Chapter 6

FUNCTIONS AND POINTERS

Chapter 6

- Function and parameter declarations
- Returning values
- Variable scope
- Variabe storage classes
- Pass-by-reference
- Recursion
- Passing arrays to functions
- Pointers
- The *typedef* declaration statement

Function and parameter declarations

- **User-defined program units are called *subprograms*. In C++ all subprograms are referred to as *internal functions*.**
  - A complex problem is often easier to solve by dividing it into several smaller parts, each of which can be solved by itself.
  - These parts are sometimes made into *functions*
  - main() : functions to solve the original problem
- ***Defining a Function:*** *data_type name_of_function (parameters){*
  statements;
  }
- **We could use also *external functions* (e.g., `abs`, `ceil`, `rand`, etc.) grouped into specialized libraries (e.g., `math`, etc.)**

- **A function definition consists of four parts:**
  - *A reserved word indicating the data type of the function's return value.*
  - *The function name*
  - *Any parameters required by the function, contained within (and).*
  - *The function's statements enclosed in curly braces { }.*

- **Example :**

```
int FindMax (int x, int y) {
    int maximum;
    if ( x > = y)
        maximum = x;
    else
            maximum = y;
    return maximum;
}
```

Advantages of function:

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

How to call functions

- **We have to designate a data type for function since it will return a value from a function after it executes.**
- **Variable names that will be used in the function header line are called *formal parameters*.**
- **To execute a function, you must invoke, or call it according to the *following syntax:***

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

- **The values or variables that you place within the parentheses of a function call statement are called *actual parameters*.**
  - Example: findMax(firstnum, secnum);

Function Prototypes

- **A *function prototype* declares to the compiler that we intend to use a function later in the program.**
- **If we try to call a function at any point in the program prior to its function prototype or function definition, we will receive an *error* at compile time.**
  - The lines that compose a function within a C++ program are called a *function definition*.
- **The function definition can be placed anywhere in the program after the function prototypes.**
- **If a function definition is placed in front of `main()`, there is no need to include its function prototype.**

Example:

```
// Finding the maximum of three integers
#include <iostream.h>

int maximum(int, int, int);   // function prototype
int main() {
   int a, b, c;
   cout << "Enter three integers: ";
   cin >> a >> b >> c;
   cout << "Maximum is: " << maximum (a, b, c) << endl;
   return 0;
}
// Function maximum definition
// x, y and z are parameters
int maximum( int x, int y, int z) {
   int max = x;
   if ( y > max )      max = y;
   if ( z > max )      max = z;
   return max;
}
```

The output of the above program:

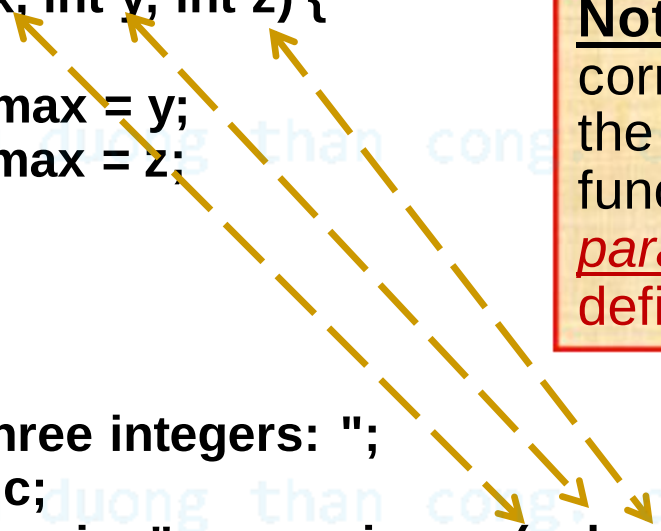
Enter three integers: 22 85 17
Maximum is: 85

- **Note**: it is possible to omit the **function prototype** if the **function definition** is placed before it is called.

Example:

```
// Finding the maximum of three integers
#include <iostream.h>

// Function maximum definition with parameters x, y and z
int maximum( int x, int y, int z) {
   int max = x;
   if ( y > max )     max = y;
   if ( z > max )     max = z;
   return max;
}
int main() {
   int a, b, c;
   cout << "Enter three integers: ";
   cin >> a >> b >> c;
   cout << "Maximum is: " << maximum (a, b, c) << endl;
   return 0;
}
```



Note: there is one-to-one correspondence between the *arguments* in a function call and the *parameters* in the function definition.

Passing by Value

- **If a variable is one of the actual parameters in a function call, the called function receives a *copy* of the values stored in the variable.**
- **After the values are passed to the called function, control is transferred to the called function.**
  - Example: The statement *findMax(firstnum, secnum);* calls the function *findMax()* and causes the values currently residing in the variables *firstnum* and *secnum* to be passed to *findMax()*.
- **The method of passing values to a called function is called *pass by value*.**

RETURNING VALUES

- To actually **return** a value to a **variable**, you must include the ***return*** statement within the **called function**.
- The syntax for the ***return*** statement is either

```
return value;
```

or

```
return(value);
```

- Values passes back and forth between **functions** must be of the same data type.

