Advanced Programming Concepts

You will learn how advanced programming constructs and concepts are implemented in 'C'

Variable Types

- Simple (atomic)
  - Can't be subdivided further into meaningful sub-elements
  - Examples: integers, floating point/real, characters etc.
- Composite
  - The variable can be treated as a single entity (one whole) e.g., one lecture section of a course can be passed to a function.
  - At the same time that composite entity consists of individual elements e.g., information about each student in a lecture can be individually extracted and changed.
  - Typically composite types can take two forms:
    - Homogeneous: each part of the composite type must store the same type of information e.g., a list of characters that store the letter grade for a student.
    - Heterogeneous: while each part of the composite type can store the same type of information there is no enforced requirement that it does so e.g., a student record consists of a number of fields and each field can store different kinds of information.

Homogeneous Composite Types

•In many languages it's implemented as an array.

•(Note: each element MUST store the same type of information…which is why it's homogeneous).

Format (creating an array variable):

<type of each element> <name of the array> [*<no of elements>*];

Example (creating an array variable):

```
int lecture1_grades [100];
int leture2_grades [100];
```

- **Creates a list of 100 elements.**
- **Each element consists of an integer.**
- **Elements are numbered from 0 – 99.**

Homogeneous Composite Types (2)

•Assigning a value to a single array element

- lecture1_grades [0] = 100;

•To copy the contents of one array to another a loop is needed.

```
for (i = 0; i < 100; i++)
{
      lecture1_grades [i] = lecture2_grades [i];
}
```

Homogenous Types (3)

- Arrays can consist of multiple dimensions.
- (So far you've just seen arrays of a single dimension which are lists).
- The number of characteristics used to describe the data being stored will determine the number of dimensions used for the array.

Homogenous Types (4)

- Example: the grades for a single lecture could be stored in a 1D array while the grades for multiple lecture sections could be stored in a 2D array.

Student →

Lecture section ↓

	First student	Second student	Third student	…
L01				
L02				
L03				
L04				
:				
L0N				

Homogenous Composite Types (5)

- **Format** (declaring an array with multiple dimensions):
  <type of each element> <name of the array> [<*size of dimension* $_1$>] [<*size of dimension*$_2$>]...[<*size of dimension*$_n$>];

- **Example** (declaring an array with multiple dimensions):

```
int grades [100][10];
```

100 Rows (0 – 99) **100 Columns (0 – 99)**

James Tam

Accessing Elements In Arrays With Multiple Dimensions

- Typically it's done using multiple nested loops.
- **Example**:

```
/* Initializing each array element to a starting value */
  main ()
  {
    int r;
    int c;
    int matrix [10] [10];
    for (r = 0; r < 10; r++)
      for (c = 0; c < 10; c++)
        matrix [r][c] = 0;
  }
```

James Tam

1D Character Arrays

- A character array (sometimes referred to as a ‘string’) is a special case because the last occupied element of the array should be marked with the null character (ASCII value 0) to reduce the risk of accessing memory beyond the bounds of the array.
- Example:

```
char list1 [6] = “abcde”;
char list2 [6] = “abc”;
```

- The format specifier for displaying a string onscreen is “%s”

1D Character Arrays (2)

- Some common and useful string functions (in the library string.h).
- Be sure to include in the header of your program a reference to the library.

```
#include <string.h>
```

- String length
  - Counts the number of occupied array elements
  - **Format:**

    <*size*> strlen (<*string*>);

  - **Example:**

    ```
    size = strlen (“hello”);
    ```

1D Character Arrays (3)

- String copy
  - Copies one string into another string.
  - **Format:**

```
strcpy (<destination string>, <source string>);
```

  - **Example:**

```
char str1 [8];;
strcpy (str1, “hello”);
```

  - Note: it is the responsibility of the programmer using this function to ensure that the destination string is long enough to contain the source string.

