path, char *argv[]);

  - Execvp() works the same as execv(), but searches the environmental path for the executable.
- int spawnl(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
  - Spawnl() is used to spawn a program from within a program.
  - The mode to use is passed as mode.
  - The available modes include:
    - P_WAIT temporarily stops the execution of the parent process until the child process is completed. Both parent and child are in memory.
    - P_NOWAIT allows both parent and child process to execute simultaneously, and is not available in DOS.
    - P_OVERLAY removes the parent process from memory, just like exec, and replaces it with the child process.
  - The program to be executed is passed as path.
  - The arguments are passed individually as arg0, arg1, ..., arg*n*.
  - At least one argument must be passed to the new program, arg0.
  - Arg0 is always the new program's name.
  - A NULL must be passed after the last argument.
  - Spawnl() returns a -1 on error.
  - Spawnl() performs no searching for a file, so path must be set to an appropriate value.
- int spawnlp(int mode, char *path, char *arg0, arg1, ..., argn, NULL);
  - Spawnlp() works the same as spawnl(), but searches the environmental path for the executable.
- int spawnv(int mode, char *path, char *argv[]);
  - Spawnv() works in a similar manner to execv(), but allows the use of a spawn mode.
- int spawnvp(int mode, char *path, char *argv[]);
  - Spawnvp() works the same as spawnv(), but searches the environmental path for the executable.
- int system(const char *command);
  - System() evokes a copy of the command interpreter and passes command to it.
  - The command to be carried out is passed as command, and its value can be anything the command interpreter can process; even batch files.
  - System() functions in two ways.
    - If command is NULL, system() returns zero if it cannot load the interpreter.
    - If command is not NULL, system() returns zero if it loaded the interpreter successfully.
  - If an error occurs, system() returns -1.
  - System() is found in dos.h.

What needs to be done to use functions that evoke a child process.

- To use exec, spawn, or system, just include process.h or dos.h in the program by placing an #include at the top of the file, then link with the standard library.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Examples.

- See example disk for program CARGSEXM.C.

Quick Test.

1. What happens to the parent if an execv() is used?
2. When does control return to the parent process if P_WAIT is used?
3. What is the difference between execv() spawnv() with P_OVERLAY?
4. What function is used to call a batch file?
5. Which argument always must be passed when using execv() or spawnv()?
6. Which functions can be used to search the environmental path?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Working with DOS

Utilizing environmental variables.

- OS environmental variables can be used for user settings, data paths, etc.
- C can access the OS environmental variables using functions found in stdlib.h
- char *getenv(const char *name);
  - Getenv() can be used to retrieve the value of an environmental variable, or to delete it for the current program.
  - The variable's name is passed as name, and must be capitalized.
  - To delete a variable, pass the variable's name followed by '=' as name.

```
char x[256]="";
strcpy(x,getenv("DATAFILE="));
```

  - The return value of getenv() is a pointer to the value of the variable. If the variable does not exist, NULL is returned.
- int putenv(const char *name);
  - Putenv() is used to change the value of an environmental variable for the current program.
  - To alter a variable, pass the variable's name followed by '=' and a setting as name.

```
putenv("DATAFILE=MYFILE.TXT");
```

  - The return value of putenv() is a 0 on success, or -1 on failure.

Accessing the system time.

- There are times when a program must work with the system date and time, and the functions in dos.h aid in getting the job done.
- void getdate(struct date *datep);
  - Getdate() is used to retrieve the system date.
  - Getdate() stores the date in the structure pointed to by datep.
  - Structure date follows:

```
struct date
{ int da_year;
  char da_day;
  char da_mon;
};
```

- void setdate(struct date *datep);
  - Setdate() is used to set the system date.
  - Setdate() sets the date to the values stored in the structure pointed to by datep.
- An example of setdate() and getdate() follow:

```
struct date mydate;
getdate(&mydate);
mydate.da_year++;
setdate(&mydate);
```

- void gettime(struct time *timep);
  - Gettime() is used to retrieve the system time in 24 hour format.
  - Gettime() stores the time in the structure pointed to by timep.
  - Structure time follows:

```
struct time
{ unsigned char ti_min;
  unsigned char ti_hour;
  unsigned char ti_hund;
  unsigned char ti_sec;
};
```

