H.O.#5
Fall 2015
Gary Chan

Functions and File I/O

Topics

- Introduction to functions
  - Function and memory
- Passing by value, references, and function pointers
- Passing arrays to functions
- Return by value and references
- Local and global variable scopes
- Recursion
- File I/O

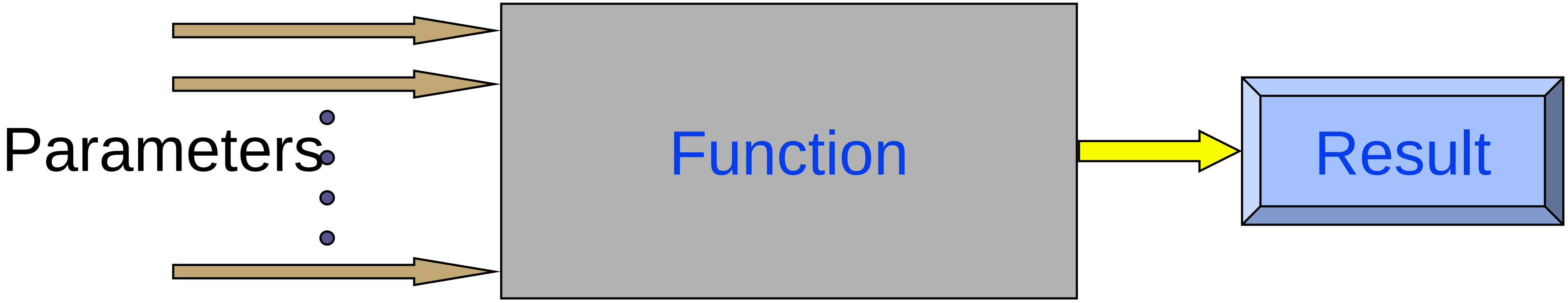
Introduction to Functions

- A complex problem is often easier to solve by dividing it into several smaller parts, each of which can be solved by itself.
- This is called top-down programming.
- These parts are called functions in C++ (also sometimes called subprograms).
- `main()` then executes these functions so that the original problem is solved.

Advantages of Functions

- Functions separate the concept (what is done) from the implementation (how it is done).
- Functions make programs easier to understand.
- Functions make programs easier to modify.
- Functions can be called several times in the same program, allowing the code to be reused.

Function Input and Output

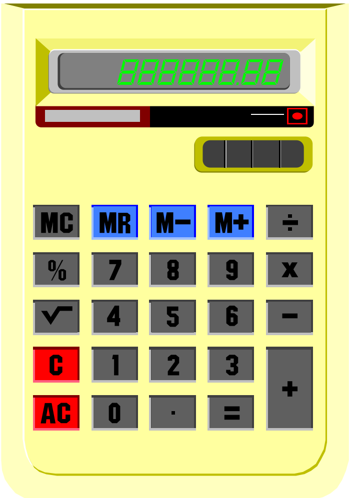


C++ Functions

- C++ allows the use of both internal (user-defined) and external functions.
- External functions (e.g., `cin`, `cout`, `rand`, etc.) are usually grouped into specialized libraries (e.g., `iostream`, `cstdlib`, `cmath`, etc.)

Mathematical Functions

- `#include <cmath>`

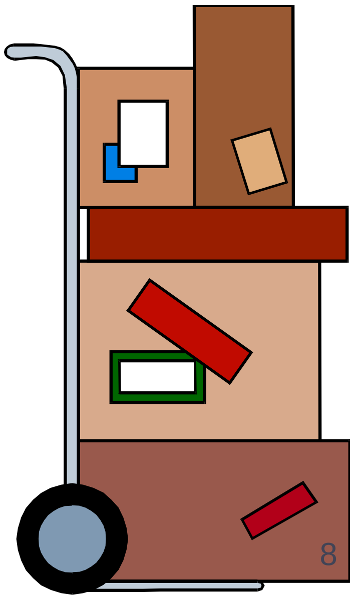


- double log(double x) natural logarithm
- double log10(double x) base 10 logarithm
- double exp(double x) e to the power x
- double pow(double x, double y) x to the power y
- double sqrt(double x) positive square root of x
- double ceil(double x) smallest integer not less than x
- double floor(double x) largest integer not greater than x
- double sin(double x), cos(double x), tan(double x), etc...

Functions in a Program

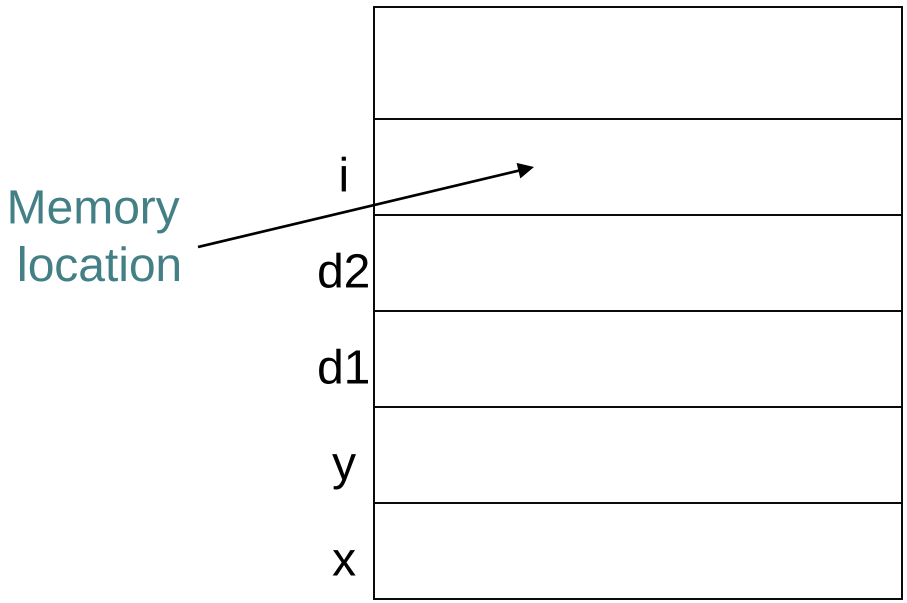
- Every time a function is called, a "stack" is created with a starting memory address. The return of the function removes the stack back to the caller
- `main` is the "mother" function
- C++ programs usually have the following form:

```
// include statements
// function prototypes
// main() function
// user-defined functions
```



Functions & Memory

- Every function needs a place to store its local variables. Collectively, this storage is called the *stack*
- This storage (memory aka "RAM") is a series of storage spaces and their numerical addresses
- Instead of using raw addresses, we use variables to associate a name to an address (kept track by the compiler)
- All of the data/variables for a particular function call are located in a *stack frame*

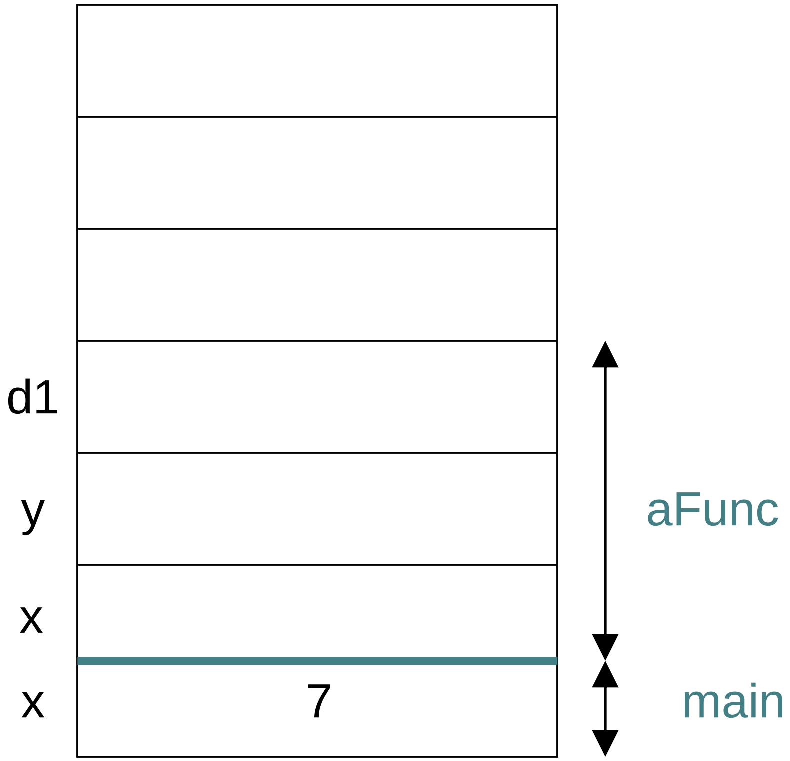



```
void aFunc(int x, int y) {
  double d1, d2;
  int i;
}
```

Functions & Memory (cont)

- When a function is called, a new stack frame with a starting address is set aside
- Parameters and return values are passed *by copy* (ie, they're copied into and out of the stack frame)
- When a function finishes, all its stack frame is reclaimed

```
void aFunc(int x, int y) {
  double d1 = x + y;
}
int main(int argc,
         const char ** argv) {
  int x = 7;
  aFunc(1, 2);
  aFunc(2, 3);
  return 0;
}
```



Function Prototype

- The function prototype *declares* the interface, or input and output parameters of the function, leaving the implementation for the function *definition*.
- The function prototype or declaration has the following syntax:

```
<type> <function name>(<type list>);
```

- Example: A function that prints the card (J) given the card number (11) as input:

```
void printcard(int);
```

(This is a **void** function - a function that does not return anything.)

- You can write a function prototype anywhere in your program
- You can write a function prototype many times in your program

`return` and `exit`

- A function may return values to its environment
  - `return`
  - `exit`
- `return` returns a value to the caller function
  - `return( -1 ), return( i+j),` etc.
  - Control is given to the caller
  - Memory as occupied by the *function* is reclaimed
  - In `main(),` calling `return` basically exits the program (returns to the caller which is the operating system)
- `exit` returns control and value directly to the operating system
  - `exit( -1 ), exit( i+j ),` etc.
  - Usually in case of error or you want to exit the program early without going back to `main()`
  - The whole program exits and the whole memory space is reclaimed by the OS
  - The return code may be used to signal to the OS on the state/error encountered before exiting

Function Definition

- The function definition can be placed anywhere in the program after the function prototypes.

- You can place a function definition in front of `main()` . In this case there is no need to provide a function prototype for the function, since the function is already defined before its use.