1D Character Arrays (4)

- String concatenation
  - Concatenates (“glues” the second string onto the end of the first string string)
  - **Format:**

```
strcat (<string 1>, <string 2>);
```

  - **Example:**

```
char str1 [16]  = "hello";
char str2 [8]  = "there";
strcat (str1, str2);
```

  - Note: it is the responsibility of the programmer using this function to ensure that the destination string (first string) is long enough to contain the contents of the two strings.

1D Character Arrays (5)

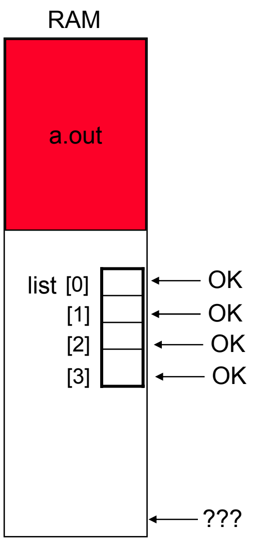
- String compare
  - Compares two strings.
    - Returns a negative value if string1 is less than string 2.
    - Returns zero if string 1 and string 2 are equal.
    - Returns a positive value if string 1 is greater than string 2.
  - **Format:**

    <integer> strcmp (*<string 1>*, *<string 2>*);

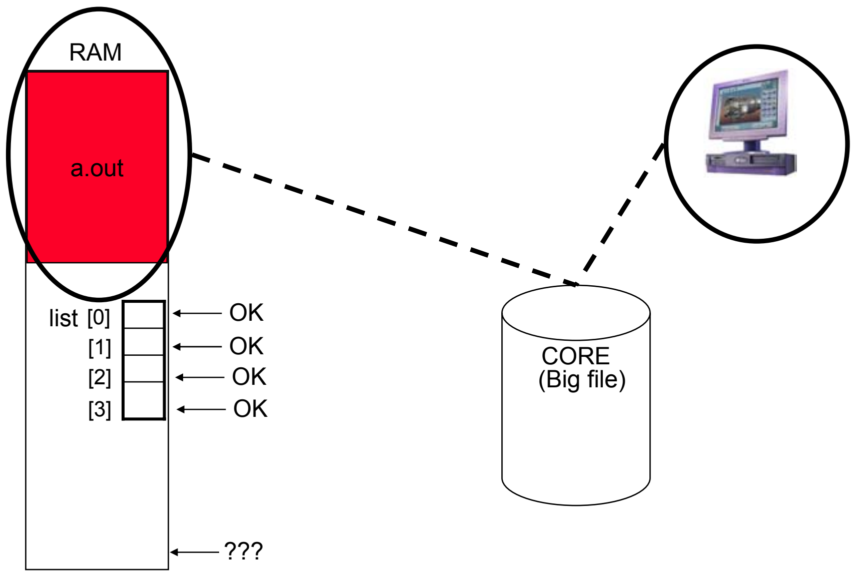
  - **Example:**

```
char pass [80];
printf ("Enter password: ");
scanf ("%s", &pass);
if (strcmp (pass, "password") == 0)
   printf ("Bad password so you got it");
```

Segmentation Faults And The Array Bounds (2)



Segmentation Faults And The Array Bounds (2)



Wav file from "The SImpsons"

Homogenous Composite Types: Good Style

- Because the size of the array is fixed after it's been declared it's regarded as good style to use a constant in conjunction with an array declaration:
- **Example**:

```
main ()
{
   int const SIZE = 10;
   int list [SIZE];
   for (int i = 0; i < SIZE; i++)
       list [i] = 0;
}
```

Heterogeneous Types

- In C the composite type (where each ‘portion’ of the whole doesn’t have to store the same type of information as the other portions) is implemented as a ‘struct’.
  - Other languages use other constructs e.g., Pascal uses the ‘record’, Java and Python uses ‘classes’ etc.
- In this case you are defining a new type of variable that can be created and used so you first need to define to the compiler the number and the type of fields for the struct.
- **Format** (defining the struct):

```
typedef struct
{
    <type of field1> <name of field1>;
    <type of field2> <name of field2>;
          :                :
   <type of fieldn> <name of fieldn>;
} <name of the struct>;
```

