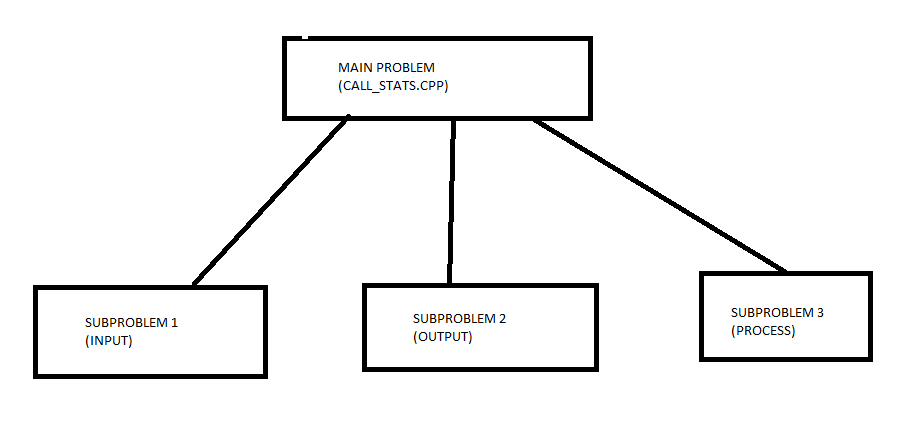
**4.1 TOP-DOWN DESIGN**

* **Divide-and Conquer Technique**
* **Break problem up into smaller subproblems; when each subproblem is solved main problem is solved**
* **example: main program call functions**
* **SEE THE FOLLOWING CHART**

**TOP-DOWN DESIGN**

****

* **SEE THE FOLLOWING SKELETON**

**void Input()**

**{**

**//stub – place holder for functionality to be implemented**

**}**

**void Output()**

**{**

**//stub**

**}**

**void Process()**

**{**

**// stub**

**}**

**int main()**

**{**

**//driver – used to test the functionality of your program**

**Input();**

**Process();**

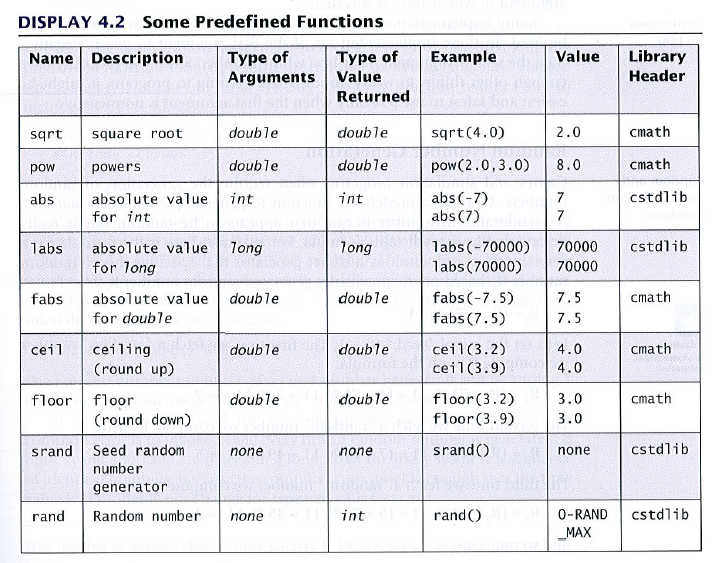
**Output();**

**return 0;**

**}**

**4.2 Predefined Functions**

* **Functions that are already implemented in C++ liabraries**
* **All you need to know to be able to use are the arguments needed for a call and what will be returned or done by the function(s)**
* **examples: sqrt, pow, abs, tTabs, fabs, ceil, floor, srand, rand and so forth**
* **see figure below from pg 187 (DISPLAY 4.2).**
* **some libraries: cstdlib, cmath, iostream, fstream, etc..**



* **Type casting – allows us to change the type of an argument in an expression**
* **syntax: static\_cast<double> (Expression\_of\_Type int)**
* **examples:**
  + **static\_cast<double>(9)**
  + **static\_case<double> x**

**4.3 PROGRAMMER-DEFINED FUNCTIONS**

* **function that are implemented by you**
* **formal parameter list**

**void Input(ifstream & in, call\_record & customer\_record)**

* **declaration (prototype)**

**void Input(ifstream &,call\_record &);**

**void Input(ifstream & in, call\_record & customer\_record);**

* **signature- function name + formal parameter list**

**void Input(ifstream & in, call\_record & customer\_record);**

* **header – resides on top of the function body**

void Input(ifstream & in, call\_record & customer\_record)