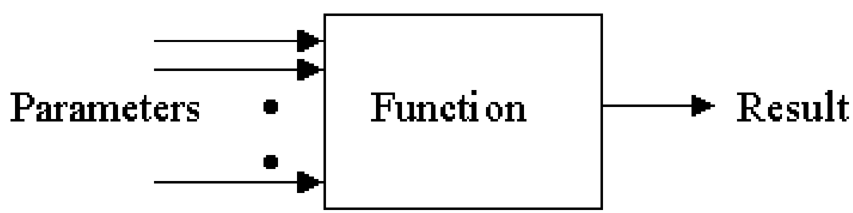
- A **function definition** has following syntax:

```
<type> <function name>(<parameter list>){
    <local declarations>
    <sequence of statements>
}
```

Function Call

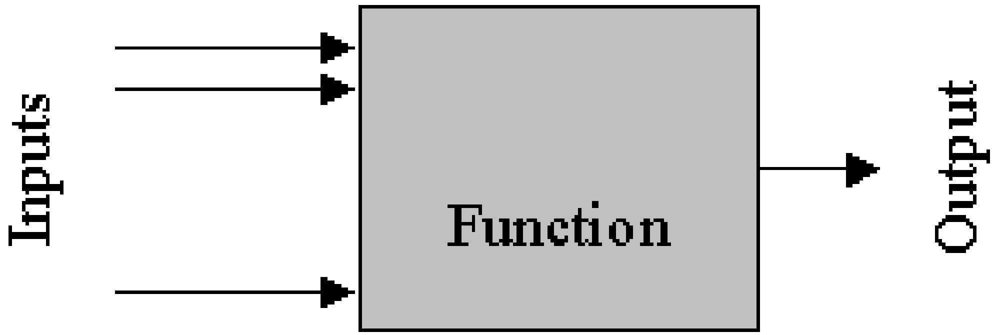
- A **function call** has the following syntax:

```
<function name>(<parameter list>)
```



- There is a one-to-one correspondence between the parameters in a function call and the parameters in the function definition.

Functions



- A function returns a single result (assuming the function is not a void function)
  - The return code usually signals whether an operation is successful or not, or indicates the exit condition
  - If multiple parameters need to be modified, use passing by reference (later)
- One of the statements in the function body should have the form:

```
return <expression>;
```

- The value passed back by return should be of the same type as the return type of the function.

Printing Cards

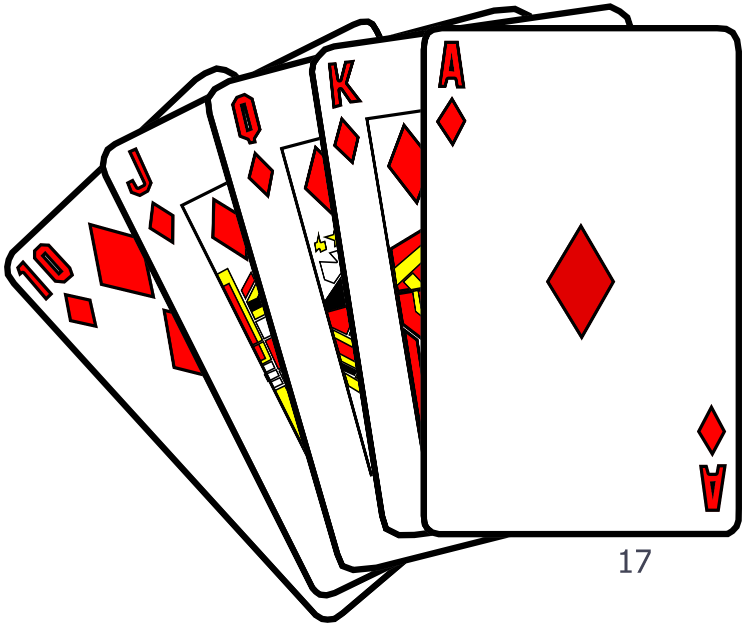
- The `main()` program which calls `printcard()`

```
#include <iostream>
using namespace std;
void printcard(int);        // function prototype
int main(){
    int c1, c2, c3, c4, c5;
    // pick cards
    . . .
    // print cards
    printcard(c1);
    printcard(c2);
    printcard(c3);
    printcard(c4);
    printcard(c5);
    // find score
    // print score
}
```

Printing Cards

- A function that prints the card (J) given the card number (11) as input:

```
void printcard(int cardnum){
      if(cardnum==1)
            cout << "A";
      else if(cardnum>=2 && cardnum<=10)
            cout << cardnum;
      else if(cardnum==11)
            cout << "J";
      else if(cardnum==12)
            cout << "Q";
      else if(cardnum==13)
            cout << "K";
}
```



Absolute Value

```
#include <iostream>
using namespace std;
int absolute(int);              // function prototype for absolute()
int main(){
        int x, y, diff;
        cout << "Enter two integers (separated by a blank): ";
        cin >> x >> y;
        diff = absolute( x - y);
        cout << "The absolute difference between " << x
                  << " and " << y << " is: " << diff << endl;
        return 0;
}
// Define a function to take absolute value of an integer
int absolute(int x){
        if (x >= 0)     return x;
        else            return -x;
}
```

Absolute Value (alternative)

- Note that it is possible to omit the function prototype if the function is placed before it is called.

Hint to the compiler to put the function into the caller codes: Makes codes run faster

```
#include <iostream>
using namespace std;
inline int absolute(int x){
        if (x >= 0)      return x;
        else             return -x;
}
int main(){
        int x, y, diff;
        cout << "Enter two integers (separated by a blank): ";
        cin >> x >> y;
        diff = absolute( x - y);
        cout << "The absolute difference between " << x
                 << " and " << y << " is: " << diff << endl;
        return 0;
}
```

```
return( (x > 0)? x: -x);
```

Adding Numbers

- Consider the following function:

```
int add(int a, int b){
    int result = a+b;      }  return( a + b );
    return result;         }
}
```

- We might call the function using the syntax:

```
int main(){
    int sum;
    sum = add(5, 3);
    return 0;
}
```

- This would result in variable sum being assigned the value 8.

Three-Point Distance

```
#include <iostream>
#include <cmath>
using namespace std;
double dist(double, double, double, double);
int main(){
        double x1, y1, // coordinates for point 1
               x2, y2, // coordinates for point 2
               x3, y3; // coordinates for point 3

        cout << "Enter x & y coordinates of the 1st point: ";
        cin >> x1 >> y1;
        cout << "Enter x & y coordinates of the 2nd point: ";
        cin >> x2 >> y2;
        cout << "Enter x & y coordinates of the 3rd point: ";
        cin >> x3 >> y3;
```

Three-Point Distance

```
      cout <<"The distance from point 1 to 2 is: "
            << dist(x1,y1,x2,y2) << endl;
      cout <<"The distance from point 2 to 3 is: "
            << dist(x2,y2,x3,y3) << endl;
      cout <<"The distance from point 1 to 3 is: "
            << dist(x1,y1,x3,y3) << endl;
      return 0;
}


// Function for computing the distance between 2 pts
double dist(double x1, double y1, double x2, double y2) {
      double dist;
      dist = sqrt( (x2-x1)*(x2-x1) + (y2-y1)*(y2-y1) );
      return dist;
}
```

Pass by Value

- The examples above are all with parameters passed by value
- An important fact to remember about parameter passing by value in a function is that changes to the parameters inside the function body have no effect outside of the function
- This is due to parameter copying in the call

Function Call by Value

```
void f(int x) { cout << "value of x = " << x << endl;
                    x = 4; return; }

int main() { int v = 5;
             f(v);
             cout << "value of v = " << v << endl;
             return 1;}
```

Output: Value of x = 5

Value of v = 5

- When a variable *v* is passed *by value* to a function *f*, its value is copied to the corresponding variable *x* in *f*
- Any changes to the value of *x* does NOT affect the value of *v*

Pass by Value: Example 1

- For example, consider the following code:

```
int sum(int a, int b){
        a = a + b;
        return a;
}
void main(){
        int x, y, z;
        x = 3;  y = 5;
        z = sum(x,y);
}
```

- What is the value of `x`, `y`, and `z` at the end of the `main()` program?

Pass by Value: Example 1

- The answer: 3, 5, and 8.

- Even though the value of parameter `a` is changed, the corresponding value in variable `x` does not change.
  - This is why this is called pass by value.
- The value of the original variable is *copied* to the parameter, therefore changes to the value of the parameter do not affect the original variable.
- In fact, all information in local variables declared within the function will be lost when the function terminates.
- The only information saved from a pass by value function is in the return statement.

Pass by Value: Example 2

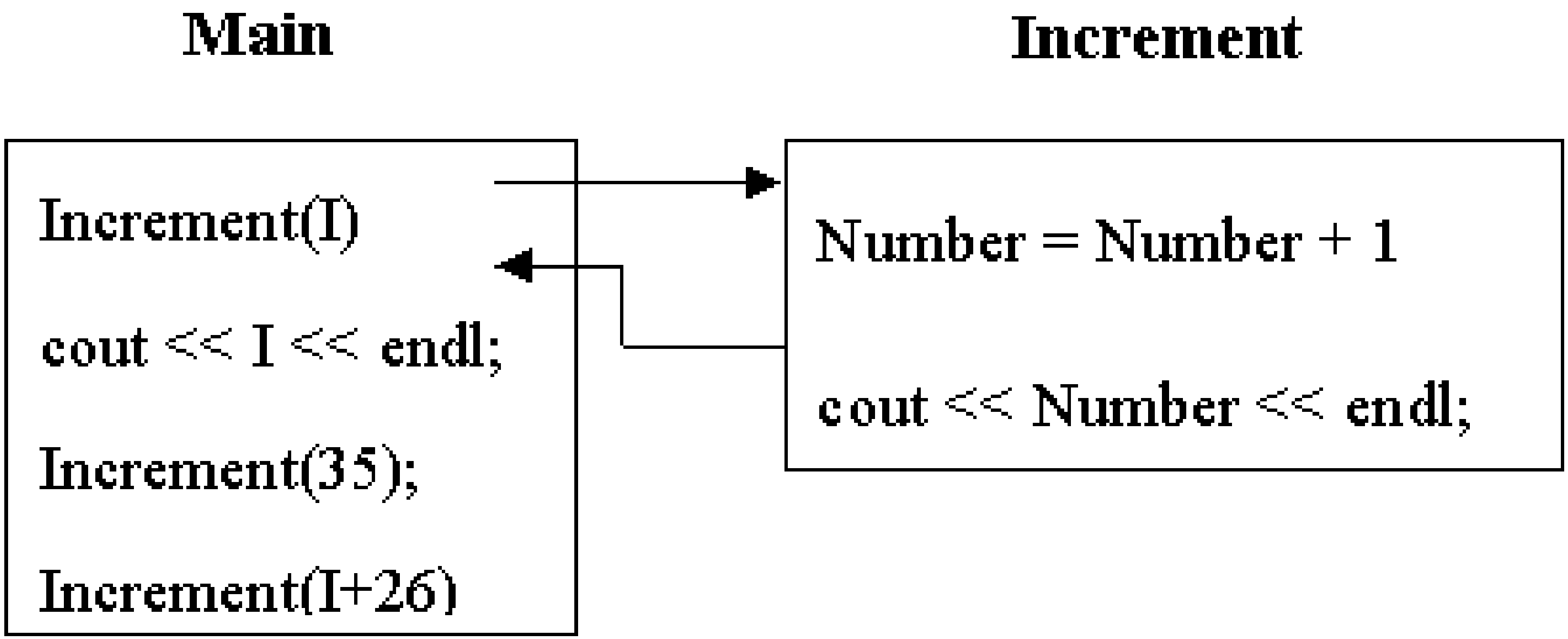
- **An example to show how the function does not affect a variable which is used as a parameter:**

```
// Test the effect of a function
// on its parameter
#include <iostream>
using namespace std;


void Increment(int Number) {
        Number = Number + 1;
        cout << "The parameter Number is: "
             << Number << endl;
}
```