Heterogeneous Types (2)

- **Example** (defining a struct):

```
typedef struct
{
     int telephone;
     int id;
     char name [128];
} Student;
```

Example (declaring a variable):

```
Student bart;
Start lisa;
```

Heterogeneous Types (3)

- **Example** (assignment, entire struct):

```
lisa = bart;
```

- **Example** (assignment, single field):

```
lisa.telephone = bart.telelphone;
```

Displaying Information About A Struct

- Recall: A struct is a new type of variable that needs to be defined.
- Output functions such as "printf" won't be able to display the contents of new variable types.
- Consequently the only types of information that can be displayed are predefined types (it often means that the different fields of the struct must be displayed in succession).
- **Examples:**

```
printf ("%d", lisa.telephone);
printf ("%s", lisa.name);
```

Question: What will happen with the following statement? Why?

```
printf ("%d", telephone);
```

Binary Vs. Decimal Numbers

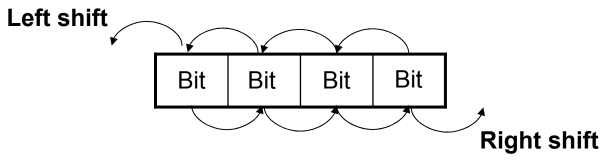
Decimal value	Binary value	Decimal value	Binary value
0	0000	8	1000
1	0001	9	1001
2	0010	10	1010
3	0011	11	1011
4	0100	12	1100
5	0101	13	1101
6	0110	14	1110
7	0111	15	1111

Why Is C Regarded As A Powerful And Low-Level Language

- Operations right down to the level of the bits can be performed.
  - That means that the individual bits of a non-composite type such as character can be modified and extracted.
- It's easy to invoke functions that are built into the operating system.

Bit Level Operations

•Bit shifting



- **Format:**
  Variable name = Value to shift << Number of shifts
- **Example:**
  - unsigned int num = 2;
  - num = num << 2;
  - num = num >> 2;

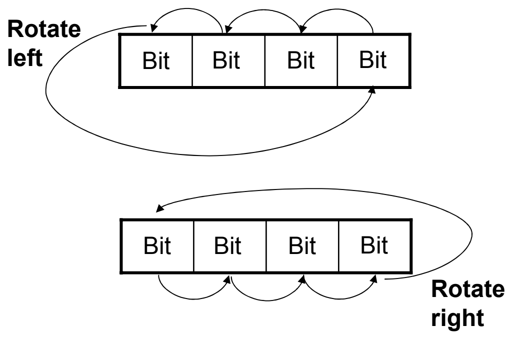
Bit Level Operations (2)

•Bitwise operations

- Bitwise AND ‘|’
  Variable name = Value 1 & Value 2
- Bitwise OR ‘|’
  Variable name = Value 1 | Value 2
- Bitwise NEGATION ‘~’
  Variable name = ~Value

Bit Level Operations (3)

•Bit rotations:



•Although it's not implemented in C with an existing operator you can write the program code for this new operation using the existing ones.

There Are Benefits To Using Bitwise Operations

•In some applications speed is an issue important
- Scenarios where a guaranteed response time is mandatory
- E.g., the software used to fly an airplane

•Large and complex programs
- E.g., Complex simulations (Biology, Economics)
- E.g., games that draw complex graphics

System Calls

- C programs have the ability to call the commands of the operating system and utility programs installed with the operating system.
- This provides a useful set of utilities for the programmer.
  - E.g., If the programmer wants spell checking functions within their own program rather than writing it, a system call may be made to the spell checker that's included with the operating system.

- **Format**:

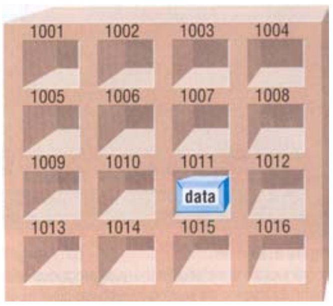
```
system ("<system command to execute>");
```

- **Example**:

```
system ("dict superflous");
```

Variables

- Variables are stored in memory (RAM).
- Each variable is stored in a 'slot' and with each one numbered sequentially.



