# ΗΥ-150
# Προγραμματισμός

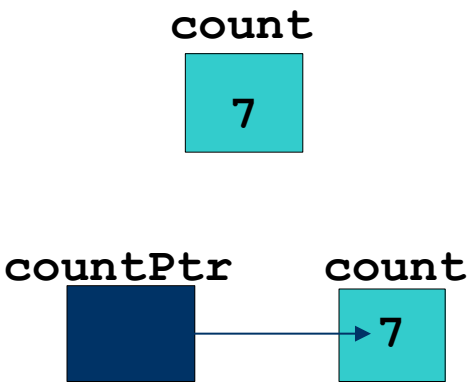
## Δείκτες (Pointers)

# Δείκτες

- Τι είναι:
  - τύπος μεταβλητών (όπως integer)
- Τι αποθηκεύουν:
  - την διεύθυνση στη μνήμη άλλων μεταβλητών
- Τι χρειάζονται:
  - Κυρίως, για δυναμική διαχείριση μνήμης και δυναμικές δομές δεδομένων
  - Call-by-reference
  - Στενή σχέση με τους πίνακες
- Πηγή πολλών λαθών που δύσκολα ανιχνεύονται

# Δείκτες

```
int Counter = 7;
int *PointerToCounter;

PointerToCounter = &Counter;
```




| Όνομα | Θέση Μνήμης | Τιμή |
|---|---|---|
| | ... | ... |
| Counter | 231000 | 7 |
| | ... | ... |
| PointerToCounter | 234000 | 231000 |

# Δήλωση Δεικτών:

- **Το *** χρησιμοποιείται για να δηλώσει δείκτη

  **int *myPtr;**

  – Ορίζει δείκτη σε **int** (pointer of type **int ***)

- Δήλωση πολλών μεταβλητών τύπου δείκτη απαιτεί τη χρήση ενός ***** μπροστά από κάθε μία:

  **int *myPtr1, *myPtr2;**

- Μπορούμε να δηλώσουμε δείκτη σε οποιοδήποτε είδος μεταβλητής (ακόμα και άλλο δείκτη)

  **double *myPtr2;**

  **char  **myPtr2;**

# Δήλωση Δεικτών:

- Χρήση του χαρακτήρα * για τη δήλωση

```
int variable;
   /* declares an int variable */
int * variable;
   /* declares a pointer to an int */
int variable[5];

int *variable;
int foo[5];
variable = foo;
int * variable[5];

   /* declares an array of 5 pointers, to 5 ints */
void * variable;
   /* pointer to anything */
```

# Τελεστές Δεικτών

- &μεταβλητή
  - q = &a;
  - Επέστρεψε την διεύθυνση στην οποία είναι αποθηκευμένη η μεταβλητή

- *δείκτης
- int a, *q;
  - a = *q;
  - Επέστρεψε τα περιεχόμενα της διεύθυνσης στην οποία δείχνει ο δείκτης