Inline function

- For small functions, you can use the *inline* keyword to request that the compiler replace calls to a function with the function definition wherever the function is called in a program.

Example:

```
// Using an inline function to calculate the volume of a cube.
#include <iostream.h>
inline double cube(double s) { return s * s * s; }
int main()
{
  cout << "Enter the side length of your cube:  ";
  double side;
  cin >> side;
  cout << "Volume of cube with side "
       << side << " is " << cube(side) << endl;
  return 0;
}
```

Function Overloading

- C++ enables several functions of the same name to be defined, as long as these functions have different sets of parameters (at least their types are different).
- This capability is called *function overloading*.
- When an overloaded function is called, the compiler selects the proper functions by examining the *number, types and order* of the arguments in the call.
- Function overloading is commonly used to create several functions of the same name that perform similar tasks but on different data types.

Example:

```
void showabs(int x)
{
   if( x < 0)
     x = -x;
     cout << “The absolute value of the integer is “ << x << endl;
}
void showabs(double x)
{
   if( x < 0)
     x = -x;
     cout << “The absolute value of the double is “ << x << endl;
}
```


If the function calls “showabs(10);”,
then it causes the compiler to use the *1st version* of the function *showabs*.

Default arguments

- **C++ allows *default arguments* in a function call.**
- **Default argument values are listed in the function prototype and are automatically passed to the called function when the corresponding arguments are omitted from the function call.**
  - *Example*: The function prototype

    void example (int, int = 5, float = 6.78);

  provides default values for the two last arguments.

- **If any of these arguments are omitted when the function is actually called, compiler supplies these default values.**

```
example(7,2,9.3);      // no default used
example(7,2);          // same as example(7, 2, 6.78)
example(7);            // same as example(7, 5, 6.78)
```

VARIABLE SCOPE

- **Recall**:
  - A sequence of statements within { … } is considered a block of code. The part of the program where you can use a certain identifier (variable or constant) is called the scope of that identifier. Scope refers to where in your program a declared identifier is allowed used.
  - The scope of an identifier starts immediately after its declaration and ends when the "innermost" block of code within which it is declared ends.
  - It is possible to declare the same identifier in another block within the program.
- *Global scope* refers to variables declared outside of any functions or classes and that are available to all parts of your program.
- *Local scope* refers to a variable declared inside a function and that is available only within the function in which it is declared.

Example:

```
#include <iostream.h>
int x;                  // create a global variable named firstnum
void valfun();          // function prototype (declaration)
int main()
{
  int y;                // create a local variable named secnum
  x = 10;               // store a value into the global variable
  y = 20;               // store a value into the local variable
  cout << "From main(): x = " << x << endl;
  cout << "From main(): y =  " << y << endl;
  valfun();             // call the function valfun()
  cout << "\nFrom main() again: x = " << x << endl;
  cout << "From main() again: y = " << y << endl;
  return 0;
}
```

```
void valfun() {
  int y;              // create a second local variable named y
  y = 30;             // this only affects this local variable's value
  cout << "\nFrom valfun(): x = " << x << endl;
  cout << "From valfun(): y = " << y << endl;
  x = 40;             // this changes x for both functions
  return;
}
```

The output of the above program:

From main(): x = 10
From main(): y = 20

From valfun(): x = 10
From valfun(): y = 30

From main() again: x = 40
From main() again: y = 20

Scope Resolution Operator

- When a **local variable** has the same name as a **global variable**, all uses of the **variable's name** within the **scope** of the **local variable** refer to the **local variable**.

- In such cases, we can still access to the **global variable** by using ***scope resolution operator*** (**::**) immediately before the **variable name**.

- The **::** operator tells the **compiler** to use the **global variable**.

Example :

```
#include <iostream.h>
float number = 42.8;        // a global variable named number
int main()
{
  float number = 26.4;      // a local variable named number
  cout << "The value of number is " << ::number << endl;
  return 0;
}
```

The output of the above program:

The value of number is 42.8

VARIABLE STORAGE CLASS

- The lifetime of a variable is referred to as the *storage duration*, or *storage class*.
- Four available storage classes: *auto*, *static*, *extern* and *register*.
- If one of these class names is used, it must be placed before the variable's data type in a declaration statement.

Examples:

```
auto int num;
static int miles;
register int dist;
extern float price;
extern float yld;
```

Local Variable Storage Classes

- Local variables can only be members of the ***auto***, ***static***, or ***register*** storage classes.
- Default: ***auto*** class.

Automatic Variables

- The term ***auto*** is short for automatic.
- ***Automatic storage duration*** refers to variables that exist only during the lifetime of the command block (such as a function) that contains them.