- Typically computers programs access a slot via the variable name but slots may also be accessed via the numerical address (depending upon the programming language).

Picture from Computers in your future by Pfaffenberger B

Variables (2)

- The variable types covered so far store information (atomic, homogenous, heterogeneous).
- Some variables don't store information but instead store an address in memory. That address can contain information.
  - Because these types of variables refer to (or "point" to) an address they're referred to as pointers.
  - Pointers are used in several contexts while writing programs and how they can be used WILL BE REFERRED TO LATER when functions and parameter passing is discussed.

Visualizing data variables

Data variable (e.g., integer)
Data (e.g., 10)

Visualizing pointer variables

Data variable @ address 1000 in RAM
10
Pointer variable (stores an address)
Address 1000

Pointers

- **Format** (declaration):
  <Type of memory referred to> * *<Pointer name>*;

- **Example** (declaration):
  int *num_ptr;

Pointers (2)

- Important distinction: because there are two memory locations involved with pointer variables you need to understand when:
  - you are working with the pointer,
  - or you are working the memory referred to by the pointer.
- Recall: stating the name of a variable will refer to the memory slot associated with that variable name.

```
int num1 = 1;
int num2 = 2;
printf (“%d”, num1);
```

- With a pointer:
  - Referring to the name of the pointer will yield an address (because a pointer stores an address not data) e.g., num_ptr
  - Preceding the name of a pointer with a star will reference the address contained in the pointer (“de-reference” the pointer to access what the pointer “points to”) e.g., *num_ptr

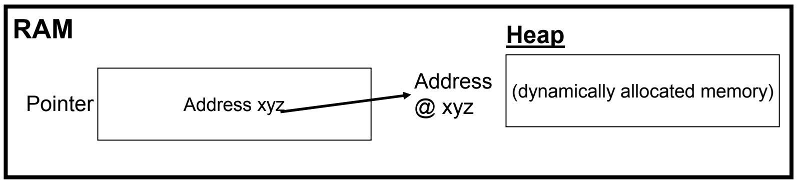
Pointers: An Example

The complete example (“pointer1.c”) can be found in the UNIX directory: /home/courses/219/examples/c_advanced.

```
main ()
{
  int *num_ptr;
  int num = 2;
  printf ("num: %d\n", num);
  num_ptr = &num;
  printf ("*num_ptr: %d\n", *num_ptr);
  printf ("num_ptr: %d\n", num_ptr);
  printf ("&num: %d\n", &num);
  *num_ptr = 888;
  printf ("num: %d\n", num);
  printf ("*num_ptr: %d\n", *num_ptr);
}
```

A Way In Which Pointers May Be Used

- Pointers can be used to dynamically allocate memory (create a memory space to be used as a variable).
- Functions such as "malloc"[1] be used to dynamically allocate memory as needed and the address of this memory can be stored and accessed through a pointer.
  - It's "dynamically" because it's allocated when malloc is called.
  - This is similar to how local variables come into scope as functions are called (but the memory used for local variables are stored on "the stack" whereas memory used for functions such as malloc are stored in the "heap").



1) Malloc can be found in the library "stdlib.h"

Using Pointers: An Example

- The complete example ("pointer2.c") can be found in the UNIX directory: /home/courses/219/examples/c_advanced

```
main ()
{
  char *list_ptr;
  int list_length;
  int i = 0;

  printf ("Enter the length of the string: ");
  scanf ("%d", &list_length);
  list_ptr = malloc (list_length+1);
  getchar ();
```

Using Pointers: An Example (2)

```
   if (list_length > 0)
   {
     printf ("Enter the string (max length) %d: ", list_length);

     do
     {
       list_ptr[i] = getchar();
       i++;
     } while (i < list_length);

     i = 0;
     while (i < list_length)
     {
       printf ("%c", list_ptr[i]);
       i++;
     }
     printf ("\n");
   }
 }
```