- void settime(struct time *timep);
  - Settime is used to set the system time in 24 hour format.
  - Settime() sets the time to the values stored in the structure pointed to by timep.
- An example of settime() and gettime() follows:

```
struct time mytime;
gettime(&mytime);
mytime.ti_min == 60 ? mytime.ti_min = 0 :
mytime.ti_min++;
settime(&mytime);
```

- int getftime(int handle, struct ftime *ftimep);
  - Getftime() is used to retrieve a UNIX opened file's time.
  - Gettime() stores the time in the structure pointed to by ftimep.
  - Structure ftime follows:

```
struct ftime
{ unsigned ft_tsec: 5;    /* two seconds */
  unsigned ft_min: 6;     /* minutes */
  unsigned ft_hour: 5;    /* hours */
  unsigned ft_day: 5;     /* days */
  unsigned ft_month: 4;   /* months */
  unsigned ft_year: 7;    /* year - 1980*/
};
```

  - Zero is returned on success, and -1 on failure.
- int setftime(int handle, struct ftime *ftimep);
  - Setftime is used to set a UNIX opened file's time.
  - Setftime() sets the time to the values stored in the structure pointed to by ftimep.
  - Zero is returned on success, and -1 on failure.
- An example if setftime() and getftime() follows:

```
struct ftime mytime;
int myfile = 0;
myfile = open(...);
if (!getftime(&mytime))
{ ...
  setftime(&mytime);
}
```

Working with file level functions.

- Functions that allow file level processing are found in dos.h.
- int _dos_getfileattr(const char *path, unsigned *attribp);
  - _dos_getfileattr() is used to return the DOS file attributes.
  - The path name of the file is passed as path.
  - A pointer to an unsigned integer that will be used to store the attribute is passed as attribp.
  - The OR style attributes follow:

```
_A_RDONLY Read-only attribute
_A_HIDDEN Hidden file
_A_SYSTEM System file
_A_VOLID  Volume label
_A_SUBDIR Directory
_A_ARCH   Archive
_A_NORMAL No bits set
```

  - The function returns zero on success.
- int _dos_setfileattr(const char *path, unsigned attrib);
  - _dos_setfileattr() is used to set the DOS file attributes.
  - The path name of the file is passed as path.
  - An unsigned integer that holds the attribute is passed as attrib.
  - The function returns zero on success.
- An example of _dos_setfileattr() and _dos_getfileattr() follows:

```
unsigned int x = 0;
_dos_getfileattr("MYFILE.TXT",&x);
_dos_setfileattr("MYFILE.TXT",_A_NORMAL);
...
_dos_setfileattr("MYFILE.TXT",x);
```

- int unlink(const char *filename);
  - Unlink() is used to delete a file from disk.
  - The file to be deleted is passed as filename.
  - Unlink() returns zero if the file was deleted.
  - An example of Unlink() follows:

```
if (unlink("myfile.txt"))
  puts("Error deleting file.");
```

- int rename(const char *oldname, const char *newname);
  - Rename() is used to rename a closed file to a new name, and found in stdio.h.
  - The old name is passed as oldname, and the new name is passed as newname.
  - The return value is zero on success.

Handling directories from C.

- Being able to handle directories is important, and functions are found in dir.h.
- int chdir(const char *path);
  - Chdir() is used to change the current directory of the current drive.
  - The new directory is passed as path.

  - Chdir() returns zero on success.
  - An example of chdir() follows:

```
if (chdir("\\MYDIR"))
  puts("No such directory");
```

- int findfirst(const char *pathname, struct ffblk *ffblk, int attrib);
  - Findfirst() is used to search the directory for the first file that matches pathname.
  - The pattern is passed as pathname, and can contain '*' and '?' characters.
  - A pointer to the structure filled by findfirst() with information about the found file is passed a ffblk.
  - Structure ffblk follows:

```
struct ffblk
{ char ff_reserved[21];  /* reserved by DOS */
  char ff_attrib;        /* attributes */
  int ff_ftime;          /* file time */
  int ff_fdate;          /* file date */
  long ff_fsize;         /* file size */
  char ff_name[13];      /* file name */
};
```

  - The attribute found files must match is passed as attrib.
  - Attrib can be:

```
FA_RDONLY Read-only attribute
FA_HIDDEN Hidden file
FA_SYSTEM System file
FA_LABEL  Volume label
FA_DIREC  Directory
FA_ARCH   Archive
```