Example:

```
include <iostream.h>
void testauto();    // function prototype
int main(){
  int count;          // count is a local auto variable
  for(count = 1; count <= 3; count++)
      testauto();
  return 0;
}
void testauto(){
  int num = 0;     // num is a local auto variable
  cout << "The value of the automatic variable num is "
          << num << endl;
  num++;
  return;
}
```


The output of the above program:

The value of the automatic variable num is 0

The value of the automatic variable num is 0

The value of the automatic variable num is 0

Local static variables

- In some applications, we want a **function** to remember values between **function calls**. This is the purpose of the ***static*** storage class.
- A local ***static* variable** is not created and destroyed each time the **function** declaring the **static variable** is called.
- Once created, **local static variables** *remain in existence* for the life of the program.
- *However, locally declared **identifiers** cannot be accessed outside of the block they were declared in.*

Example:

```
#include <iostream.h>
int funct(int);    // function prototype
int main()
{
  int count, value;          // count is a local auto variable
  for(count = 1; count <= 10; count++){
     value = funct(count);
     cout << count << '\t' << value << endl;
  }return 0;
}

int funct( int x)
{
  int sum = 100;     // sum is a local auto variable
  sum += x;
  return sum;
}
```

The output of the above program:

```
1    101
2    102
3    103
4    104
5    105
6    106
7    107
8    108
9    109
10   110
```

Note: The effect of increasing *sum* in *funct()*, before the function's return statement, is lost when control is returned to *main()*.

Note:

A **local *static* variable** is not created and destroyed each time the **function** declaring the **static variable** is called. Once created, **local static variables** ***remain in existence*** for the life of the program.

Example :

```
#include <iostream.h>
int funct( int);    // function prototype
int main()
{
  int count, value;          // count is a local auto variable
  for(count = 1; count <= 10; count++){
     value = funct( count);
     cout << count << '\t' << value << endl;}
  return 0;
}
```

```
int funct( int x)
{
  static int sum = 100;     // sum is a local auto variable
  sum += x;
  return sum;
}
```

The output of the above program:

1	101
2	103
3	106
4	110
5	115
6	121
7	128
8	136
9	145
10	155

Note:

1. The initialization of ***static*** **variables** is done only once when the program is first compiled. At compile time, the **variable** is created and any initialization value is placed in it.
2. All **static variables** are set to zero when no explicit initialization is given.

Register Variables

- ***Register variables*** **have the same time duration as automatic variables.**
- **Register variables are stored in CPU's internal registers rather than in memory.**

Examples:

```
register int  time;
register double difference;
```

Global Variable Storage Classes

- **Global variables are created by definition statements external to a function.**
- **Once a global variable is created, it exists until the program in which it is declared is finished executing.**
- **Global variables may be declared as *static* or *extern* (but not both).**

External global variables

- **The purpose of the *external* storage class is to extend the scope of a global variable beyond its normal boundaries.**

External global variables

```
//file1
int price;
float yield;
static double coupon;
…
int main(){
    func1():
    func2():
    func3():
    func4():
}
int func1();
{
  …
}
int func2();
{
  …
}//end of file1
```

```
//file2
double interest;
int func3();
{
  .
  .
}
int func4();
{
  .
  .
}
//end of file2
```

Although the variable *price* has been declared in *file1*, we want to use it in *file2*. Placing the statement *extern int price* in *file2*, we can extend the scope of the variable *price* into *file2*.

```
//file1
int price;
float yield;
static double coupon;
.
.
int main(){
    func1():
    func2():
    func3():
    func4():
}
```

```
extern double interest;
int func1();
{
  .
  .
}
int func2();
{
  .
  .
}
//end of file1
```

```
//file2
double interest;
extern int price;
int func3();
{
  .
  .
}
int func4();
{
  extern float yield;
  .
  .
}
//end of file2
```

Note:

1. We cannot make external *static* variables.
2. The scope of a *global static* variable cannot extend beyond the file in which it is declared.

PASS BY REFERENCE

- **Reference Parameters**

  Two ways to invoke functions in many programming languages are:

  - **call by value**
  - **call by reference**

- **When an argument is passed *call by value*, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.**
- **With *call-by-reference*, the caller gives the called function the ability to access the caller's data directly, and to modify that data if the called function chooses so.**

- **To indicate that the function parameter is passed-by-reference, simply follow the parameter's type in the function prototype of function header by *an ampersand* (&).**

- **For example, the declaration**

  int& count

  in the function header means "*count* is a reference parameter to an *int*".