Functions: What You Know

- Recall: functions must be defined before they can be used (called).
- Python: function definition:

  def <*function name*> ():

  body
- Python: function call:

  function name ()

Functions: Format

- **Format** (definition):

```
<function name> ()
{
    <Body>
}
```

- **Format** (call):
  - *<function name>* ();

- **Example**:

$$\left.\begin{array}{l}\texttt{fun ()}\\ \texttt{\{}\\ \quad\texttt{printf ("In fun");}\\ \texttt{\}}\end{array}\right\}\textbf{Function definition}$$

$$\begin{array}{l}\texttt{main ()}\\ \texttt{\{}\\ \quad\left.\texttt{fun ();}\right\}\textbf{Function call}\\ \texttt{\}}\end{array}$$

The 'Main' Function

- It's a special function in C (and in Java)
- It's the function that is automatically executed when the program is run.

Local Variables

- To minimize the amount of memory that is used to store the contents of variables and to reduce side effects (variables being modified by accident) only declare variables when they are needed.
- Local variables: variables declared within a set of braces are local to those braces:.

```
{
   int num;
```

After this declaration 'num' comes into scope (is accessible).

```
 }
```

After this enclosing brace 'num' goes out of scope (is no longer accessible).

Function Return Values

- Recall: the value of local variables aren't visible after a function ends.
- Return values can be used to return the current state of a variable back to the caller of a function.
- Note: a function can only return zero values or exactly one value (but not more than one value).
- **Format** (function definition):

```
<function return type>
<function name> ()
{
   return <value to be returned: a constant or variable>;
}
```

Function Return Values (2)

- The value that is returned from a function is usually assigned to a variable.
- Valid function return values include the types that you've been introduced to thus far (e.g., int, char etc) plus any new types that you may have defined (using a "typedef").
- A function that returns no value should return a "void" value.
- If a return type is not specified then the compiler will assume something of type "int" is to be returned.

Function Return Values (3)

- The complete program ("temperature.c") can be found in the UNIX directory: /home/courses/219/examples/c_advanced.
- **Example**:

```
void
introduction ()
{
  printf ("Celsius to Fahrenheit converter\n");
  printf ("------------------------------\n");
  printf ("This program will convert a given Celsius temperature to\n");
  printf ("an equivalent Fahrenheit value.\n");
}

int
convert (int celsius)
{
  int fahrenheit;
  fahrenheit = celsius * (9 / 5) + 32;
  return fahrenheit;
}
```

Function Return Values (4)

```
main ()
{
  int celsius;
  int fahrenheit;

  printf ("Type in the celsius temperature: ");
  scanf ("%d", &celsius);
  fahrenheit = convert (celsius);
  printf ("\nFahrenheit value: %d\n", fahrenheit);
}
```

Function Parameters

- Since local variables aren't visible outside of a function they may need to be passed in as parameters (inputs) into functions.
- **Format** (defining function):

<function return type>
<function name> (*<type of parameter$_1$> <name of parameter$_1$>*, *<type of parameter$_2$> <name of parameter$_2$>*,...*<type of parameter$_n$> <name of parameter$_n$>*)
{
    body
}

- **Format** (calling function):

<function name> (*<name of parameter$_1$>*, *<name of parameter$_2$>*,...*<t<name of parameter$_n$>*);

Function Parameters (2)

- The complete program ("temperature2.c") can be found in the UNIX directory: /home/courses/219/examples/c_advanced.
- **Example**:

```
void
introduction ()
{
  printf ("Celsius to Fahrenheit converter\n");
  printf ("-------------------------------\n");
  printf ("This program will convert a given Celsius temperature to\n");
  printf ("an equivalent Fahrenheit value.\n");
}

void
display (celsius, fahrenheit)
{
  printf ("\nConversions\n");
  printf ("Celsius value: %d\n", celsius);
  printf ("Fahrenheit value: %d\n", fahrenheit);
}
```