  - The return value is zero on success.
- int findnext(struct ffblk *ffblk);
  - Findnext() is used only after findfirst() to locate the next file that matches the specifications set by findfirst().
  - Ffblk works just as it does in findfirst();
  - The return value is zero on success.
  - An example of find first follows:

```
struct ffblk myffblk;
if (!findfirst("MYFILE.*",&myffblk,0))
{ puts(myffblk.ff_name);
  while (findnext(&myffblk))
    puts(myffblk.ff_name);
}
```

- int getcurdir(int drive, char *directory);
  - Getcurdir() is used to retrieve the current directory for a given drive.
  - The drive to check is passed as drive (0 = default, 1 = A, 2 = B, 3 = C, ...).
  - Getcurdir() copies the directory into the memory pointed to by directory.
  - Getcurdir() returns zero on success.
  - An example of getcurdir() follows:

```
char x[256] = "";
getcurdir('C' - 'A' + 1,x);
```

- int getdisk(void);
  - Getdisk() is used to return the current drive number.

  - Getdisk() returns the current drive number (A=0, B=1, C=2, ...).
- int mkdir(const char *path);
  - Mkdir() is used to create a new directory.
  - The name of the new directory is passed as path.
  - The return value of mkdir() is zero on success.
- int rmdir(const char *path);
  - Rmdir() is used to remove an empty directory.
  - The name of the directory to be removed is passed as path.
  - The return value of rmdir() is zero on success.
- int setdisk(int drive);
  - Setdisk() is used to change the current drive.
  - The new drive is passed as drive.
  - Setdisk() returns the total number of drives available.
- void getdfree(unsigned char drive, struct dfree *dtable);
  - Getdfree() is used to retrieve the available disk spaces, and is found in dos.h.
  - The drive to be checked is passed as drive (0 = default, 1 = A, 2 = B, 3 = C, ...).
  - Getdfree() fills the dfree structure pointed to by dtable.
  - Structure dfree follows:

```
struct dfree
{ unsigned df_avail;   /* available clusters */
  unsigned df_total;   /* total clusters */
  unsigned df_bsec;    /* bytes per sector */
  unsigned df_sclus;   /* sectors per cluster */
};
```

Other DOS level functions.

- Some miscellaneous functions from stdlib.h include.
- int rand(void);
  - Rand() returns a random number.
- int random(int num);
  - Random() returns a random number less than num.
- void randomize(void);
  - Randomize() starts the random number sequence in a unique location.
- Some miscellaneous functions from dos.h include.
- void delay(unsigned milliseconds);
  - Delay() is used to slow a program.
  - Delay() delays a program by the milliseconds passed as milliseconds.
- void nosound(void);
  - Nosound() turns off a current sound set by sound().
- void sleep(unsigned seconds);
  - Sleep() is used to suspend processing by the number of seconds passed as seconds.

- void sound(unsigned frequency);
  - Sound() is used to generate continuous sound from the speaker.
  - The frequency is passed as frequency.
  - The sound continues until it is stopped by a call to nosound();
- extern int errno;
  - Errno is a variable that is used to track the last processing error.
- extern unsigned char _osmajor;
  - _osmajor is a variable that is set to the major version level of the OS.
- extern unsigned char _osminor;
  - _osminor is a variable that is set to the minor version level of the OS.

What needs to be done to use DOS level functions.

- To use DOS level functions, just include the proper header file in the program by placing an #include at the top of the file, then link with the standard library.

Examples.

- See example disk for program CARGSEXM.C.

Quick Test.

1. How can the cluster size of the current drive be found?
2. How can file attributes be changed?
3. Which function deletes a file from disk?
4. How could the value of an environmental variable be gotten?
5. Which function starts the random number generator in an unique location?
6. How can a file be renamed?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

C++ .mak SDK

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Interrupt Processing

General overview of interrupt processing.