Pass by Value: Example 2

```
void main() {
        int I = 10;

        //parameter is a variable
        Increment(I);
        cout << "The variable I is: "
             << I << endl;

        //parameter is a constant
        Increment(35);
        cout << "The variable I is: "
             << I << endl;

        //parameter is an expression
        Increment(I+26);
        cout << "The variable I is: "
             << I << endl;
}
```

Pass by Value: Example 2



Pass by Value: Example 3

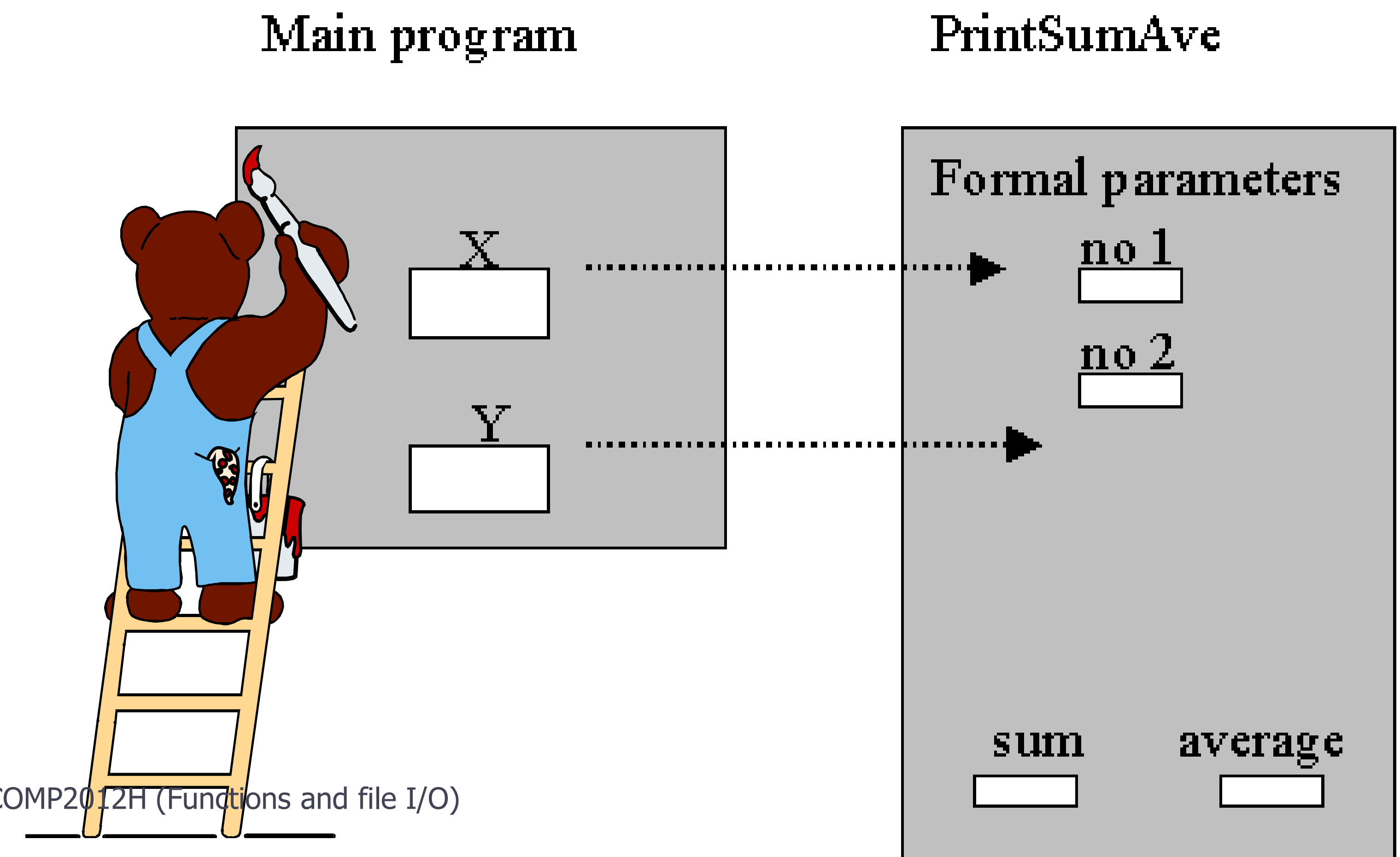
```
// Print the sum and average of two numbers
// Input: two numbers x & y
// Output: sum - the sum of x & y
//         average - the average of x & y
#include <iostream>
using namespace std;
void PrintSumAve ( double, double );
void main ( ) {
        double x, y;

        cout << "Enter two numbers: ";
        cin >> x >> y;
        PrintSumAve ( x , y );
}
```

Pass by Value: Example 3

```
void PrintSumAve (double no1, double no2) {
        double sum, average;

        sum = no1 + no2;
        average = sum / 2;
        cout << "The sum is " << sum << endl;
        cout << "The average is " << average << endl;
}
```

Pass by Value: Example 3

- Data areas after call to `PrintSumAve()` :

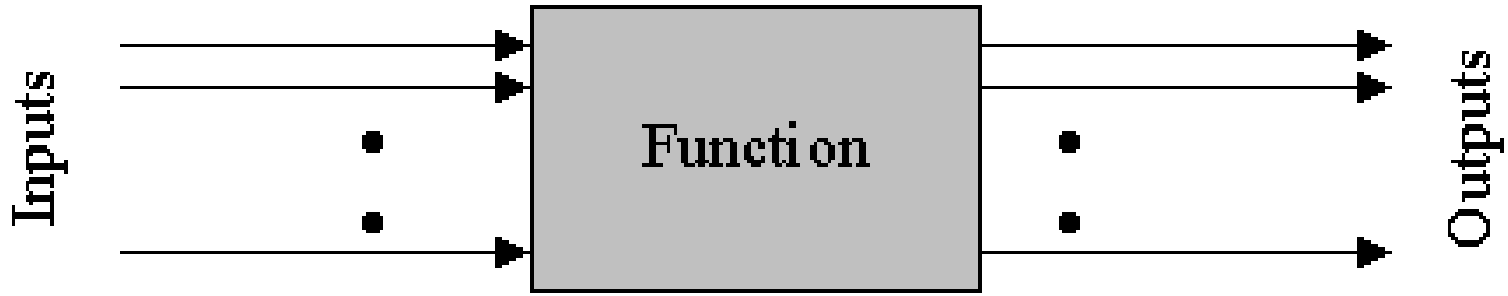


Default Values and Overloading

```
void foo( int i = 1, int j = 2, int k = 3 ){
  cout << "(i, j, k) = (" << i << ", " << j << ", " << k << ")\n";
}

// compilation error -- clash with the above definition
/* void foo( int i ){
  cout << " 2 * i = " << 2 * i << endl;
}
*/

void foo( char * cptr ){
  cout << cptr;
}
int main(){

  foo( 10 );
  foo( 10, 9 );
  foo( “abcde\n” );
  return 0;
}
```

```
cssu5:~> a.out
(i, j, k) = (10, 2, 3)
(i, j, k) = (10, 9, 3)
abcde
```

Multiple Outputs: Passing Parameters by Reference



- To have a function with multiple outputs, we have to use pass by reference.
- Remember: reference is an alias. Both variables, though of different names, refer to the same memory space
- Reference (address) of parameter is passed to the function, instead of its value.
- If the function changes the parameter value, the changes will be reflected in the program calling it.
- How to pass parameters by reference:

```
<type>& <variable>, ... , <type>& <variable>
```

Reference Variables

- Reference and constant reference variables are commonly used for parameter passing to a function
- They can also be used as local variables or as class data members
- A reference (or constant reference) variable serves as an alternative name for an object (an existing memory location)

```
int m = 10;
int & j = m;
cout <<"value of m = " << m << endl; //value of m printed is 10
j = 18;
cout << "value of m = " << m << endl; //value of m printed is 18
```

- A reference variable must always refer to some other object.

```
int m = 10;
int & j = m; //valid
int & k; //compilation error
int & k = 1; //cannot assign a non-constant reference to a
             // constant: compilation error
const int & n = 10+22; // OK. n is 32.
```

Constant Reference

- A *constant reference variable* *v* refers to an object whose value cannot be changed through *v*.

```
int m = 8;
const int & j = m; //ok
m = 16; //valid and now j is changed to 16
j = 20; //compilation error as j is a constant reference
int & m = m;  // compilation error as a reference cannot refers to itself

const int n = 10; // same as int const n = 10;
int & k = n;  // cannot assign a reference to a constant variable:
              //compilation error as n is a constant
const int & r = n; // ok as both r and n are constants
```

Types of Variable and Their Allowed References

- `const int x = 10` means that the *cell/object* `x` itself is a constant
- `const int & y = x` means that the *reference* `y` is a constant
  - A non-constant reference CANNOT refer to a constant cell

	const reference	(Non-constant) Reference
Object	ok	ok
Constant object	ok	X

lvalue and rvalue

- An lvalue (locator value) represents an object that occupies some identifiable location in memory (i.e. has an address).
  - Not all lvalues can be assigned to. Those that can are called *modifiable lvalues*.
  - An lvalue refers to an object that persists beyond a single expression.
- An rvalue is an expression that does not represent an object occupying some identifiable writable location in memory.
- `4 = var;          // ERROR, as the left operand 4 is not lvalue`
- `(var + 1) = 4; // ERROR, as (var+1) is not lvalue`

```
int globalvar = 20;

int& foo()
{
    return globalvar;
}

int main()
{
    foo() = 10;  // foo returns a lvalue
    return 0;
}
```

```
const int a = 10; // 'a' is a (non-modifiable)lvalue
a = 10;           // but it can't be assigned!
```