Function Parameters (3)

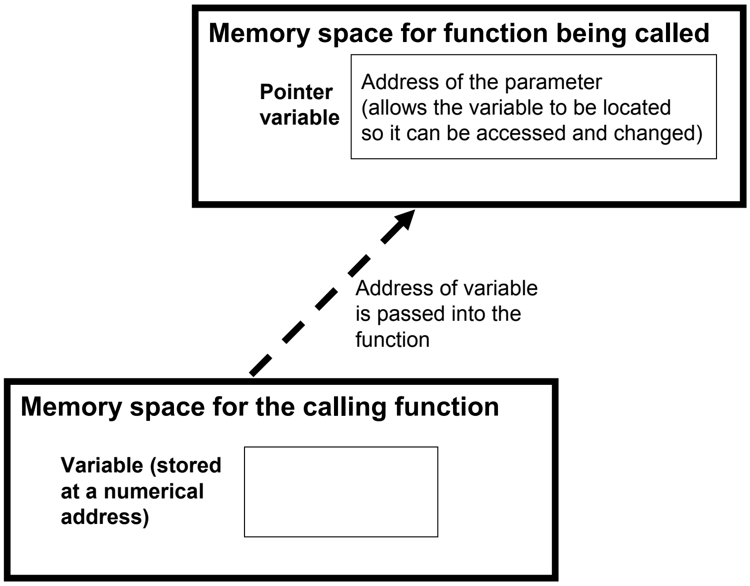
```
void
convert ()
{
   int celsius;
   int fahrenheit;
   printf ("Type in the celsius temperature: ");
   scanf ("%d", &celsius);
   fahrenheit = celsius * (9 / 5) + 32;
   display (celsius, fahrenheit);
}

main ()
{
   introduction ();
   convert ();
}
```

Functions: Approaches To Passing Parameters

1. Pass by value (*the value stored is passed*)
   - The *value* of the parameter is passed into a local variable.
   - It allows the *value* stored in the parameter to be accessed (but not changed, only the copy can be changed).
   - Unless otherwise specified parameters will be passed by value (default).

2. Pass by reference (*a reference to the original variable is passed*)
   - A *reference* to the original parameter is passed into function that not only allows access to value stored in the parameter but also the original parameter can be changed.

Passing By Reference


Memory space for function being called
Pointer variable
Address of the parameter (allows the variable to be located so it can be accessed and changed)
Address of variable is passed into the function
Memory space for the calling function
Variable (stored at a numerical address)

Pass By Reference (2)

- **Format** (function call):

  <function name> (&*<name of parameter$_1$>*, &*<name of parameter$_2$>*...
  &*<name of parameter$_n$>*);

- **Format** (function definition):
  - *<Function return value>*
  - *<Function name>* (**<type of parameter$_1$> <name of parameter$_1$>*, **<type of parameter$_2$> <name of parameter$_2$>*,...**<type of parameter$_n$> <name of parameter$_n$>*)

```
  {
      body
  }
```

& = "The address of" e.g., &num = address of variable num

* = "Pointer to location" *num = a pointer to a numerical variable.

Pass By Reference (3)

- **Example**:

```
void
fun (int *num_ptr, int num)
{
  *num_ptr = 10;
  num = 10;
  printf ("In fun: %d %d\n", *num_ptr, num);
}

main ()
{
  int num1 = 2;
  int num2 = 3;
  printf ("Pre-fun: %d %d\n", num1, num2);
  fun (&num1, num2);
  printf ("Post-fun: %d %d\n", num1, num2);
}
```

Passing Arrays As Parameters

- When an array is passed to a function as a parameter it actually isn't the array itself that is passed.
- Instead what's passed is a pointer to the first element of the array which allows the elements of the array to be accessed.
- **Format (definition)**:
  - (Method 1):

    <Function return value>
    <Function name> (*<Array element type>* *<Array name>* [*<Array size>*])

  - (Method 2):

    <Function return value>
    <Function name> (*<Array element type>* *<Array name>* [])

  - (Method 3):

    <Function return value>
    <Function name> (*<Pointer type>* * *<Pointer name>*)

Passing Arrays As Parameters (2)

- **Example**:
  - (Method 1):

```
void
fun (int list [10])
```

  - (Method 2):

```
void
fun (int list [])
```

  - (Method 3):

```
void
fun (int * list)
```

Scope

•Indicates where an identifier (variable, constant, function) may be accessed in a program.