- An interrupt is a process interruption that passes execution control to a specified function.
- The function is by an ordinal value that refers to a table entry containing the address, and not by the function's address directly.
- When the interrupt function completes its task, control is returned to the calling function.
- Interrupts can be logically placed into four groups:
  - Inter-program interrupts are made when the function requesting the interrupt is in the same program as the resulting function.
  - Inter-application interrupts are made when the function requesting the interrupt is in a separate program than the resulting function.
  - OS level interrupts are made when the requested interrupt is a DOS function.
  - BIOS level interrupts are made when the requested interrupt is a BIOS function.
- The table that holds the interrupt function address is called the interrupt table.
  - The location in memory of the interrupt table is 0x00.
  - The interrupt table is an array of addresses. Each position in the array is referred to as an interrupt vector.
  - The ordinal value that is used to call an interrupt function refers to its vector in the interrupt table.
  - The function addresses that are stored in the interrupt table must be correct. Serious failure can result if a stale address is left in the table.
  - The interrupt table is limited to a max. of 256 entries, and most are used by the OS.
- Interrupt routines are special function that are labeled with the key work interrupt.
  - Because interrupt table space is limited, a single interrupt function will often have multiple services that it can perform.
  - The standard way of passing arguments to functions is not valid for ISRs.
  - Registers are the only way to pass data in or out of an interrupt routine.
  - Before the interrupt is generated any data to be passed is placed in registers.
  - Small integer values can be placed in a register directly, but if the data is larger, a pointer to the data is placed in the register.
  - The service requested is also placed in a register.
  - Results are passed back to the calling function by registers.
  - Without instructions on what an ISR is expecting, there is no way of knowing what values placed in what registers will produce what outcome.
- Because there is no standard library function wrapper around the ISRs, processing speed is increased.
- Safety is compromised using ISRs because the overhead that the standard library added to the calls was primarily safety checks.

Calling DOS interrupt services.

- Most user DOS services are found on vector 0x21.
- To call the services provided by interrupt 0x21, a special function is provided.
- int intdos(union REGS *inregs, union REGS *outregs);
- Intdos() is used to provide easier access to the interrupt 0x21.
- The registers values to be set for the interrupt are packaged in the structure REGS, and passed as inregs.
- The registers values set by the ISR are returned in the structure pointed to by outregs.
- Intdos() avoids incorrectly set registers.
- Intdos() sets the machine registers to the values passed by inregs before generating the interrupt.
- When the interrupt returns, intdos() places the new register values into outregs.
- REGS is really a union that allows for the setting of the whole, low, or high portion of the register.
- Structure REGS follows:

```
struct WORDREGS
{ unsigned int ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS
{ unsigned char al, ah, bl, bh, cl, ch, dl, dh;
};

union  REGS
{ struct  WORDREGS x;
  struct  BYTEREGS h;
};
```

Calling Other interrupt services.

- Although intdos() works well for vector 0x21, any interrupt vector can be called with the same convenience; including the high speed BIOS services.
- int int86(int intno, union REGS *inregs, union REGS *outregs);
- Int86() allows a function to generate any interrupt, and still have the registers handled properly.
- The vector to generate is passed as intno.
- The registers values to be set for the interrupt are packaged in the structure REGS, and passed as inregs.
- The registers values set by the ISR are returned in the structure pointed to by outregs.
- Intdos() avoids incorrectly set registers.
- Intdos() sets the machine registers to the values passed by inregs before generating the interrupt.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

- When the interrupt returns, intdos() places the new register values into outregs.

Hooking your code onto an interrupt vector.

- To implement inter-application interrupts a function must have in interrupt vector assigned to it, and must be declared using the keyword interrupt.
- Before a vector can be changed, its original value must be preserved.
- The application function's address is placed in the interrupt table.
- The application function is responsible for detecting calls to it, and calling the old interrupt address when required.
- If the old address is not called, other processes will fail.
- Before the application terminates, it must reset the interrupt table back to its old address value. If the address is not restored other programs will fail.
- void interrupt(*getvect(int interruptno)) ();
  - Getvect() is used to return the current address stored in a vector.
  - The vector number is passed as interruptno.
  - Getvect() returns the address of the vector requested.
- void setvect(int interruptno, void interrupt (*isr) ());
- Setvect() is used to change the address stored in a vector.
  - The vector number is passed as interruptno.
  - The address of the interrupt function is passed as isr.

What needs to be done to use interrupt processing functions.

- To use any interrupt functions, just include DOS.H in the program by placing an #include <dos.h> at the top of the file, then link with the standard library.

Examples.

- See example disk for program INTEREXM.C.

Quick Test.

1. What makes an interrupt call different than a function call?
2. What are the different categories of interrupts?
3. How can an application function generate a BIOS interrupt?
4. What must be done when changing the interrupt table?
5. Why use interrupts, and what risk is incurred?
6. How should DOS interrupts be called?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Libraries and Object Modules

Breaking your programs into object modules.