Pass by Reference: An Example

- To show how the function affects a variable which is used as a parameter:

```
#include <iostream>
using namespace std;
void Increment(int& Number){
  Number = Number + 1;
  cout << "The parameter Number: " << Number << endl;
}

int main(){
  int I = 10;
  Increment(I);   // parameter is a variable
  cout << "The variable I is: " << I << endl;
  return 1;
}
```

Function Call by Reference

```
void f(int &x) { cout << "value of x = " << x << endl;
                 x = 4; return; }

int main() { int v = 5;
             f(v);
             cout << "value of v = " << v << endl;
             return 1;}
```

Output: Value of x = 5

Value of v = 4

- When a variable *v* is passed *by reference* to a parameter *x* of function *f*, *v* and the corresponding parameter *x* refer to the same variable
- Any changes to the value of *x* DOES affect the value of *v*

Pass by Reference: Example 2

- It is possible to use both pass by reference and pass by value parameters in the same function.

```
// Print the sum and average of two numbers
// Input: two numbers x & y
// Output: sum - the sum of x & y
//         average - the average of x & y
#include <iostream>
using namespace std;

void SumAve (double, double, double&, double&);
```

Pass by Reference: Example 2

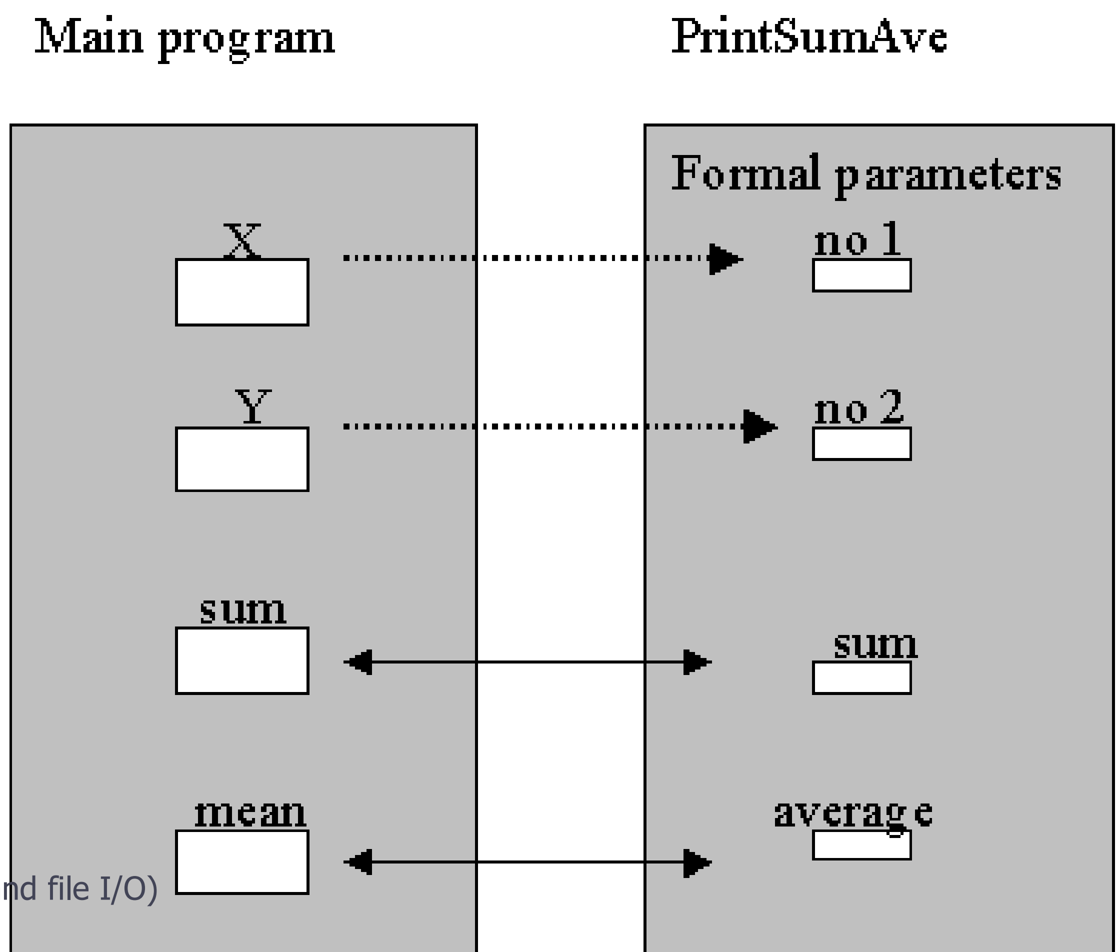
```
void main ( ) {
        double x, y, sum, mean;

        cout << "Enter two numbers: ";
        cin >> x >> y;
        SumAve (x, y, sum, mean);
        cout << "The sum is " << sum << endl;
        cout << "The average is " << mean << endl;
}
```

Don't need to put const & here because double is a primitive type; copying is not expensive

```
void SumAve(double no1, double no2, double& sum,
      double& average) {
        sum = no1 + no2;
        average = sum / 2;
}
```

Pass by Reference: Example 2

- Data areas after call to SumAve:



Pass by Reference: Example 3

```
// Compare and sort three integers
#include <iostream>
using namespace std;
void swap (int&, int&);
void main ( ) {
        int first, second, third;  // input integers
        // Read in first, second and third.
        cout << "Enter three integers: ";
        cin >> first >> second >> third;
        if (first > second) swap (first, second);
        if (second > third) swap (second, third);
        if (first > second) swap (first, second);
        cout << "The sorted integers are " << first <<
             " , " << second << " , " << third << endl;
}
```

Pass by Reference: Example 3

```
// Function for swapping two integers
void swap (int& x, int& y) {
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

Function Call by Value

```
void f(int x) { cout << "value of x = " << x << endl;
                  x = 4; return; }

main() { int v = 5;
              f(v);
              cout << "value of v = " << v << endl;}
```

Output: Value of x = 5

Value of v = 5

- When a variable *v* is passed *by value* to a function *f*, its value is copied to the corresponding variable *x* in *f*
- Any changes to the value of *x* does NOT affect the value of *v*
- Call by value is the default mechanism for parameter passing in C++

Function Call by Reference

```
void f(int &x) { cout << "value of x = " << x << endl;
                 x = 4; return; }

main() { int v = 5;
         f(v);
         cout << "value of v = " << v << endl;}
```

Output: Value of x = 5

Value of v = 4

- When a variable *v* is passed *by reference* to a parameter *x* of function *f*, *v* and the corresponding parameter *x* refer to the same variable
- Any changes to the value of *x* DOES affect the value of *v*

Function Call by Constant Reference

```
void f( const int &x ) { cout << "value of x = " << x << endl;
                  x = 4;  // invalid as x is a rvalue
                  }
int main() { int v = 5;
             f(v);
             cout << "value of v = " << v << endl; return 1;
           }
```

- Passing variable *v by constant reference* to parameter *x* of *f* will NOT allow any change to the value of *x.*
- It is appropriate for passing large objects that should not be changed by the called function.

Usage of Parameter Passing

- *Call by value* is appropriate for small objects that should not be changed by the function
  - Because small objects may be copied without too much overhead
- *Call by constant reference* is appropriate for large objects that should not be changed by the function
  - Large objects should not be copied for efficiency concern
- *Call by reference* is appropriate for all objects that may be changed by the function
  - An alternative approach is to use pointer, and pass the address of the object into the function

Final remarks

- Advantage of passing by reference: Large data objects do not have to be copied
  - Saving much CPU time and storage
- Be careful with the parameters though
  - You may modify them unintentionally
  - To safeguard modification, use `const` in the parameter instead

```
foo( const int & bar, const int & foobar);
```

  - For code clarity and to minimize bug, you should always put `const` in front of the variable if you are not going to change the variable in the function

Function Pointer: Passing function name as a parameter to a function

- The executable of a program gets a certain space in the main-memory
  - To store the compiled codes and variables, and steps to manipulate the variables in machine codes
- A function name is an address pointing to the execution codes of the function
  - When the function is called, the codes and variables are loaded to the call stack at execution
- A function name is a pointer which points to the address of the function
- Therefore, a function name can be passed as a parameter to another function

```
#include <iostream>
using namespace std;

// put a and b in ascending order
void ascending( double & a, double & b ){
  if( a > b ){
    double tmp = a;
    a = b;
    b = tmp;
  }
  return;
}

// put a and b in descending order
void descending( double & a, double & b ){
  if( a < b ){
    double tmp = a;
    a = b;
    b = tmp;
  }
  return;
}
```

This is a function prototype with (*) added around function name

```
// function pointer
void order( void (*criteria)(double &, double &),
            double & a,
            double & b) {
  (*criteria)(a, b);  // same as criteria(a,b);
  return;
}

int main(){
  double x = 0.7;
  double y = 0.5;

  order( ascending, x, y );
  cout << x << "  " << y << endl;

  order( descending, x, y );
  cout << x << "  " << y << endl;

  return 1;
}
```

Output:

```
0.5  0.7
0.7  0.5
```

Array Element Pass by Value

- Individual array elements can be passed by value or by reference
- Pass-by-value example:

```
void printcard(int c) {
        if(c==1)
             cout << "A";
        ...
}
void main() {
        int cards[5] ;
        ...
        for(int n=0; n<5; n++)
             printcard(card[n]);
}
```

Array Element Pass by Reference

- Pass-by-reference example:

```
void swap(int& x, int& y) {
        int temp;
        if (x > y){
              temp = x;
              x = y;
              y = temp;
        }
}
void main() {
        int A[10] = {9,8,7,6,5,4,3,2,1,0};
        swap(A[3], A[5]);
}
```

Array Element Pass by Reference

- Before:

9	8	7	6	5	4	3	2	1	0
0	1	2	3	4	5	6	7	8	9

- After:

9	8	7	4	5	6	3	2	1	0
0	1	2	3	4	5	6	7	8	9

Passing Entire Arrays to Functions

- Arrays can be passed to functions in their entirety.
- All that is required is the address of the first element and dimensions of the array.
- The remainder of the array will be passed by reference automatically.

Arrays to Functions: An Example