Example :

```
#include <iostream.h>
int squareByValue( int );
void squareByReference( int & );
int main()
{
   int x = 2, z = 4;
   cout << "x = " << x << " before squareByValue\n"
        << "Value returned by squareByValue: "
        << squareByValue( x ) << endl
        << "x = " << x << " after squareByValue\n" << endl;
```

```
  cout << "z = " << z << " before squareByReference" << endl;
   squareByReference( z );
   cout << "z = " << z << " after squareByReference" << endl;
   return 0;
}
int squareByValue( int a )
{
   return a *= a;   // caller's argument not modified
}
void squareByReference( int &cRef )
{
   cRef *= cRef;    // caller's argument modified
}
```

The output of the above program:

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByReference

z = 4 before squareByReference
z = 16 after squareByReference

RECURSION

- **In C++, it's possible for a function to call itself. Functions that do so are called *seft-referential* or *recursive functions*.**
- **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: Factorial**

1! = 1;

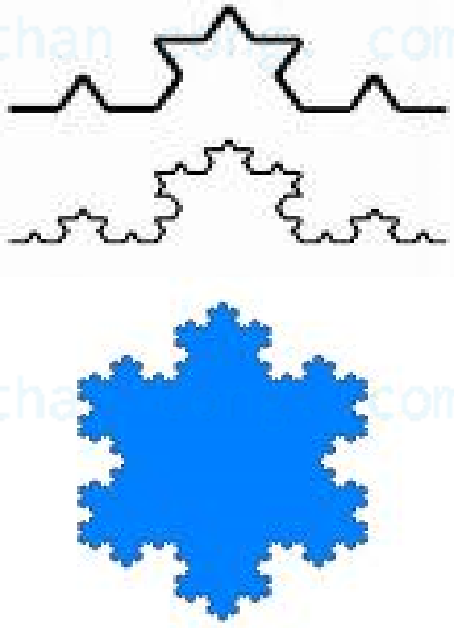
2! = 2*1 = 2*1!

3! = 3*2*1=3*2!

…. **n! = n*(n-1)!**

The factorial function is only defined for *positive* integers.

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



Example :

```
#include <iostream.h>
#include <iomanip.h>
unsigned long factorial( unsigned long );
int main(){
   for ( int i = 0; i <= 10; i++ )
      cout << setw( 2 ) << i << "! = "
           << factorial( i ) << endl;
   return 0;
}
// Recursive definition of function factorial
unsigned long factorial( unsigned long number ){
   if (number < 1)          // base case
      return 1;
   else                     // recursive case
      return number * factorial( number - 1 );
}
```

The output :

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

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.

- **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.**

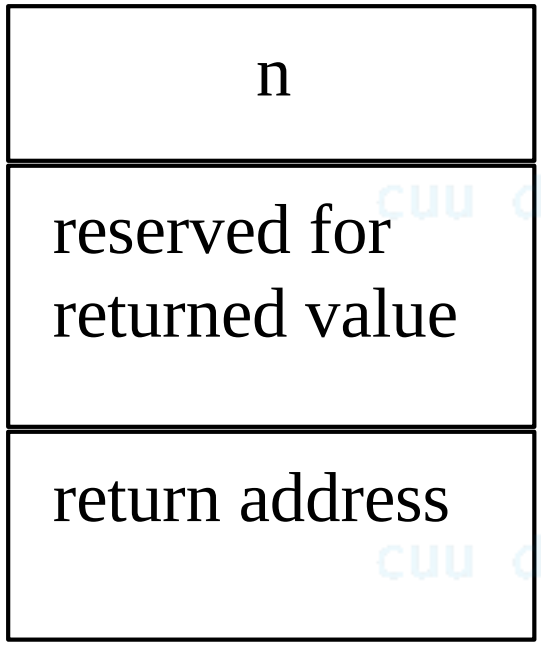
Example: The factorial function

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

How the Computation is Performed

- **The mechanism that makes it possible for a C++ function to call itself is that C++ allocates new memory locations for all function parameters and local variables as each function is called.**
- **There is a *dynamic data area* for each execution of a function. This allocation is made dynamically, as a program is executed, in a memory area referred as the stack.**
- **A *memory stack* is an area of memory used for rapidly storing and retrieving data areas for active functions.** Each function call reserves memory locations on the stack for its parameters, its local variables, a return value, and the address where execution is to resume in the calling program when the function has completed execution (*return address*).
- **Inserting and removing items from a stack are based on last-in/first-out mechanism.**

- **Thus, when the function call *factorial(n)* is made, a data area for the execution of this function call is pushed on top of the stack.**



The data area for the first call to factorial

- **The progress of execution for the recursive function *factorial* applied with n = 3 is as follows:**

factorial(3) = 3*factorial(2)
= 3*(2*factorial(1))
= 3*(2*(1*factorial(0)))
= 3*(2*(1*1))
= 3*(2*1)
= 3*2
= 6

Recursion

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!)

Recursion vs. Iterative structure

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

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

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!