- Applications are rarely stored in one module.
- Mostly applications are formed from many modules that are compiled separately and linked together.
- There are many ways in which breaking an application into modules is a benefit.
  - Smaller modules are more easily managed, and allow for multiple programmer development.
  - The focus of a smaller module can be tighter and the code is normally more modular.
  - Utility style functions can be separated from application specific modules, and be available to other projects.
  - Compile time will decrease, because only the modules that have changed need to be recompiled.
  - Change can be thoroughly managed, because a clean accounting of what should have changed can be made.
- To implement multiple modules, the header and source modules must be organized properly.
  - Each module will require it own header file that will be included by any other module that makes use of the module.
    - The header file should include any other header files required for the module to compile.
    - All data that needs to be declared in the header file should be declared as external data.
    - Code macros can appear freely in the header file.
    - Only type definitions that are needed by user modules should be included in the header file.
    - Prototypes for all functions that are needed by user modules must be included in the header file, and argument name should be descriptive.
    - Comments that describe the functions, data, type definitions, and macros from a usage perspective should be included.
    - Duplicate inclusions can be avoided by enclosing the header file's code within a #ifndef that tests for a macro defined by the header file.

```
#ifndef MYHEADER
#define MYHEADER
...
#endif
```

  - The source module is where all the function bodies are placed.
    - To guarantee that the header file matches the source file, the source file must include its own header.
    - The real data copy of any data that was declared external in the header file should be placed in the source file.
    - The size of the module should be kept under 64K.

  - A revision history should be found as a comment at the top of the source module.
  - Only one source module can contain the main() per application.
  - Each module is compiled only; linkage is done at a later time.
- Modules should be organized by subject type.

Building a reusable library of functions.
- .OBJ modules can be clumsy to deal with.
- A better solution is to combine related .OBJ modules within a .LIB file.
- .LIB files are easier and more efficient to link with.
- The external utility TLIB is used to make a .LIB file, and to add .OBJ modules to it.

General rules about organized development.
- Properly modularized applications are the open door to infrastructure.
- Modules that are usable by multiple applications should be kept in a centralized library base.
- The code that is application specific should be kept in a centralized application repository.
- All code should be version controlled, and backed up.
- Many companies have on-line help that explains the code libraries and applications.
- Library groups are focus teams dedicated to the development and maintenance of the centralized library base.
- Application development groups are teams that are focused on the business problem.
- The benefits of organized development: include faster development, better quality, lower testing costs, cleaner maintenance, ect.
- There is no fixed time frame for organizing development teams; they are an ongoing effort.
- Libraries can't be developed, they are adopted.

Using non-standard libraries and object modules.
- To implement any library is simple.
  - Include the required header file in the application.
  - Use the functions.
  - Specify the .LIB on the linkage line, or in the project.

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Quick Test.

1. How should data be declared in a header file?
2. How do multiple modules affect compilation?
3. What should be included in a header file?
4. What are some benefits of an organized development team?
5. How can duplicate inclusions be avoided?
6. How many modules can contain a main() per application?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Make Files

A general overview of dependencies and dependence processors.

- Command line builds are the only production worthy processes.
- Building from the command line presents some new challenges.
- Modules within an application are dependent on each other.
- The order in which application modules are compiled and linked is very important.
- When a component changes, any modules that compile with that component must re-compile, and any modules that link with that component must re-link.
- If an application is linked before one or more of its modules are compiled, the new code will not be included in the .EXE.
- Keeping track of the dependencies by hand is impossible.
- Batch files are sometimes used to guarantee the proper dependencies order, but they cannot compile and link only what needs to be.
- A dependence processor called MAKE.EXE is used to properly compile and link an application.
- MAKE is date driven, and it compiles or links based upon each component's date and time stamp.
- MAKE uses a data file called a make file as instructions for building the system.
- Make files have an extension of .MAK.
- A make file is a listing of targets to build.
- Targets are normally equated to .EXEs, .LIBs, and .OBJs, but can also be fictitious.
- All make files have one fictitious target call ALL, and the application executable is its only dependent.
- ALL is the starting point for the make file.
- Each target has a list of dependencies that it is dependent on, and a command block associated with it.
- Dependencies can be other targets or files.
- The lists of targets and dependencies form a tree that represents all the elements of the application.
- MAKE recurses the list of dependencies and searches from the leafs to the trunk for any dependents that are newer than its target.
- When a dependent is newer than its target, the target is out of date, and the command statements assigned to the target are executed.
- The command statements normally are a compile or link statements, but can be any executable command or DOS call.
- Make files can be generated by the environment from a project, or they can be written by hand.