```
//Find the largest value in an array
//input: n - number of elements to check
//    a[ ] - array of elements
// output:index to the largest element
#include <iostream>
using namespace std;
int max_element(int n, const int a[]) {
        int max_index = 0;
        for (int i=1; i<n; i++)
                if (a[i] > a[max_index])
                        max_index = i;
        return max_index;
}
int main() {
        int A[10] = {9,8,7,6,5,4,10,2,1,0};
        cout << A[max_element(10,A)] << endl; return 1;
}
```

> This is passed as a reference, i.e., not the entire array is passed.
> The same effect as int * a

Arrays to Functions: Example 2

```
//Add a[i] and b[i] and store the sum in c[i]
//Array elements with subscripts ranging from
//0 to size-1 are added element by element
void  add_array(int size, const double a[],
                const double b[], double c[]){
      for (int i=0; i<size; i++)
            c[i] = a[i] + b[i];
}
```

In main():

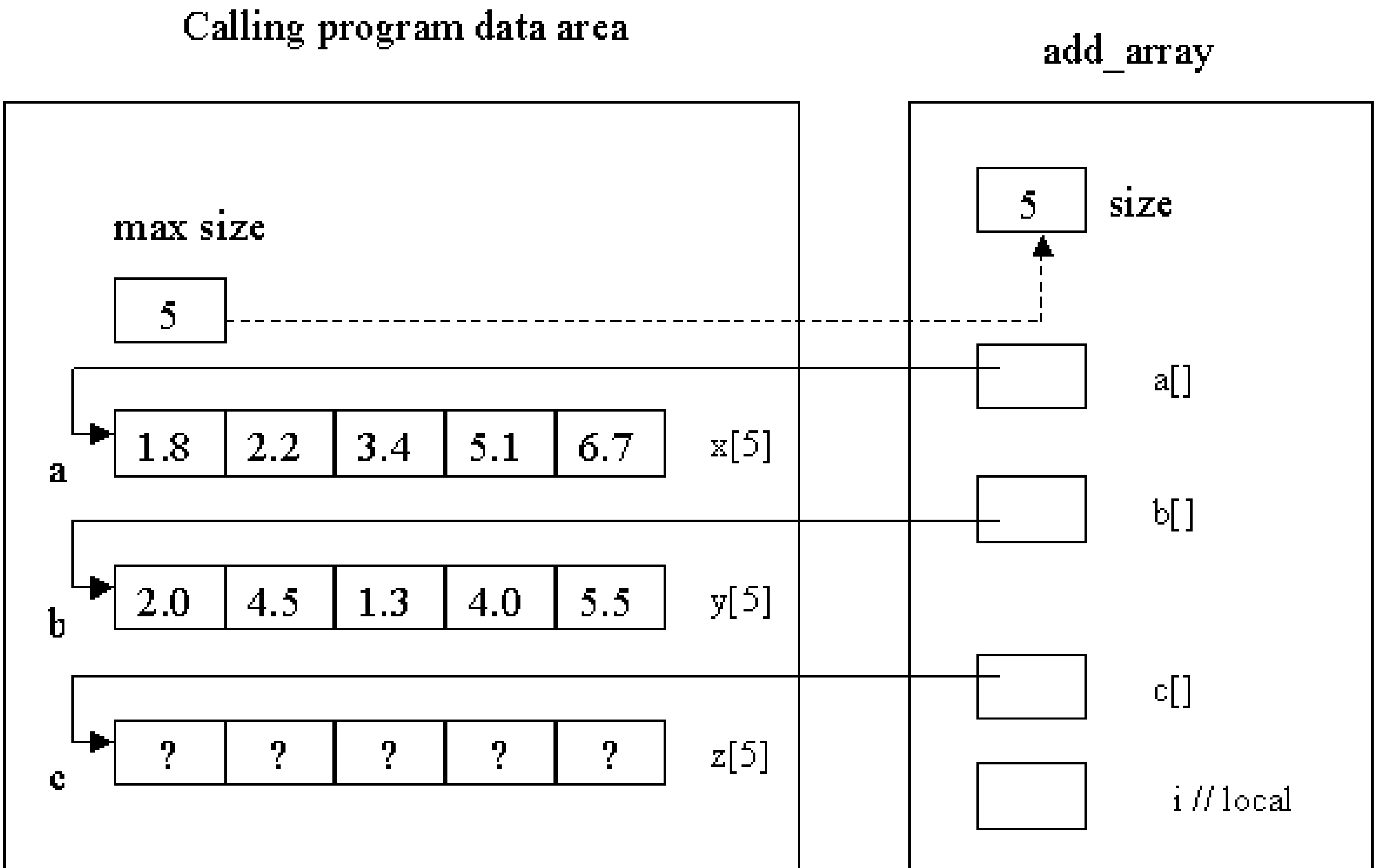
```
      add_array (5, x, y, z );
```

May also be written as:

```
…, const double * a,
const double * b,
const double * c)
```

to refer to unchangeable array elements

Arrays to Functions: Example 2 (Note that the arrays are not copied)



Passing Multidimensional Arrays

- How to pass a multidimensional array to a function:

```
void displayBoard(int b[][4]);
// or simply displayBoard(int [][4]);

void main(){
     int board [4][4];
     ...
     displayBoard(board);
     ...
}
void displayBoard(int b[][4]){
// could also be: void displayBoard(int b[4][4]){
// but NOT: void displayBoard(int b[][]){
     ...
}
```

Passing Multi-Dimensional Array

- When passing a multidimensional array to a function, only the size of the 1st dimension is optional, the 2nd, 3rd, etc. dimensions *have to* be specified.
- This is because the compiler treats multi-dimensional array as a linear memory space and calculates address accordingly
- For 2D array, the physical address of `int A[i][j]` in a statement is computed at compilation time as `A + 4*i*MAX_j + 4*j`, which means that the dimension of each row (`MAX_j`) has to be known to the compiler
- For an array index `int A[i][j][k]` in a statement, the physical address representation is computed at compilation time as

```
A + 4 * i * (MAX_j * MAX_k) +
     4 * j * MAX_k + k * 4
```

  where `MAX_j` and `MAX_k` are the maximum ranges of `j` and `k`, respectively
- As shown above, the dimension sizes are needed for address computation at compilation time.
- C++ compiler relies on the programmer to make sure that the array would NOT be accessed out of range and hence will not check whether `i, j, k` exceed their dimension sizes during compilation time.

Modify a Variable with Return by Reference

```
int & bar (int n, int * iptr){
  return iptr[n/2];
}
const int & foobar (int n, int * iptr){
  return iptr[n/2];
}

int main(){
  int j;
  int A[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};

  j = bar( 10, A );
  cout << j << endl;
  bar( 10, A ) = 10;  // address is written into
  cout << A[ 10/2 ] << endl;

  j = foobar( 10, A );
  cout << j << endl;
  foobar( 10, A ) = 10; // not valid: compiler complains

  return 0;
}
```

```
5
10
10
```

Testing and Debugging Functions

- One major advantage of functions is that they can be designed, coded and tested separately from the rest of the program.
- Use a "driver" program to test a function with several inputs:

```
void main( ) {
        int i;
        for (i = 1; i <= 13; i++){
                printcard(i);
                cout << " ";
        }
}
```

Testing and Debugging Functions

- If a yet-to-be written function is needed in testing a program, replace it with a "stub" for testing.
- A stub has the same interface as the original function, but not the full implementation.
- Oftentimes, a stub contains just a simple `return` or `cout` command.

```
void printcard(int i) {
        cout << i;
}
```

Variable Scope

Scope

The scope of a declaration is the block of code where the identifier is valid for use.

- A global declaration is made outside the bodies of all functions and outside the main program. It is normally grouped with the other global declarations and placed at the beginning of the program file.
- A local declaration is one that is made inside the body of a function. Locally declared variables cannot be accessed outside of the function they were declared in.
- It is possible to declare the same identifier name in different parts of the program.

Scope: Example 1

```
int y = 38;
void f(int, int);
void main( ){
        int z=47;
        while(z<400){
              int a = 90;
              z += a++;
              z++;
        }
        y = 2 * z;
        f(1, 2);
}
void f(int s, int t){
        int r = 12;
        s = r + t;
        int i = 27;
        s += i;
}
```

scope of `y`

scope of `f`

scope of `z`

scope of `a`

scope of `r`

scope of `i`

scope of `s & t`

Scope: Example 2

- `Number` in `Increment()` is the global variable.

```
#include <iostream>
 using namespace std;
 int Number; //global variable
 void Increment(int Num) {
         Num = Num + 1;
         cout << Num << endl;
         Number = Number + 1;
 }
 void main() {
         Number = 1;
         Increment(Number);
         cout << Number << endl;
 }
```

```
Output: 2 2
```

Global Variables

- Undisciplined use of global variables may lead to confusion and debugging difficulties.

- Instead of using global variables in functions, try passing local variables by reference.

Scope: Example 3

```
int Number; //global variable
void Increment(int& Num) {
        Num = Num + 1;
        cout << Num << endl;
        Number = Number + 1;
}
void main() {
        Number = 1;
        Increment(Number);
        cout << Number << endl;
}
```

```
Output: 2 3
```

- **When Increment is called,** `Num` **refers to global variable** `Number`
- `Number = Number + 1` **also refers to global variable** `Number`.

Scope: Example 4

```
int Number; //global variable
void Increment(int Number) {
        Number = Number + 1;
        cout << Number << endl;
}
void main() {
        Number = 1;
        Increment(Number);
        cout << Number << endl;
}
```

This is local variable

Output: 2 1

- The scope of the global variable Number does not include Increment(), because Increment() already has a local parameter of the same name.
- Thus, the changes made to Number are lost when control returns to the main program.

Scope: Example 5

Local variables

```
int A,B,C,D;
void Two(int A, int B, int& D) {
      B = 21; D = 23;
      cout <<A<< " " <<B<< " " <<C<< " " <<D<< endl;
}
void One(int A, int B, int& C) {
      int D;     // Local variable
      A = 10; B = 11; C = 12; D = 13;
      cout <<A<< " " <<B<< " " <<C<< " " <<D<< endl;
      Two(A,B,C);
}
void main() {
      A = 1; B = 2; C = 3; D = 4;
      One(A,B,C);
      cout <<A<< " " <<B<< " " <<C<< " " <<D<< endl;
      Two(A,B,C);
      cout <<A<< " " <<B<< " " <<C<< " " <<D<< endl;
}
```

Scope: Example 5

- **Output:**

```
10 11 12 13  // from One
10 21 23 23  // from Two
1 2 23 4     // from main
1 21 23 23   // from Two
1 2 23 4     // from main
```

Compiler Scope Rule

- Compiler always looks for local variables of the closest scope first