* **function call (invocation)**

**Input(in, my\_record);**

* **actual arguments**

**Input(in, my\_record);**

* **Boolean functions – functions that return a value of true or false**

**bool Is\_Array\_Empty(int Array[ ], int count)**

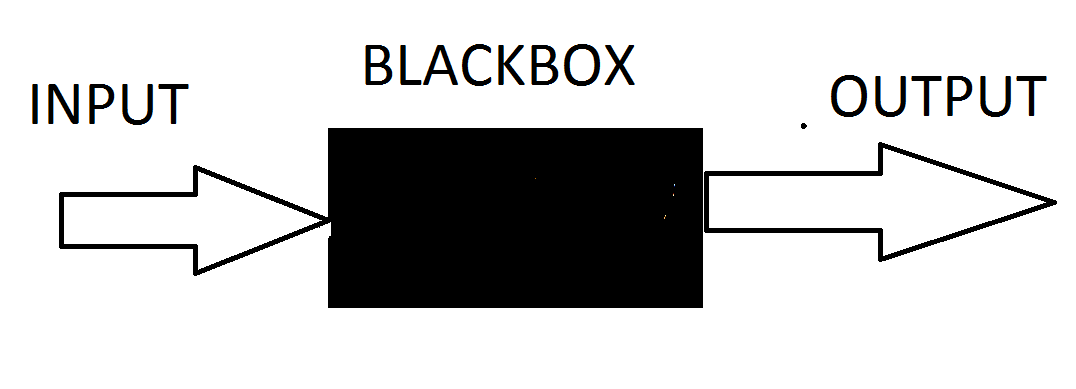
**{**

**return count == );**

**}**

**4.4 Procedure Abstraction**

* **Black-Box Analogy**

****

* **abstracting away from the details of the implementation**
* **focus on what you want to do, not how**
* **Steps in developing a program:**
  + **Define Problem (specification) eg. your assignment**
  + **Analysis the Problem**
  + **Design an Algorithm**
  + **Implement the Algorithm – coding**
  + **Test Program (locate and fixed bugs)**
  + **Retire Program**

**4.5 SCOPE OF A NAME (VARIABLES AND FUNCTIONS)**

**The scope is the region in a program where a name (identifier) has meaning. As you already know, all identifiers must be declared before they can be used**

* **{ } are use to define a scope**
* **give example of source file**
* **give example of formal parameter**
* **give example inside loop**
* **give example inside if-else**
* **give example inside function**
* **Global Constants and Global Variables (advantages and disadvantages)**

const double PI = 3.14 // what is the scope of PI

void Input(ifstream &,call\_record &); //what is the purpose of prototypes

void Output(const call\_record &);

void Process(const call\_record &);

void Input(ifstream & in, call\_record & customer\_record) // example using the call by reference mechanism in C++ -- call record is passed by reference --note & operator

{

//what is the scope of Input?

//what is the scope of the formal parametger and the local variables?

int x = 5, y;

if (x == 5)

{

int y = 15;

}

y = 20;

}

void Output(const call\_record & customer\_record)

{

//what is the scope of Output?

//what is the scope of the formal parameters?

}

void Process(call\_record & customer\_record)

{

//what is the scope of the formal parameter?

}

//Here is your driver to test the program

int main( )

{

//what is the scope of main?

// what is the scope of customer\_record?

call\_record customer\_record;

return 0;

}

**4.6. FUNCTION NAME OVERLOADING**

* **occurs when two or more function have the same name but different signatures.**
* **In C functions cannot have the same name**
* **examples:**

**void Add(int, int, int);**

**void Add(double, int, ch); //ok different signature**

**int Add(int, int, int); //no, same signature**

**MORE ON FUNCTIONS**

**Using functions in programs has several advantages which include: 1) allows for code reuse; 2) allows programs to be decomposed into smaller components (modularization); 3) they make debugging easier; 4) they make testing easier; 5) they allow for the implementation of recursion; 6) allow for generalizing a program; 7) allow information hiding.**

1. **Definitions**

**We will define several terms you need to know to really understand functions. They are as follows:**

1. **Formal parameters reside inside a function header. They allow the actual arguments to be passed to a function. The scope of formal parameters is local to the function where they are declared.**
2. **Actual arguments reside in the function that did the invocation (call). Their values are passed to the called function through the formal parameters.**
3. **The header of a function includes the function return type, the function name, and the formal parameter list enclosed in parenthesis.**
4. **The signature of a function includes the function name and the formal parameter list enclosed in parenthesis.**
5. **The function prototype includes the function return type, the function name, and the formal parameter list enclosed in parenthesis. The prototype ends with a semi-colon.**
6. **Function declarations (prototypes) provide information to the compiler to aid in the setup for a function call.**

**STRUCTURES (CLASSES WITH PUBLIC AND ONLY DATA MEMBERS)**

A **structure** is a heterogeneous data type. We say it is heterogeneous because it may be composed of members that all have different types. The members of a structure are accessed using the dot operator, and all members are public by default.

1. **Definitions**

We will define several terms that you need to know to really understand structures. They are as follows:

1. The **dot operator** is used to access the members of a structure. The symbol, ‘**.**’, is used to represent the dot operator.
2. The **total amount of memory** a structure uses is equal to the sum of all the memory (bytes) used by its members.
3. A structure is a **user defined data type**.
4. A structure is a **complex data type** because it may be composed of other structures and data types.
5. A structure is a heterogeneous data type because it may be composed of members that all have different types.
6. **Declaration Syntax**

**To declare a structure:**

struct structure\_name

{

field\_type\_1 field\_name\_1;

. . . .

field\_type\_n field\_name\_n;

};

**Notice that the member fields are enclosed in braces and that the right brace is followed by a semicolon.**

class structure\_name

{

public:

field\_type\_1 field\_name\_1;

. . . .

field\_type\_n field\_name\_n;

};

**Notice that the member fields are enclosed in braces and that the right brace is followed by a semicolon.**

POINTERS AND REFERENCES

A pointer holds the address of a memory location. In other words, a pointer contains a reference to another variable. A reference in C++ is a variable that references a memory address. References are often referred to as “aliases” of other variables.

1. **Definitions**

We will define several terms to help you understand pointers. They are as follows:

1. **Pointers** hold memory addresses.
2. When you **de-reference** a pointer, you retrieve the contents of the memory address that is stored in the pointer.
3. A **reference** is an alias for memory that is allocated elsewhere.
4. The “**\***” operator is used to declare and de-reference a pointer. Please pay close attention to the context in which the operator is used.
5. The “**&**” operator is called the “address of” operator.
6. The “**&**” operator is also used to declare references in C++.
7. **Declaration Syntax**

**Declaration for a pointer:**

type \* pointer\_name;

**Declaration for a reference:**

type & reference\_name;