Examples.

- See example disk for program MAKESEXM.MAK.

Quick Test.

1. Can a file be a target?
2. Can any DOS command be executed from MAKE.EXE?
3. Why is compilation and linkage order important?
4. What is a dependency?
5. Where is the starting point for a make file?
6. Can a target be a dependency?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Binary Graphics

The background on binary graphics.

- Although DOS is text based, a PC can be run in graphics mode.
- MS-Windows is graphics based.
- Graphics mode allows the display of graphic images.
- Graphics mode breaks the screen into pixels, which are the smallest video unit.
- The number of pixels that fill a screen depend on the hardware and video mode.
- Graphics mode is very computer specific.
- Common video systems include: Herc, CGA, MCGA, EGA, VGA, and SVGA.
- VGA is now the "standard" for displays.
- Native graphics programming is very complex; consequently, an interface library is normally used.
- A common interface library is Borland's Binary Graphics Interface (BGI).
- BGI provides similar capabilities as console I/O, but extends capabilities to include basic shapes.
- BGI requires that .BGI drivers be available to the running application.
- The .BGI drivers are found in the BGI directory of the Borland product.

Manipulating video with binary graphics functions.

- void far arc(int x, int y, int stangle, int endangle, int radius);
  - Arc() is used to draw an arc on the screen.
  - The starting location is specified by x and y.
  - The starting angle is passed as stangle, and the ending angle is passed as endangle.
  - The arc's radius is passed as radius.
- void far bar(int left, int top, int right, int bottom);
  - Bar() is used to draw a bar on the screen.
  - The bar's starting and ending points are passed as left, top, right, and bottom.
- void far bar3d(int left, int top, int right, int bottom, int depth, int topflag);
  - Bar3d() is used to draw a 3D bar on the screen.
  - The bar's starting and ending points are passed as left, top, right, and bottom.
  - The depth of the 3D effect is passed as depth.
  - Topflag controls whether or not the top of the bar is drawn.
- void far circle(int x, int y, int radius);
  - Circle() is used to draw a circle on the screen.
  - The starting point is passed as x and y.
  - The radius is passed as radius.
- void far cleardevice(void);
  - Cleardevice() is used to clear a graphics screen.
- void far closegraph(void);
  - Closegraph is used to return to text mode.

- void far detectgraph(int far *graphdriver, int far *graphmode);
  - Detectgraph() is used to automatically detect what video system is on the computer.
  - The appropriate graph driver number is returned in graphdriver.

```
DETECT    0    (requests autodetection)
CGA       1
MCGA      2
EGA       3
EGA64     4
EGAMONO   5
IBM8514   6
HERCMONO  7
ATT400    8
VGA       9
PC3270    10
```

  - The appropriate mode is returned in graphmode.

```
Mode      #    Screen    Palette
VGALO     0    640 x 200 16 color
VGAMED    1    640 x 350 16 color
VGAHI     2    640 x 480 16 color
```

- void far getimage(int left, int top, int right, int bottom, void far *bitmap);
  - Getimage() is used to copy a section of the screen into a buffer.
  - The block's starting and ending points are passed as left, top, right, and bottom.
  - The address of the memory to be used to hold the image is passed as bitmap.
- int far getmaxx(void);
  - Getmaxx() returns the maximum width of the current mode.
- int far getmaxy(void);
  - Getmaxy() returns the maximum height of the current mode.
- unsigned far getpixel(int x, int y);
  - Getpixel() returns the color of the pixel at x, y.
- unsigned far imagesize(int left, int top, int right, int bottom);
  - Imagesize() returns the total size of an image for the current mode.
  - The block's starting and ending points are passed as left, top, right, and bottom.
- void far initgraph(int far *graphdriver, int far *graphmode, char far *pathtodriver);
  - Initgraph() is used to set the system into graphics mode.
  - The graph driver and mode are passed as graphdriver and graphmode.
  - The path to the .BGI file is passed as pathtodriver.
- void far line(int x1, int y1, int x2, int y2);
  - Line() is used to draw a line on the screen.
  - The starting and ending positions are passed as x1, y1 and x2, y2.
- void far lineto(int x, int y);
  - Lineto() is used to draw a line from the current position to position x, y.
- void far outtextxy(int x, int y, char far *textstring);
  - Outtextxy() is used to write to the screen.
  - The starting location is passed as x, y.
  - The text to write is passed as text.