```
int A=0, B=1;

void foo( int A ){// A is local to foo
   A = 100;
   return;
}

void bar( int B ){// B is local to bar
   B = 100;
   return;
}

int main(){
  int A[10]; // ok
  foo( A );  //compilation error as A is an array
       //and foo takes an integer
  bar( B );
  cout << B;  // print out 1
}
```

`static` Variable

- Static variable is only allocated once and remains within the scope of the function

```
void foo( void ){
  static int i = 0;
  i++;
  cout << i << endl;
  return;
}

int main( void ){
  for( int j = 0; j < 10; j++)
    foo();
  return 0;
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Recursion

Outline

- Recursion definition and techniques
- 6 examples
  - Factorial
  - Exponential function
  - Number of digit 0
  - Fibonacci number
  - Binary search
  - Permutation

Recursive Thinking

- In some problems, it may be natural to define the problem in terms of the problem itself.
- Recursion is useful for problems that can be represented by a simpler version of the same problem.
- Example: the factorial function

```
6! = 6 * 5 * 4 * 3 * 2 * 1
```

We could write:

```
6! = 6 * 5!
```

Recursion

- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- A function is defined recursively if it has the following two parts
- An anchor or base case
  - The function is defined for one or more specific values of the parameter(s)
- An inductive or recursive case
  - The function's value for current parameter(s) is defined in terms of previously defined function values and/or parameter(s)
- Recursive call (also called the recursion step)
  - The function launches (calls) a fresh copy of itself to work on the smaller problem
  - Can result in many more recursive calls, as the function keeps dividing each new problem into two conceptual pieces
  - This sequence of smaller and smaller problems must eventually converge on the base case; otherwise the recursion will continue forever

Recursion

- Recursion is one way to decompose a task into smaller subtasks. At least one of the subtasks is a smaller example of the same task.
- The smallest example of the same task has a non-recursive solution.

Example: The factorial function

```
n! = n * (n-1)! and 1! = 1
```

Example 1: factorial function

In general, we can express the factorial function as follows:

```
n! = n * (n-1)!
```

Is this correct? Well… almost.

The factorial function is only defined for *positive* integers. So we should be a bit more precise:

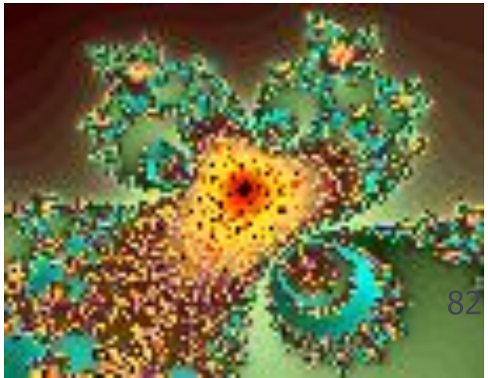
```
n! = 1            (if n is equal to 1)
n! = n * (n-1)!   (if n is larger than 1)
```

factorial function

The C++ equivalent of this definition:

```
int fac(int numb){
    if(numb<=1)
        return 1;
    else
        return numb * fac(numb-1);
}
```

***recursion* means that a function calls itself**



factorial function

- Assume the number typed is 3, that is, numb=3.

```
fac(3) :
3 <= 1 ?                 No.
fac(3) = 3 * fac(2)
  fac(2) :
     2 <= 1 ?            No.
     fac(2) = 2 * fac(1)

         fac(1) :
            1 <= 1 ?     Yes.
            return 1

     fac(2) = 2 * 1 = 2
     return fac(2)

 fac(3) = 3 * 2 = 6
 return fac(3)

 fac(3) has the value 6
```

```
int fac(int numb){
   if(numb<=1)
      return 1;
   else
      return numb * fac(numb-1);
}
```

factorial function

For certain problems (such as the factorial function), a recursive solution often leads to short and elegant code. Compare the recursive solution with the iterative solution:

Recursive solution

```
int fac(int numb){
   if(numb<=1)
     return 1;
   else
     return numb*fac(numb-1);
}
```

Iterative solution

```
int fac(int numb){
   int product=1;
   while(numb>1){
     product *= numb;
     numb--;
   }
   return product;
}
```

Recursion

To trace recursion, recall that function calls operate as a stack – the new function is put on top of the caller

We have to pay a price for recursion:

- calling a function consumes more time and memory than adjusting a loop counter.
- high performance applications (graphic action games, simulations of nuclear explosions) hardly ever use recursion.

In less demanding applications recursion is an attractive alternative for iteration (for the right problems!)

Infinite Loop...

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
   ...
```

```
 int result = 1;
while(result >0){
   ...
   result++;
}
```

Oops!

Oops!

Infinite Recursion

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
    return numb * fac(numb-1);
}
```

Oops!

No termination condition

Or:

```
int fac(int numb){
    if (numb<=1)
        return 1;
    else
        return numb * fac(numb+1);
}
```

Oops!

Recursion

We must always make sure that the recursion *bottoms out*:

- A recursive function must contain at least one non-recursive branch.
- The recursive calls must eventually lead to a non-recursive branch.

Example 2: Exponential Function

- How to write `exp(int numb, int power)` recursively?
  - n^p = n * n^(p-1)

```
int exp(int numb, int power){
      if(power ==0)
            return 1;
      return numb * exp(numb, power -1);
}
```

Example 3: number of zero

- Write a recursive function that counts the number of zero digits in an integer
- **`zeros(10200)`** returns 3.
- #zeros(n) = #zeros(n/10) + y, where y is 1 if it is 0, or 0 otherwise

```
int zeros(int numb){
    if(numb==0)                  // 1 digit (zero/non-zero):
        return 1;                // bottom out.
    else if(numb < 10 && numb > -10)
        return 0;
    else                         // > 1 digits: recursion
        return zeros(numb/10) + zeros(numb%10);
}
```

```
zeros(10200)
zeros(1020)                                      + zeros(0)
zeros(102)                            + zeros(0) + zeros(0)
zeros(10)                 + zeros(2) + zeros(0) + zeros(0)
zeros(1) + zeros(0) + zeros(2) + zeros(0) + zeros(0)
```

Example 4: Fibonacci numbers

- Fibonacci numbers:

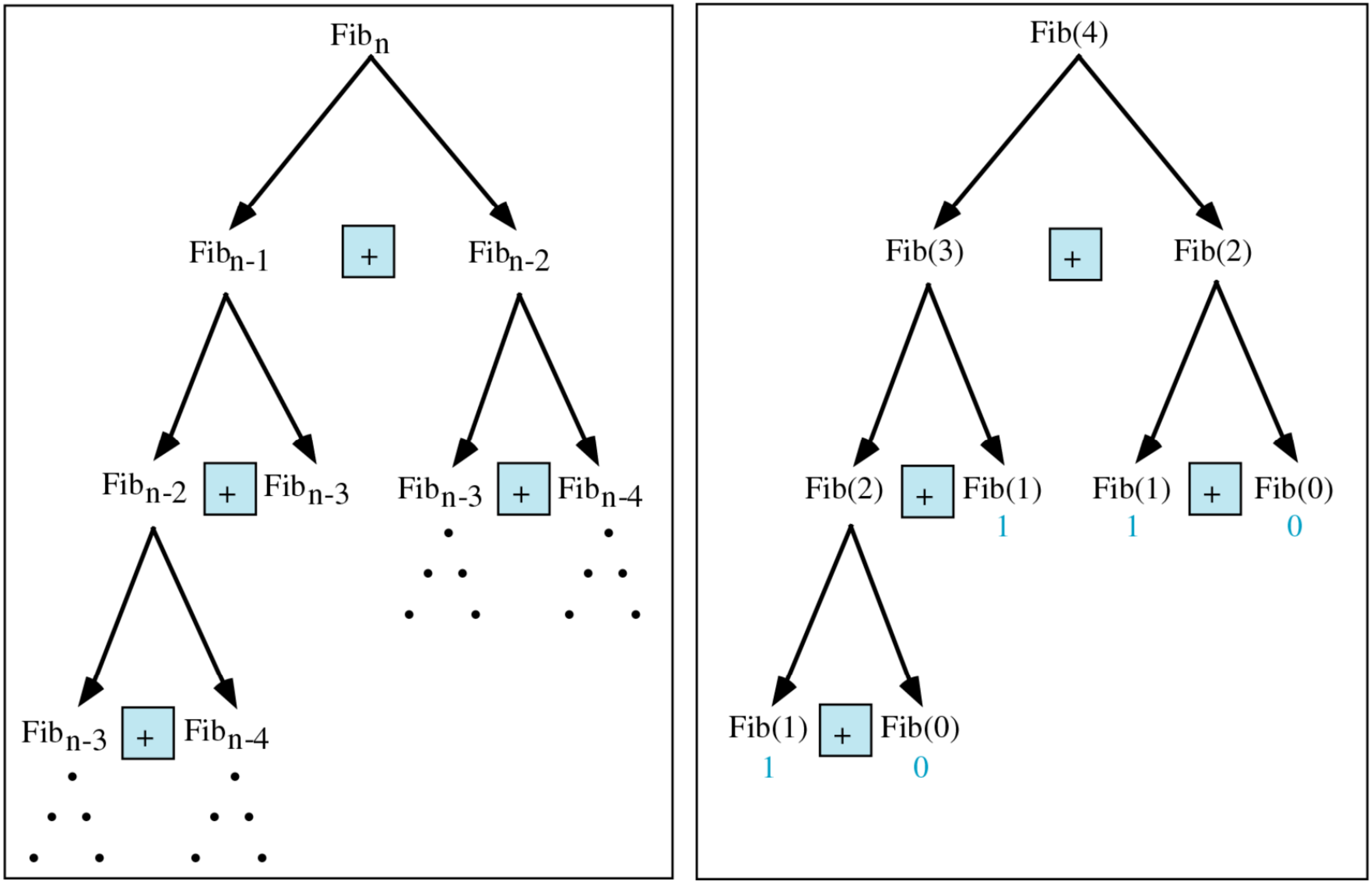
  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  where each number is the sum of the preceding two.

- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(number) = F(number-1)+ F(number-2);`

Example 2: Fibonacci numbers

```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well
int fib(int number)
{
   if (number == 0) return 0;
   if (number == 1) return 1;
   return (fib(number-1) + fib(number-2));
}

int main(){       // driver function
   int inp_number;
   cout << "Please enter an integer: ";
   cin >> inp_number;
   cout << "The Fibonacci number for "<< inp_number
          << " is "<< fib(inp_number)<<endl;
  return 0;
}
```


Fibn
Fibn-1
+
Fibn-2
Fibn-2
+
Fibn-3
Fibn-3
+
Fibn-4
Fibn-3
+
Fibn-4
Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
1
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0

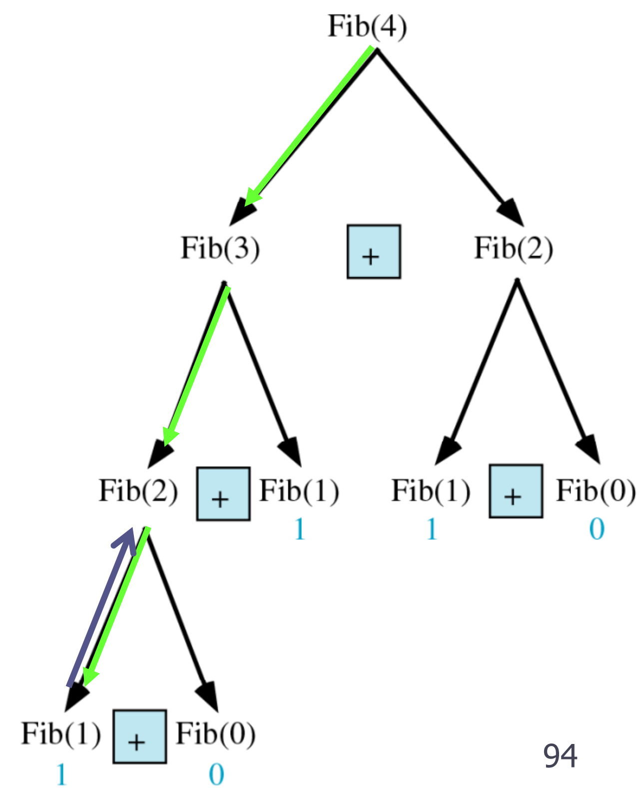
(a) Fib(n) (b) Fib(4)

Trace a Fibonacci Number

- Assume the input number is 4, that is, num=4:

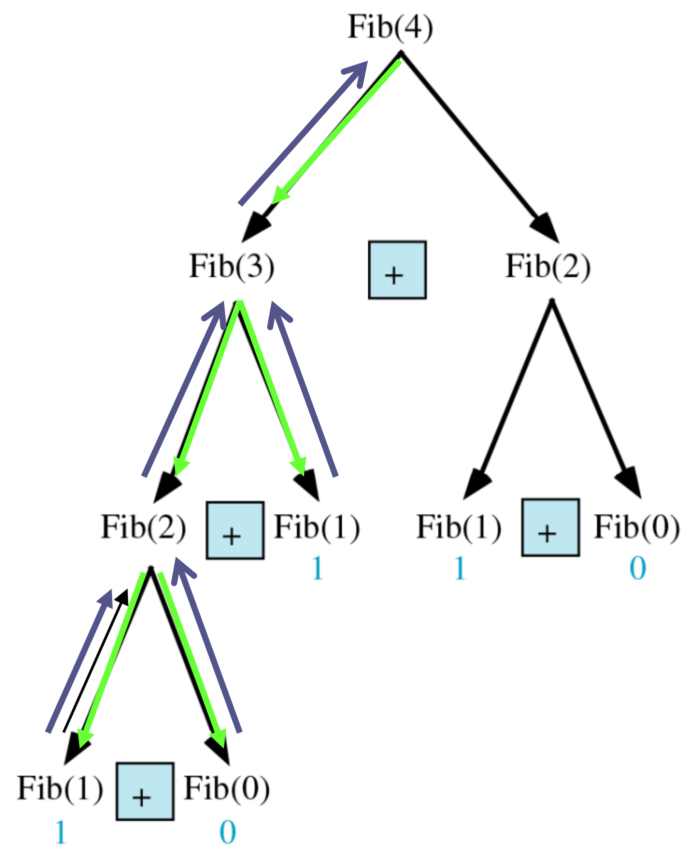
```
fib(4):
  4 == 0 ? No;    4 == 1?   No.
  fib(4) = fib(3) + fib(2)
  fib(3):
     3 == 0 ? No; 3 == 1? No.
     fib(3) = fib(2) + fib(1)
     fib(2):
       2 == 0? No; 2==1? No.
       fib(2) = fib(1)+fib(0)
        fib(1):
           1== 0 ? No; 1 == 1? Yes.
               fib(1) = 1;
       return fib(1);
```

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```



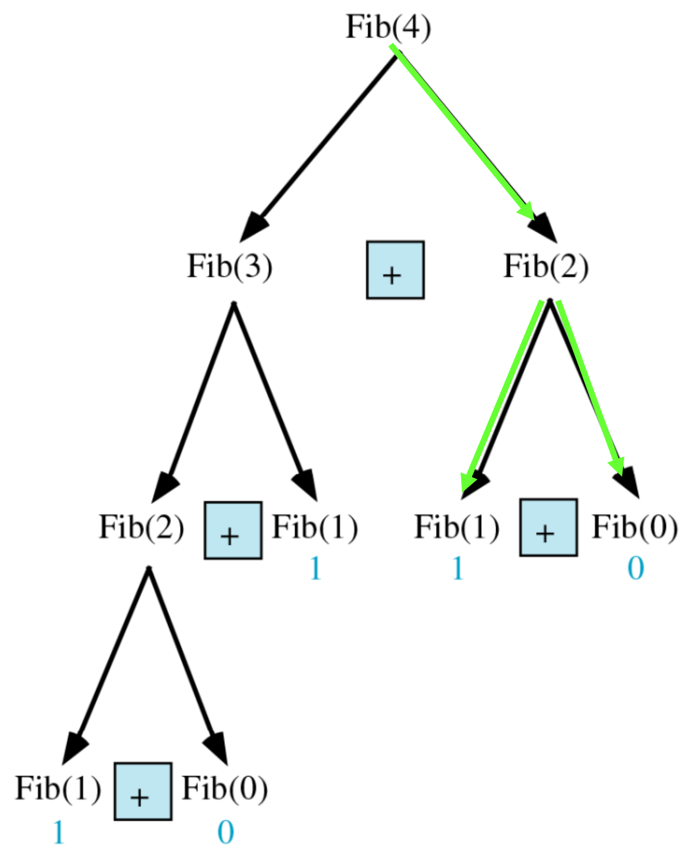
Trace a Fibonacci Number

```
fib(0):
      0 == 0 ?  Yes.
      fib(0) = 0;
      return fib(0);
 fib(2) = 1 + 0 = 1;
 return fib(2);
fib(3) = 1 + fib(1)
  fib(1):
   1 == 0 ? No; 1 == 1? Yes
   fib(1) = 1;
   return fib(1);
fib(3) = 1 + 1 = 2;
return fib(3)
```



Trace a Fibonacci Number

```
fib(2):
2 == 0 ? No; 2 == 1? No.
fib(2) = fib(1) + fib(0)
fib(1):
    1== 0 ? No; 1 == 1?  Yes.
       fib(1) = 1;
 return fib(1);
  fib(0):
     0 == 0 ?   Yes.
       fib(0) = 0;
       return fib(0);
  fib(2) = 1 + 0 = 1;
  return fib(2);
fib(4) = fib(3) + fib(2)
       = 2 + 1 = 3;
return fib(4);
```



A Much More Efficient Implementation: Fibonacci number w/o recursion

```
//Calculate Fibonacci numbers iteratively
//much more efficient than the recursive solution

int fib( int num ){
  int fn;       // f[n]
  int f2 = 0;   // f[n-2]
  int f1 = 1;   // f[n-1]

  if( num == 0 )
    return 0;
  if( num == 1 )
    return 1;

  for( int i = 2; i <= num; i++ ){
    fn = f1 + f2;     // f[n] = f[n-1] + f[n-2]
    f2 = f1;          // f[n-2] <- f[n-1]
    f1 = fn;          // f[n-1] <- f[n]
  }
  return fn;
}
```

Example 5: Binary Search in a Sorted Array

- Search for an element in an *ordered* array
  - Sequential search
  - Binary search
- Binary search
  - Compare the search element with the middle element of the array
  - If not equal, then apply binary search to half of the array (if not empty) where the search element may be.

Binary Search (driver)

```
int main() {
    const int array_size = 8;
    int list[array_size]={1, 2, 3, 5, 7, 10, 14, 17};
    int search_value;

    cout << "Enter search value: ";
    cin >> search_value;
    cout << bsearchr(list,0,array_size-1,search_value)
         << endl;

    return 0;
}
```

Binary Search with Recursion

```
// Searches an ordered array of integers using recursion
int bsearchr(const int data[], // input: array
             int first,        // input: lower bound
             int last,        // input: upper bound
             int value         // input: value to find
        )// output: index if found,  otherwise return -1

{  int middle = (first + last) / 2;
   if (data[middle] == value)
      return middle;
   else if (first > last)
      return -1;
   else if (value < data[middle])
      return bsearchr(data, first, middle-1, value);
   else
      return bsearchr(data, middle+1, last, value);
}
```

Binary Search w/o Recursion

```
// Searches an ordered array of integers
int bsearch(const int data[], // input: array
            int size,     // input: array size
            int value     // input: value to find
           ){             // output: if found, return the index
                          // index; otherwise, return -1

      int first, last, middle;
        first = 0;
        last = size - 1;
      while (true) {
        middle = (first + last) / 2;
        if (data[middle] == value)
            return middle;  // found
        else if (first > last)
            return -1;      // not found
        else if (value < data[middle])
            last = middle - 1;  //lower half
        else
            first = middle + 1;  //upper half
    }
}
```

Recursion General Form

- How to write recursively?

```
int recur_fn(parameters){
  if(stopping condition)
      return stopping value;
  // other stopping conditions if needed
  return function of recur_fn(revised parameters)

}
```

Example 6: Permutation

- Generate all the permutation sequences of *n* numbers
  - *n!* number of sequences
  - For *n*=3: abc, acb, bac, bca, cba, cab
- Let the numbers be labeled as a1 a2 a3 … an
- The permuted sequences are
  a1 perm(a2 a3 a4…an),
  a2 perm(a1 a3 a4…an),
  a3 perm(a2 a1 a4…an),
  ….
  an perm(a2 a3 a4 … a1)

- The above can be implemented in a `for` loop with a `swap` function
  - Swap the first element with the ith one, so that the ith one is the leading element
  - Perform permutation recursively
  - Swap ith one back to its original position to start another loop
- (base case) If there is only 1 element, this must be the last element in the permuted sequence and there is nothing to be permuted. In this case, simply print out the whole sequence from the beginning of the array to the end

Permutation Codes

```
template <class T>
inline void swap( T& a, T& b){
  // swap a and b
  T temp = a;
  a = b;
  b = temp;
}
```