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);
}
```

Or:

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

Oops!
No termination condition

Oops!

Recursion vs. Iteration

- **Iteration**
  - Uses repetition structures (`for`, `while` or `do…while`)
  - Repetition through explicitly use of repetition structure
  - Terminates when loop-continuation condition fails
  - Controls repetition by using a counter
- **Recursion**
  - Uses selection structures (`if`, `if…else` or `switch`)
  - Repetition through repeated method calls
  - Terminates when base case is satisfied
  - Controls repetition by dividing problem into simpler one

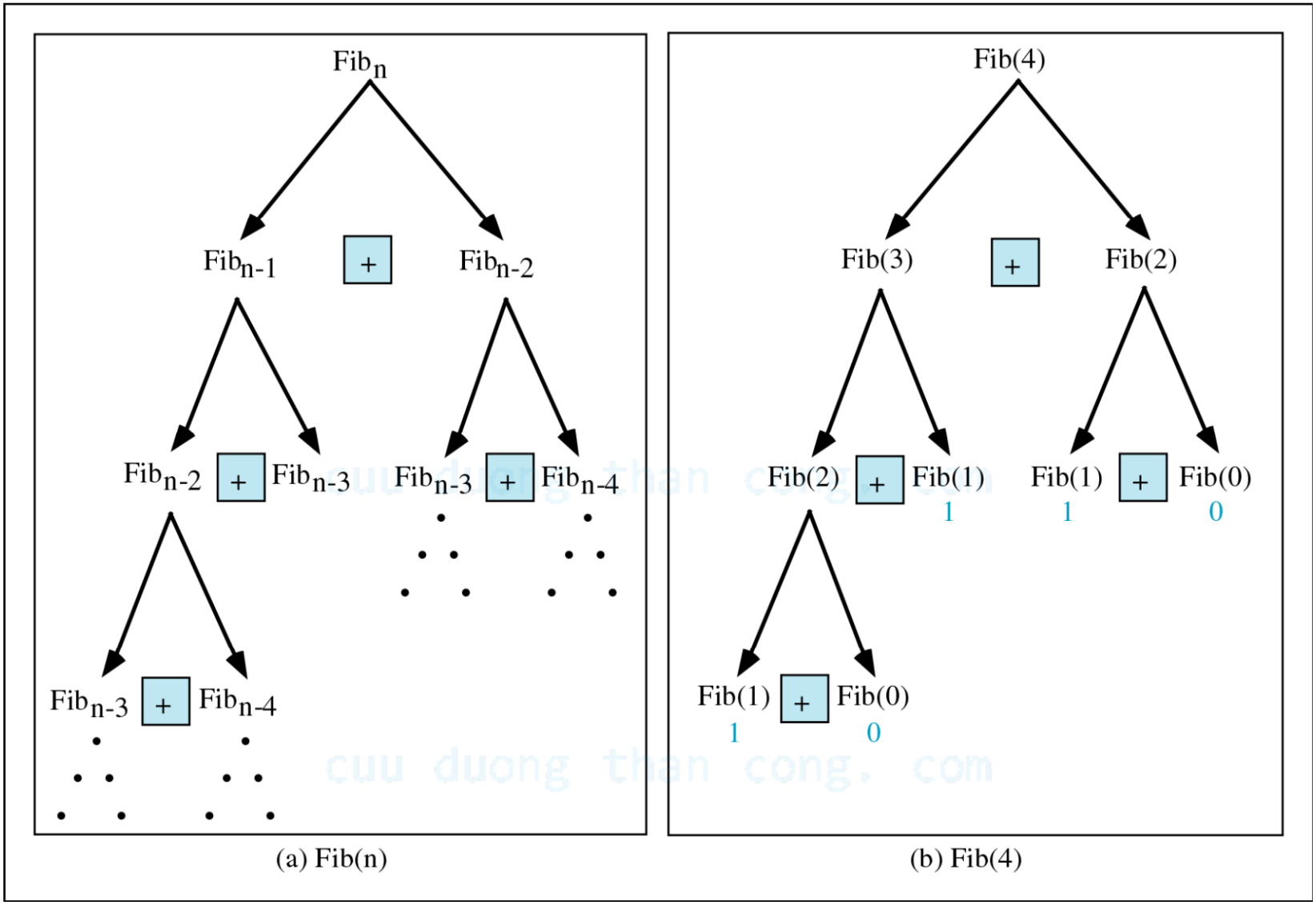
Example – Fibonacci numbers

- `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`
  where each number is the sum of the preceding two.
- Fibonacci numbers are believed to model nature to a certain extent, such as Kepler's observation of leaves and flowers in 1611.
- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(number) = F(number-1)+ F(number-2);`

$$F_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

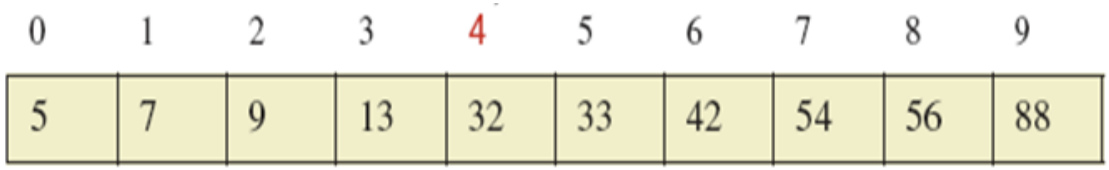
Example - Fibonacci numbers

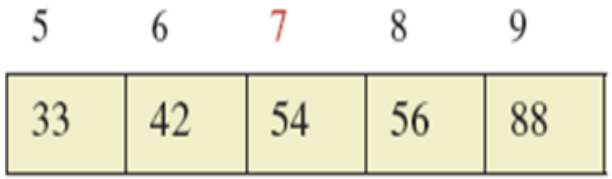
```
//Calculate Fibonacci numbers using recursive function.
//A very inefficient way, but illustrates recursion well
int fib(int number)
{
   if (number <= 0) return 0;
   if (number == 1) return 1;
   return (fib(number-1) + fib(number-2));
}
int main(){
   int inp_number;
   cout << "Please enter an integer: ";
   cin >> inp_number;
   cout << "The Fibonacci number for "<< inp_number
        << " is "<< fib(inp_number)<<endl;
  return 0;
}
```



Example - Binary Search

- Search for an element in an 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 would be.





Binary Search with Recursion