•In 'C' the scope of a variable or constant is from where it has been declared to the end of the nearest enclosing brace.

```
fun ()
{
    char ch;   }
        :      } Scope
        :      } of 'ch'
}
```

```
if (num > 0)
{
    char ch;   }
        :      } Scope
        :      } of 'ch'
}
```

Scope (2)

•Because functions are typically defined outside a set of enclosing braces their scope begins after the header until the end of the program.

```
void
fun ()
{           }
    :       }
}           }
            } Scope of
main ()     } function 'fun'
{           }
    :       }
}  <-- End of program
```

Scope: Good Design

•To minimize side effects variables should be declared locally unless there is strong argument for doing otherwise (using a variable throughout the program isn't a sufficient cause for making it a global variable).

•Because constants cannot be changed they may be safely declared outside of enclosing brackets (where they have "global" scope like a function).

What You Need In Order To Read Information From A File

1. Declare a file variable
2. Open the file
3. A command to read the information
4. Close the file

You will also be referring to functions in the stdio library so a reference to the library via #include must be made at the top of the program

1. Declaring File Variables

- Allows the program access to a file (the mechanism that connects a computer program with a physical file).

Format:

```
FILE *<File variable>;
```

Example:

```
FILE *input_fp;
```

2. Opening Files

Format:

<File variable> = fopen (<*name of the file*[1]>, "<*opening mode*[2]>");

Example (opening a file):

```
input_fp = fopen ("input.txt", "r");
```

Example (checking for a non-existent file):

```
if (input_fp == NULL)
   printf ("File 'input.txt' cannot be opened.");
```

1) The location of the file can be specified here

2) Opening modes include: read ("r"), write ("w"), append ("a")

3. Reading From A File

- It's similar to reading from the console (user types in the input) so there's many similar functions that can be used.
- **Format** (one function):

```
fscanf (<file variable>, "%<format specifier>", <variable name>);
```

- **Example function call**:

```
fscanf (input_fp, "%s", word);
```

3. Reading From A File (2)

- However the function to read information from file is typically enclosed within a loop (read from the file until):
  1. The end of the file is reached.
  2. A special marker for the end of file is reached (the sentinel value).
- **Format**:

```
while (end of file has not been reached)
        Call function to read information from the file
```

- **Example**:

```
while (fscanf (input_fp, "%s", word) >= 0)
{
    printf ("%s ", word);
}
```

3. Reading From A File (3)

- There are different functions that can be invoked to perform the actual read from file:
  1. fscanf (<*file variable*>, "%<*format specifier*>", <*variable name*>);
     - It reads a 'word' at a time from the file (words are separated by spaces or end of the line).
     - Returns a negative value when the end of the file is reached.
  2. fgets (<*pointer or character array*>,<*number of char to read* >, <*file pointer*>);
     - Reads a line at a time from the file (will read in spaces).
     - Returns NULL when the end of file is reached.
     - Reads the number of characters specified minus one (it appends a NULL at the end of the string).
     - It stops reading when:
       - The end of file has been reached.
       - The specified number of characters (minus one) has been read.
       - The end of line has been reached.
       - A read error has occurred.

3. Reading From A File (4)

- Two complete example programs ("file1.c", "file2.c") illustrating how these functions work can be found in the UNIX directory: /home/courses/219/examples/c_advanced

4. Closing The File

•Closing a file is important for many reasons:
- While a file opened by a computer program is usually closed automatically when the program ends, abnormal termination of the program may leave the file open and "locked" (inaccessible).
- Some operating systems limit the number of files that can be opened.