- void far pieslice(int x, int y, int stangle, int endangle, int radius);
  - Pieslice() is use to draw a pie slice on the screen.
  - Pieslice() works like arc(), but draws lines from the arc to x, y.
- void far putimage(int left, int top, void far *bitmap, int op);
  - Putimage() copies the image passed as bitmap to the left, top position on the screen.
  - The method used to copy the image to the screen is passed as op.

```
COPY_PUT  0    Copy
XOR_PUT   1    Exclusive or
OR_PUT    2    Inclusive or
AND_PUT   3    And
NOT_PUT   4    Copy the inverse of the source
```

- void far putpixel(int x, int y, int color);
  - Putpixel() is used to set the color of pixel x, y.
- void far rectangle(int left, int top, int right, int bottom);
  - Rectangle() is used to draw a rectangle on the screen.
  - The rectangle's starting and ending points are passed as left, top, right, and bottom.
- void far setbkcolor(int color);
  - Setbkcolor() is used to set the current background color.
  - The color to be set is passed as color.
- void far setcolor(int color);
  - Setcolor() is used to set the current color.
  - The color to be set is passed as color.
- int far textheight(char far *textstring);
  - Textheight() returns the height in pixels of textstring using the current font.
- int far textwidth(char far *textstring);
  - Textwidth() returns the width in pixels of textstring using the current font.

What needs to be done to use binary graphics functions.

- To use any graphic functions, just include GRAPHICS.H in the program by placing an #include <graphics.h> at the top of the file, then link with the graphics library.

Examples.

- See example disk for program GRAPHEXM.C.

Quick Test.

1. What is the smallest unit used in graphics functions?
2. What file needs to be available for a graphics program to run?
3. What functions could be used to build a bar graph?
4. What is the difference between graphic and text mode?
5. What function is used to get the size of an image?
6. What functions are used to change between text and graphic mode?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

Port Level Processing

The background on port level processing.

- Interfacing with peripheral devices can be done at the hardware port level.
- There is no overhead at the port level, so this is the highest speed processing available.
- Mostly all peripherals are specialized processors and have a unique code interface.
- Each peripheral is handle in a different way. Even between models of the same manufacture there are major differences.
- Because of the differences in devices, it is very difficult to obtain any portability or reliability.
- Since there is no overhead, no safety checks are performed to stop the application from ruining the computers integrity.
- Information about the hardware ports is available in PC technical reference manuals, but information about each peripheral must be gotten from the manufacture of the peripheral.

Manipulating peripheral equipment with port level functions.

- unsigned char inportb(int portid);
  - Inportb() is used to retrieve the byte value in the port specified by portid.
- int inport(int portid);
  - Inport() is used to retrieve the integer value in the port specified by portid.
- void outportb(int portid, unsigned char value);
  - Ouportb() is used to place the byte passed as value into the port specified by portid.
- void outport(int portid, int value);
  - Ouport() is used to place the integer passed as value into the port specified by portid.

What needs to be done to use port level I/O.

- To use any graphic functions, just include DOS.H in the program by placing an #include <dos.h> at the top of the file, then link with the standard library.

Examples.

- See example disk for program GRAPHEXM.C.

Quick Test.

1. What is the effect of port level I/O on safety?
2. Which function is used to get an integer port value?
3. What is the effect of port level I/O on speed?
4. Which function is used to set a byte value in a port?
5. How does port level I/O affect reliability?
6. Where can information about ports be found?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

A First Look at C++ and Objects

The basic differences between C and C++.

- C++ is a super set of C.
- C++ programs have an extension of .CPP not .C.
- The compiler implements tighter type checking in C++.
- C++ provides new implementations that include: classes, overloading, exception handing, late binding, defaulted arguments, access control, and inheritance.
- C++ adds the capability of implementing object oriented programming (OOP).

The concept of classes and objects.

- Classes are much like a C structure.
- In C++ both classes and structures have functions called methods that are used to manipulate the data of the class or structure.
- Just like a structure, a class is only a definition and does not occupy any storage space.
- An instance of ether a structure or a class occupies storage space, and is referred to as an object.
- An object is an instance of a class or a structure.
- Since both classes and structures have methods, often called member functions, that manipulate the data members, there is no need for external functions to access the data directly.
- External code uses the object by calling the member functions.
- Objects are tightly encapsulated and should appear as a black box to the external code.
- Because of the tight encapsulation, objects are very modular and can be tested individually.
- Objects are often tested with exercisers that test all the object's behaviors.
- The modularity allows for easy replacement and shortened testing cycles.
- Just like things in the real world, objects are only developed to solve a problem.
- An object can not be designed until the goal of the object is clearly defined.
- A typical OO application design starts by defining the business problem.
- Once the business problems are defined, the functional areas of the problem are declared.
- With the function areas declared, the responsibilities of each area are identified.
- For each responsibility, a statement of how the responsibility will be upheld is provided.
- The answer of how the responsibility will be upheld may be the implementation of an object or program code.
- If this process is not followed, the object is not properly defined and code will suffer.