```
// Searches an ordered array of integers using recursion
int binSearch(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 binSearch(data, first, middle-1, value);
   else
      return binSearch(data, middle+1, last, value);
}
```

Binary Search

```
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    << binSearch(list,0,
            array_size-1,search_value) << endl;

    return 0;
}
```

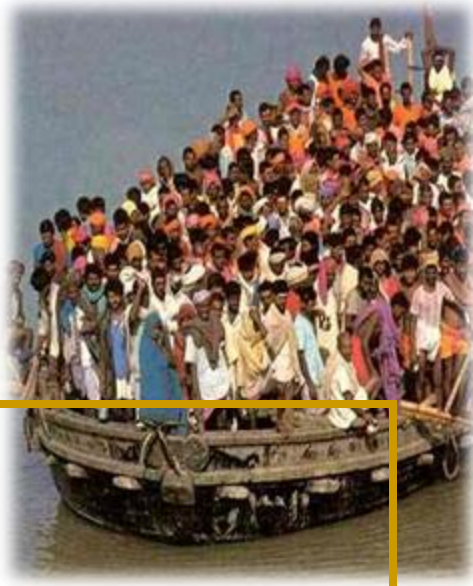
Binary Search w/o recursion

```
// Searches an ordered array of integers
int binSearch(const int data[], int size, int value
      int first, last, upper;
         first = 0;
         last = size - 1;
      while (true) {
          middle = (first + last) / 2;
          if (data[middle] == value)
               return middle;
          else if (first >= last)
               return -1;
          else if (value < data[middle])
            last = middle - 1;
          else
            first = middle + 1;
      }
}
```

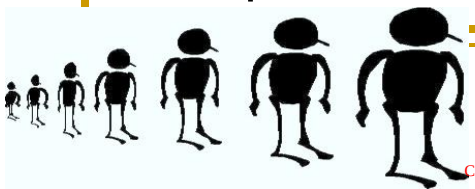
Exercise – Rabbit production

How many pairs of rabbits can be produced from a single pair in a year's time?

- **Assumptions:**
  - Each pair of rabbits produces a new pair of offspring every month;
  - each new pair becomes fertile at the age of one month;
  - none of the rabbits dies in that year.



- **Example:**
  - After 1 month there will be 2 pairs of rabbits;
  - after 2 months, there will be 3 pairs;
  - after 3 months, there will be 5 pairs (since the following month the original pair and the pair born during the first month will both produce a new pair and there will be 5 in all).



Exercise

PASSING ARRAYS TO FUNCTIONS

- To pass an array to a function, specify the name of the array without any brackets. For example, if array *hourlyTemperature* has been declared as

```
int hourlyTemperature[24];
```

- The function call statement

```
modifyArray(hourlyTemperature, size);
```

passes the array *hourlyTemperature* and its size to function *modifyArray*.

- For the function to receive an array through a function call, the function's parameter list must specify that an array will be received.

For example, the function header for function *modifyArray* might be written as

```
void modifyArray(int b[], int arraySize)
```

Notice that the size of the array is not required between the array brackets.

Example 6.7.2


```
#include<iostream.h>
int linearSearch( int [], int, int);
void main()
{
   const int arraySize = 100;
   int a[arraySize], searchkey, element;
```

```
   for (int x = 0; x < arraySize, x++)   // create some data
   a[x] = 2*x;
   cout<< "Enter integer search key: "<< endl;
   cin >> searchKey;
   element = linearSearch(a, searchKey, arraySize);
   if(element !=-1)
      cout<<"Found value in element "<< element << endl;
   else
      cout<< "Value not found " << endl;
}
int linearSearch(int array[], int key, int sizeofArray)
{
    for(int n = 0; n< sizeofArray; n++)
      if (array[n] = = key)
        return n;
    return –1;
}
```

POINTERS

- A ***pointer*** **is a special type of variable that stores the memory address of other variables.**
- **You declare a variable as a pointer by placing the *indirection operator* (*) after the data type or before the variable name.**

- **Examples:**