•**Format**:
```
fclose (<file variable>);
```

•**Example**:
```
fclose (input_fp);
```

File Input And Output: Putting It All Together

•The complete program ("file3.c") can be found in UNIX in the directory: /home/courses/219/examples/c_advanced.

```
main ()
{
  FILE *input_fp;
  FILE *output_fp;

  int midterm_exam;
  int final_exam;
  int assignments;
  float term_gpa;
  char letter;
  char input_file_name [MAX];
```

File Input And Output: Putting It All Together (2)

```
  input_fp = fopen (input_file_name, "r");
  output_fp = fopen ("letters.txt", "w");

  if (input_fp == NULL)
    printf ("File '%s' cannot be opened.", input_file_name);
  else if (output_fp == NULL)
    printf ("Unable to open output file letters.txt for writing.");
```

File Input And Output: Putting It All Together (3)

```
else
{
 while (fscanf (input_fp, "%d", &midterm_exam) >= 0)
 {
   fscanf (input_fp, "%d", &final_exam);
   fscanf (input_fp, "%d", &assignments);
   term_gpa = (midterm_exam * 0.3) + (final_exam * 0.4) + (assignments *
                  0.3);

   if ((term_gpa <= 4) && (term_gpa > 3.5))
      letter = 'A';
   else if ((term_gpa <= 3.5) && (term_gpa > 2.5))
      letter = 'B';
   else if ((term_gpa <= 2.5) && (term_gpa > 1.5))
      letter = 'C';
   else if ((term_gpa <= 1.5) && (term_gpa > 0.5))
      letter = 'D';
   else if ((term_gpa <= 0.5) && (term_gpa >= 0))
      letter = 'F';
   else
      printf ("Error in gpa");
   fprintf (output_fp, "%c\n", letter);
} /* While: read from file */
```

File Input And Output: Putting It All Together (4)

```
    fclose (input_fp);
    fclose (output_fp);
  } /* Else: if program can read from the file. */
} /* End of the program. */
```

Documenting Functions

- Some of the things that are commonly included when documenting functions:
  - Name of the function
  - Purpose of the function: functions are supposed to implement one well defined operation which can be specified in the documentation e.g., “Function intro function provides instructions about how to use program.”
- Also some of the things that are used to document an entire program can also be used to document a function (except that the documentation will describe the one function rather than the entire program):
  - What are the limitations of the function
  - What assumptions are made about the state of the program in order for the function to operate without errors (preconditions) e.g., “The parameter age cannot be a negative value.”
  - The algorithm used to implement the function e.g., “Function sort uses a bubble sort to arrange the email contacts in alphabetical order.”

After This Section You Should Now Know

- What is the difference between a simple type and a composite type
- Homogeneous composite types (arrays)
  - How to declare a variable that is an array
  - The difference between accessing the entire array and the different parts of the array (e.g., row, element etc.)
  - Passing arrays as function parameters
  - What are strings and how do common string functions work
- What is a segmentation fault and under what circumstances do they occur. How good programming style can reduce the occurrence of segmentation faults.
- Heterogeneous types (struct)
  - How to define a new type of variable via typedef
  - How do declare instances of new types
  - How to access the entire composite type vs. accessing individual fields

After This Section You Should Now Know (2)

- How do low level (bit level) operations work and when it may be useful to write programs that employ them
- What is a system call and how is it done within a C program
- Functions
  - How to define and call functions
  - How to return a value from a function
  - The difference between the two parameter passing mechanisms (value and reference) and how to write a function that employs each approach
  - How to document functions
- Scope
  - What is the difference between local vs. global scope
  - Why variables should have local scope

After This Section You Should Now Know (3)

- Files
  - How to open a file for reading vs. writing
  - How to write a program that will read from or write to a file
  - How to close a file and while is it important to explicitly close a file