- This process forces the proper definition of the business problem, and design of its solution.
- The process is sometimes refered to as what-who-what-how.

Examples.

- See example disk for program LISTSEXM.CPP.

Quick Test.

1. How does type checking in C compare to C++?
2. What is the difference between a C structure and a C++ structure?
3. How does OOP affect testing?
4. What is the difference between an object and a class?
5. Before an object can be built what must be done?
6. What is a method used for?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

MS-Windows Programming Using C

The process of developing a MS-Windows program.

- Windows is an event oriented operating system that runs in graphic mode, and has in-built interface procedures.
- Windows applications are formed into a new executable format.
- All the program code that builds the application dialog boxes is part of Windows.
- An application provides Windows with a description of what the dialogs are to look like, and Windows handles building the dialog.
- A Windows application can be logically divided into three main bodies of code: resource script, business solutions, and Windows overhead.
- The code that is dedicated to describing the interface components to Windows is called resource script.
- Resource script is a language on its own, and is stored in a .RC file.
- Although resource script can be hand coded, it is normally painted by using a resource editor.
- Like a .C module a .RC module must be compiled, but the output of a resource compile is a .RES file and a resource compiler is used.
- After a Windows application is compiled and linked, it must be bound with the .RES file.
- The business solutions are modules written to solve the business problem.
- The Windows overhead is code that deals with the Windows API, and serves no other purpose then to submit requests and manage events.
- Although the Windows overhead code can be hand coded, it is normally built by using a code generator.
- The functions that form the Windows' API are included as part of a software development kit that is available from Microsoft.
- The SDK is a package of header files and libraries that are used to interface with Windows.

The fallout of SDK programming.

- SDK programming is a very manual way of interfacing with Windows.
- Most C programmers abandoned Windows because of the complexity of the SDK.
- Borland saved Windows programming by encapsulating the SDK in an object layer called OWL.
- Microsoft refused to release an OO compiler.
- Because of the popularity of Borland products, Borland sales outnumber Microsoft by five to one.
- Microsoft responded to Borland's success by releasing a C++ compiler and the Microsoft Foundation Class library known as the MFC.
- Today the SDK is rarely used.

Examples.

- See example disk for program HELOWEXM.C.

Quick Test.

1. What do .RC files contain?
2. How did OO play a role in Windows programming?
3. What are the three functional code areas in a Windows program?
4. How are resource scripts normally built?
5. What is the SDK?
6. What was the problem with SDK programming?

www.sccarbone.com; sccarbone@sccarbone.com; (610) 888-1650

The Future of C

C vs. C++.
- C is a very important prerequisite to developing great PC packages.
- Today C is being replaced by C++.
- Most software manufactures are currently producing C++ environments.
- C++ is a better C, and all that C can do C++ can do as well or better.
- Many C applications will directly compile under C++, and others will require only minor changes.

C/C++ vs. the "wonder tools".
- In experienced hands C/C++ applications development is very fast, but experience takes time to gain.
- Many companies don't want to invest the time in training and infrastructure that is required for application development with any tool.
- 4GL languages have become very popular in today's environment because they do not require the extensive training.
- "Wonder tools" offer rapid application development, but often sacrifice performance and capability.
- The primary objective of software engineering is to solve the business problem.

Making the "wonder tools" really work.
- Mostly all 4GL products are written in C/C++, and many have the ability to call high speed C/C++ functions.
- C/C++ can co-exist with today's 4GL products, and can supply the efficiently to solve performance intensive problems.
- The combination of C/C++ and a 4GL product is normally the best solution.

Quick Test.
1. How are software manufactures affecting C?
2. What is sacrificed when implementing 4GL products?
3. Why do some companies not pursue C/C++ programming?
4. How can C/C++ enhance 4GL products?
5. Why are 4GL products popular?
6. What is the primary objective of software engineering?