```
int *pFirstPtr;
int *pSecondPtr;
```

- **You use the *address-of operator* (&) to assign to the pointer variable the memory address of another variable.**

- **Example:**

```
double dPrimeInterest;
double *pPrimeInterest;
pPrimeInterest = &dPrimeInterest;
```

Note: If *ptr* is a pointer variable, ***ptr*** means the contents of the variable pointed to by *ptr*.

Example 6.8.1

```
#include<iostream.h>
void main()
{
   int a;
   int *aPtr;
      // aPtr is a pointer to an integer
   a = 7;
   aPtr = &a;    //aPtr set to address of a
   cout << "The address of a is " << &a
    << "\nThe value of aPtr is " << aPtr;
   cout << "\n\nThe value of a is "<< a
    << "\nThe value of *aPtr is " << *aPtr
    << endl;
}
```

The output of the above program:

The address of a is 0x0065FDF4

The value of aPtr is 0x0065FDF4

The value of a is 7

The value of *aPtr is 7

Calling Functions by Reference with Pointer Arguments

- **In C++, programmers can use pointers and the dereference operator to *simulate* call-by-reference.**
- **When calling a function with arguments that should be modified, the addresses of the arguments are passed.** This is normally achieved by applying the *address-of* operator (&) to the name of the variable whose value will be used.
- **A function receiving an address as an argument must define a pointer parameter to receive the address.**

Example 6.8.2

```
// Cube a variable using call-by-reference with a pointer argument
#include <iostream.h>
void cubeByReference( int * );   // prototype
int main()
{
   int number = 5;
   cout << "The original value of number is " << number;
   cubeByReference( &number );
   cout << "\nThe new value of number is " << number << endl;
   return 0;
}
 void cubeByReference( int *nPtr )
{
   *nPtr = (*nPtr) * (*nPtr) * (*nPtr);  // cube number in main
}
```

The output of the above propgram:

The original value of number is 5
The new value of number is 125

Pointers and Arrays

- **Notice that the name of an array by itself is equivalent to the *base address* of that array. That is, *the name z in isolation is equivalent to the expression &z[0]*.**

- **Example 6.8.3**

```
#include<iostream.h>
void main()
{
    int z[] = { 1, 2, 3, 4, 5};
    cout << "The value return by 'z' itself is the addr " << z << endl;
    cout << "The address of the 0th element of z is " << &z[0]  << endl;
}
```

The output of the above program:

The value return by 'z' itself is the addr 0x0065FDF4
The address of the 0th element of z is 0x0065FDF4

Accessing Array Element Using Pointer and Offset

- If we store the address of *grade[0]* into a pointer named *gPtr*, then the expression **gPtr* refers to *grade[0].*
- One unique feature of pointers is that *offset* may be included in pointer expression.
- For example, the expression **(gPtr + 3)* refers to the variable that is *three* (elements) beyond the variable pointed to by *gPtr*.
- The number 3 in the pointer expression is an *offset*. So *gPtr + 3* points to the element *grade[3]* of the *grade* array.

Example 6.8.4

```
#include <iostream.h>

int main()
{
   int b[] = { 10, 20, 30, 40 }, i, offset;
   int *bPtr = b;   // set bPtr to point to array b

   cout << "Array b printed with:\n"
        << "Array subscript notation\n";

   for ( i = 0; i < 4; i++ )
      cout << "b[" << i << "] = " << b[ i ] << '\n';

   cout << "\nPointer/offset notation\n";
   for ( offset = 0; offset < 4; offset++ )
      cout << "*(bPtr + " << offset << ") = "
           << *( bPtr + offset ) << '\n';
   return 0;
}
```

The output of the above program:

```
Array b printed with:
Array subscript notation
b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

Pointer/offset notation
*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40
```

Pointers and Strings

- **We can scan through a string by using pointer.**
- **The name of a string by itself is equivalent to the *base address* of that string.**

- **Example 6.8.5**

```
/* Printing a string one character at a time using pointer */
#include<iostream.h>
void main( )
{
   char strng[] = "Adams";
   char *sPtr;