For some simple function, may use #define, e.g.,
#define max(x, y) ((x) > (y)? (x): (y))
// #define mult(x, y)((x)*(y))

```
// Permutation codes to permute list[ k: m ]
// simply prints the permuted sequences out list[0: m]
template<class T>
void Perm( T list[], int k, int m ){
  // generate all permutations of list[ k:m ]
  int i;
  if( k == m ){ // base case: only 1 element → simply print things out
    for( i = 0; i <= m; i++ ) // cout from array index 0
      cout << list[ i ];
    cout << endl;
  }
  else
    for( i = k; i <= m; i++ ){
      swap( list[ k ], list[ i ] );  // swap a[i] as the leading symbol
      Perm( list, k+1, m );  // permute with one fewer element and print things out
      swap( list[ k ], list[ i ] ); // restore back to the original sequence for the next iteration
    }
}
```

Usage

```
int main(){

  char str[] = "abcde";

  Perm( str, 2, 4 );  // permute on “cde”
  cout << endl;
  Perm( str, 0, 2 );  // permute on “abc”

}
```

Output:

```
abcde
abced
abdce
abdec
abedc
abecd

abc
acb
bac
bca
cba
cab
```

Recursion vs. Iteration

- Negatives of recursion
  - Overhead of repeated function calls
    - Creating stacks can be expensive in both processor time and memory space
  - Each recursive call causes another copy of the function (actually only the function's variables) to be created
    - Can consume considerable memory
- Iteration
  - Overhead of repeated function calls and extra memory allocation is removed

Recursion vs. Iteration

- Most of the problems that can be solved recursively can also be solved iteratively (nonrecursively)
- A recursive approach is usually chosen in preference to an iterative approach when
  - the recursive approach more naturally mirrors the problem and results in a program that is easier to understand and debug
  - an iterative solution is not apparent

File I/O

Using Input/Output Files

- A computer file
  - is stored on a secondary storage device (e.g., disk)
  - is permanent
  - can be used to provide input data or receive output data, or both
  - must reside in Project directory (not necessarily the same directory as the .cpp files)
  - must be opened before reading it

Using Input/Output Files

- ***stream*** - a sequence of characters
  - interactive (iostream)
    - **cin** - input stream associated with **keyboard**
    - **cout** - output stream associated with **display**
  - file (fstream)
    - **ifstream** - defines new input stream (normally associated with a file)
    - **ofstream** - defines new output stream (normally associated with a file)

Constructor

- Syntax

```
 fstream( const char *filename, openmode mode );
ifstream( const char *filename, openmode mode );
ofstream( const char *filename, openmode mode );
```

Mode	Meaning
ios::app	append output
ios::ate	seek to EOF when opened
ios::binary	open the file in binary mode
ios::in	open the file for reading
ios::out	open the file for writing
ios::trunc	overwrite the existing file

File-Related Functions

```
#include <fstream>
```

- **`foo.open(fname)`**
  - connects stream *`foo`* to the external file *`fname`*
- **`foo.get(ch)`**
  - Gets the next character from the input stream *`foo`* and places it in the character variable *`ch`*
- **`foo.put(ch)`**
  - Puts the character *`ch`* into the output stream *`foo`*
- **`foo.eof()`**
  - tests for end-of-file condition
- **`foo.close()`**
  - disconnects the stream and closes the file
- **`foo.is_open()`**
  - Tests whether the file is open

File-Related Functions

```
#include <fstream>
```

- **`foo.flush()`**
  - useful for printing out debugging information
  - sometimes programs abort before they have a chance to write their output buffers to the screen.
- **`foo.getline(char *buffer, streamsize num)`**
- **`foo.getline(char *buffer, streamsize num, char delim)`**
  - Reads the characters into buffer until

(1) num-1characters have been read

(2) A new line is encountered

(3) An EOF is encountered

(4) Until the character delim is read

Standard Input/Output Streams

- Stream is a sequence of characters
- Working with `cin` and `cout`
- Streams convert internal representations to character streams
- `>>` input operator (extractor)
- `<<` output operator (inserter)

Reading Data >>

- Leading white space skipped
- Newline character <nwln> also skipped
- Until first character is located

```
cin >> ch;
```

- Also read character plus white space as a character
  - `get` and `put` functions

File I/O

- Declare the stream to be processed:

```
#include <fstream>
ifstream ins;    // input stream
ofstream outs;   // output stream
```

- Need to open the files

```
ins.open(inFile);
outs.open(outFile);
```

<< and >>: Example 1

- You can read and write integers, doubles, chars, etc. from files just like `cin >>` and `cout <<` :

```
#include <iostream>
#include <fstream>
using namespace std;
void main(){
   ifstream fin;
   int A[4], r;
   fin.open("file1.dat");// read data file of four integers
   for(r=0; r<4; r++)   // into array
          fin >> A[r];
   fin.close();

   ofstream fout;
   fout.open("file2.dat"); // write data file
   for(r=3; r>=0; r--) // with numbers reversed
          fout << A[r] << ' ';
   fout.close();
 }
```

File I/O: Example 1

<u>`file1.dat:`</u>

`1 2 3 4`**`(eof)`**

<u>`file2.dat:`</u>

`4 3 2 1 `**`(eof)`**

File I/O: Example 2

```
// Copies indata.dat to outdata.dat
// and counts the number of lines.
// Prints file to screen too.
#include <iostream>
#include <fstream>
using namespace std;

void main(){
  ifstream ins;
  ofstream outs;
  int count=0;
  char next;

  ins.open("indata.dat");      // open the input file
  outs.open("outdata.dat");    // open the output file
```

File I/O: Example 2

```
  if( !ins.is_open()|| !outs.is_open()){//always check the following
      cerr << "file(s) cannot be open\n";
      exit( -1 );
  }
  while(true){        // loop for each line
      while(true){ // loop to read each char on line
             ins.get(next);
             if(ins.eof() || next== '\n')
                    break;
             cout << next;
             outs << next;
      }
      count++;
      cout << endl;
      if(ins.eof())
             break;
      outs << endl;
  }
  ins.close();
  outs.close();
  cout << "Number of lines copied: " << count << endl;
}
```

File I/O: Example 2

```
indata.dat:
a b c
top10 methods to count spaces

1   3(eof)

outdata.dat:
a b c
top10 methods to count spaces

1   3(eof)

Output to screen:
a b c
top10 methods to count spaces

1   3
Number of lines copied: 4
```

Another Example

- Program CopyFile.cpp demonstrates the use of the other `fstream` functions
  - `get`, `put`, `close` and `eof`
  - Copy from one file to another
- `#define` in the program associates the name of the stream with the actual file name
- `fail()` function - returns nonzero if file fails to open

CopyFile.cpp

Program Output

```
Input file copied to output file.
37 lines copied.
```

CopyFile.cpp (Header)

```
// File: CopyFile.cpp
// Copies file InData.txt to file OutData.txt

#include <cstdlib>
#include <fstream>

using namespace std;

// Associate stream objects with external file
// names
#define inFile "InData.txt"
#define outFile "OutData.txt"
```

CopyFile.cpp (Declarations)

```
// Functions used ...
// Copies one line of text
int copyLine(ifstream&, ofstream&);

int main()
{

    // Local data ...
    int lineCount;
    ifstream ins;
    ofstream outs;
```

CopyFile.cpp (Opening Input File)

```
// Open input and output file, exit on any
// error.
ins.open(inFile);
if (ins.fail ())
{
   cerr << "*** ERROR: Cannot open " <<
         inFile << " for input." << endl;
   return EXIT_FAILURE;    // failure return
}  // end if
```

CopyFile.cpp (Opening Output File)

```
outs.open(outFile);
if (outs.fail()) {
    cerr << "*** ERROR: Cannot open " << outFile
         << " for output." << endl;
    return EXIT_FAILURE; // failure return
}  // end if
```

CopyFile.cpp (Copy Line by Line)

```
    // Copy each character from inData to outData.
    lineCount = 0;
    do{
       if (copyLine(ins, outs) != 0)
          lineCount++;
    } while (!ins.eof());
    // Display a message on the screen.
    cout << "Input file copied to output file."
         << endl;
    cout << lineCount << " lines copied." << endl;
    ins.close();
    outs.close();
    return 0;          // successful return
}
```

CopyFile.cpp (copyLine procedure)

```
// Copy one line of text from one file to another
// Pre: ins is opened for input and outs for
// output.
// Post:     Next line of ins is written to outs.
//           The last character processed from
//           ins is <nwln>;
//           the last character written to outs
//           is <nwln>.
// Returns: The number of characters copied.
```

CopyFile.cpp (Character Reading)

```
int copyLine (ifstream& ins, ofstream& outs){
  // Local data ...
  const char NWLN = '\n';
  char nextCh;
  int charCount = 0;

  // Copy all data characters from stream ins to
  //    stream outs.
  ins.get(nextCh);
  while ((nextCh != NWLN) && !ins.eof()){
     outs.put(nextCh);
     charCount++;
     ins.get (nextCh);
  }  // end while
```

CopyFile.cpp (Detection of EOF)

```
    // If last character read was NWLN write it
    // to outs.
    if (!ins.eof())
    {
       outs.put(NWLN);
       charCount++;
    }
    return charCount;
}   // end copyLine
```

File I/O

- For each file stream, there is a 4-bit state flag, which contains:
  - badbit (unrecoverable error in the stream)
  - failbit (recoverable error in the stream)
  - eofbit (EOF reached)
  - goodbit (no error = none of above bits set)
- clear() is used to set the goodbit and clear other bits.
- fail() is to test if any error occurs (badbit or failbit is set), e.g., writing to an input file. Setting clear() will recover the case when failbit is set.
- bad() also tests the errors (badbit is set), but these are unrecoverable (i.e., calling clear() doesn't help). For example, writing to a file but there is no disk space.
- When you reach the end-of-file, the eofbit is set, that stops you to read the file anymore. So, you should call clear() first to reset that eofbit before the file can be read again.
  - Use `foo.seekg (0, ios::beg)` to go back to the beginning of the file for get
  - Use `foo.seekp (0, ios::beg)` to go back to the beginning of the file for put
  - Get length of a get file: `foo.seekg (0, ios::end); length = foo.tellg();` // analogous for a put file using `seekp()` and `tellp()`
- You can check these bits at any time in your program by calling rdstate().
- For details, you may refer to cplusplus.com

Passing IO streams to functions

```
// passing input and output files into function
void io_demo( istream & in, ostream & out ){

  int i, j;
  in >> i >> j;
  out << j << "  " << i << endl; //write j and i
}
int main(){

  ifstream ins;
  ofstream outs;
  ins.open( "in.txt" );
  outs.open( "out.txt" );

  io_demo( cin, cout );
  io_demo( ins, outs );

  return 0;
}
```

in.txt:
4 8
run
123 678

Outputs:
678 123
out.txt
8 4