   sPtr = &strng[0];
   cout << "\nThe string is: \n";
   for( ; *sPtr != '\0'; sPtr++)
        cout << *sPtr << ' ';
}
```

The output of the above program:

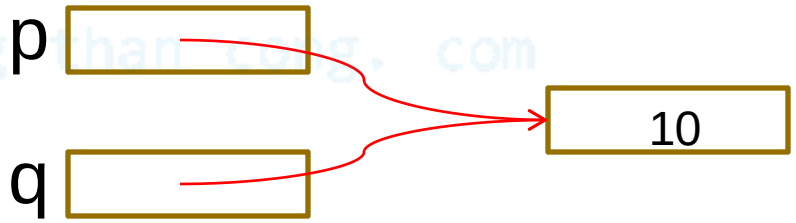
The string is:

A d a m s

Dynamically Allocating Storage

- **Besides variables, i.e. statically allocated and named storage, a dynamic storage can be dynamically allocated and be accessed through pointers**
- **Syntax:**
  - <pointer> = **new** <type>
  - <pointer> = **new** <type> [<number_of_elements>]
- **Example:**

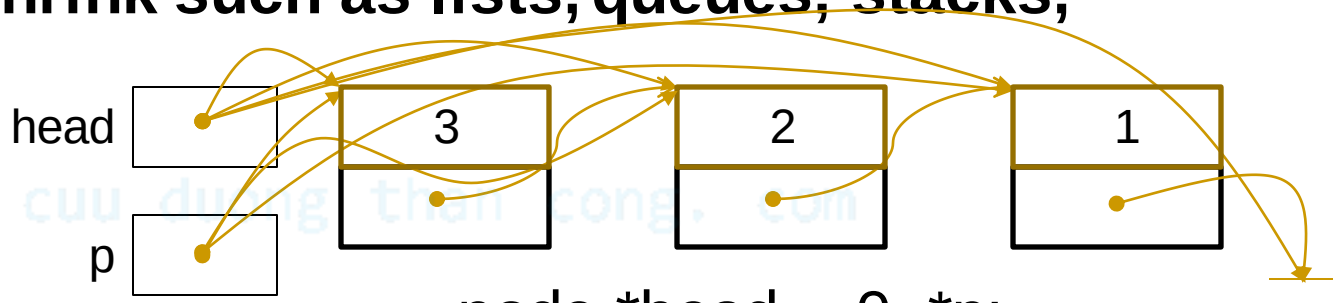
```
int *p,*q;
p = new int;
*p = 10;
q = p;
```



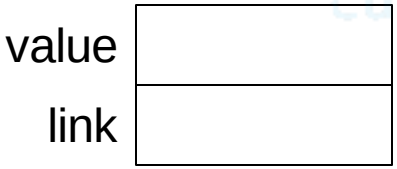
Dynamic data structure

- **Dynamic data structures are those that are able to grow and shrink such as lists, queues, stacks, trees,…**
- Example:

```
struct node {
    int value;
    struct node *link;
}
```




```
node *head = 0, *p;
for (int i = 1; i < 4; i++) {
        p = new node;
        p -> value = i;
        p -> link = head;
        head = p;
}
```



De-allocating dynamic storage

- **Dynamic storage should be de-allocated when it is not used any more; otherwise, it becomes garbage.**
- **Syntax:**
  - delete <pointer>
  - delete [] <pointer>
- **Example**

```
int *p;
p = new int;
…
delete p;
```

Exercise

Passing Structures as Parameters

- Complete copies of all members of a structure can be passed to a function by including the name of the structure as an argument to the called function.
- The parameter passing mechanism here is *call-by-value*.

Example 6.8.6

```
#include <iostream.h>
struct Employee      // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};
double calcNet(Employee);   // function prototype
int main()
{
```

```
  Employee emp = {6782, 8.93, 40.5};
  double netPay;
  netPay = calcNet(emp);       // pass by value
  cout << "The net pay for employee "
       << emp.idNum << " is $" << netPay << endl;
  return 0;
}
double calcNet(Employee temp) // temp is of data
                              // type Employee
{
  return (temp.payRate * temp.hours);
}
```

The output:

The net pay for employee 6782 is $361.665

Second way of passing a structure

- An alternative to the pass-by-value function call, we can pass a structure by *passing a pointer*.

Example 6.8.5

```
#include <iostream.h>
struct Employee  // declare a global type
{
  int idNum;
  double payRate;
  double hours;
};
double calcNet(Employee *); //function prototype
int main()
{
```

```
  Employee emp = {6782, 8.93, 40.5};
  double netPay;
  netPay = calcNet(&emp);      // pass an address
  cout << "The net pay for employee "
       << emp.idNum << " is $" << netPay << endl;
  return 0;
}

double calcNet(Employee* pt)   //pt is a pointer
{                              //to a structure of Employee type
  return (pt->payRate * pt->hours);
}
```

The output is:

The net pay for employee 6782 is $361.665

THE typedef DECLARATION STATEMENT

- **The *typedef* declaration permits us to construct alternate names for an existing C++ data type name. The syntax of a *typedef* statement:**

  typedef data-type new-type-name

- **For example, the statement:**

  typedef float REAL;

  make the name REAL a synonym for *float*. The name REAL can now be used in place of *float* anywhere in the program after the synonym has been declared.

  The definition

  REAL val;

  is equivalent to

  float val;

Example: Consider the following statement:

```
typedef struct
{
      char name[20];
      int idNum;
}  EMPREC;
```

The declaration EMPREC employee[75]; **is equivalent to**

```
struct
{
      char name[20];
      int idNum;
} employee[75];
```

Example: Consider the statement:

```
typedef  double*  DPTR;
```

The declaration: DPTR pointer1;

is equivalent to double* pointer1;