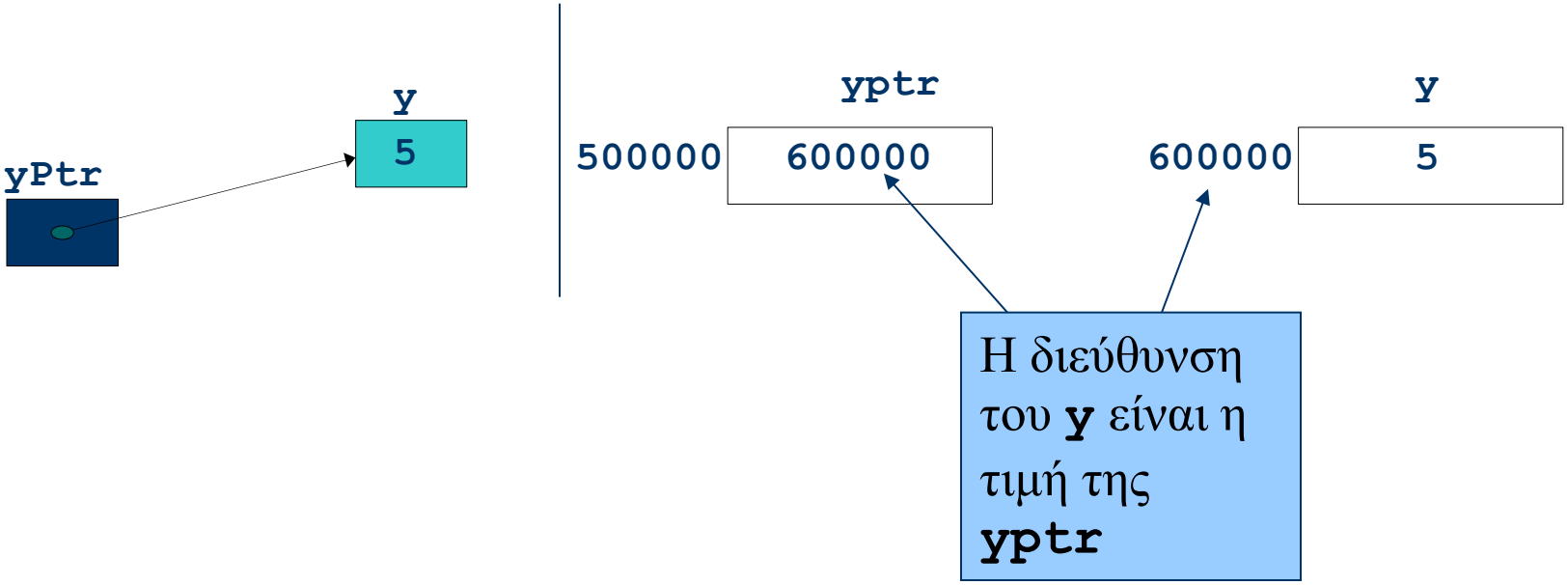
# Τελεστές Δεικτών

- **&** (address operator)

**int y = 5;**

**int *yPtr;**

**yPtr = &y; // Ο yPtr παίρνει την διεύθυνση y**



# Τελεστές Δεικτών

- * (indirection/dereferencing operator)

  **int y = 5;**

  **int *yPtr;**

  **yPtr = &y;**

  ***yPtr = *yPtr * 2;**

  - Μπορώ να αλλάζω την τιμή των μεταβλητών με τη βοήθεια δεικτών
  - Η τιμή του y γίνεται ίση με 10
  - Η μεταβλητή y αλλάζει τιμή, χωρίς στον κώδικα να φαίνεται αυτό άμεσα σε μια εντολή που εμφανίζει την y!
  - Η y δηλαδή, αποκτά ένα «ψευδώνυμο»: το *yPtr
- * και & είναι ανάποδα και αλληλοακυρώνονται

# Ανάθεση δεικτών

- Ένας δείκτης μπορεί να ανατεθεί σε άλλο δείκτη ίδιου τύπου:

```
int a = 4;
int * p;
p = &a;
int * q = p;
```

Η εντολή **int** * q = p; ορίζει το q ως δείκτη σε **int** και του δίνει αρχική τιμή το p. Τα q και p δείχνουν την ίδια θέση μνήμης και, άρα, *q και *p δείχνουν στην ίδια μεταβλητή (η a).

# Αρχικοποίηση δεικτών

- Ένας δείκτης που δεν του αποδοθεί αρχική τιμή δείχνει σε τυχαία περιοχή μνήμης. Η προσπέλαση στην τυχαία θέση μνήμης αυτή θα προκαλέσει σφάλμα εκτέλεσης, αν η συγκεκριμένη θέση δεν έχει δοθεί από το λειτουργικό σύστημα στο πρόγραμμά. Αν έχει δοθεί, και γράψουμε στη θέση μνήμης αυτή, θα "πανωγράψουμε" άλλη "δική μας" μεταβλητή χωρίς σφάλμα εκτέλεσης.
- Σε ένα οποιοδήποτε δείκτη μπορεί να γίνει απόδοση της τιμής 0 . Το 0 δεν αποτελεί αριθμό θέσης μνήμης και, επομένως, η ανάθεση αυτή υποδηλώνει ότι ο δείκτης δε δείχνει σε κανένα αντικείμενο. Σε τέτοιο δείκτη (null pointer) η δράση του (*) δεν επιτρέπεται (προκαλεί λάθος κατά την εκτέλεση αλλά όχι κατά τη μεταγλώττιση του προγράμματος).
- Η σύγκριση ενός άγνωστου δείκτη (π.χ. όρισμα συνάρτησης) με το 0 πρέπει να προηγείται οποιασδήποτε απόπειρας προσπέλασης της θέσης μνήμης στην οποία θεωρούμε ότι δείχνει. **Προσέξτε ότι σε αυτό το σημείο υπάρχει μία βασική διαφορά με την αναφορά: ένας δείκτης μπορεί να μη συνδέεται με κανένα αντικείμενο ενώ, αντίθετα, μία αναφορά είναι οπωσδήποτε συνδεδεμένη με κάποια ποσότητα**.

```
1  /*
2     Using the & and * operators */
3  #include <iostream>
4  using namespace std;
5  int main()
6  {
7     int a;          /* a is an integer */
8     int *aPtr;      /* aPtr is a pointer to an integer */
9
10    a = 7;
11    aPtr = &a;      /* aPtr set to address of a */
12
13    cout << "The address of a is " << &a << endl <<
14            "The value of aPtr is " << aPtr ;
15
16    cout << endl << endl << "The value of a is " << a << endl <<
17            "The value of *aPtr is " << *aPtr;
18
19    cout << endl << endl << "Showing that * and & are inverses " <<
20            "of each other.\n&*aPtr = " << *&a << endl <<
21            "*&aPtr = " << *&aPtr ;
22
23    return 0;
24 }
```

```
The address of a is 0012FF88
The value of aPtr is 0012FF88

The value of a is 7
The value of *aPtr is 7
Proving that * and & are complements of each other.
&*aPtr = 0012FF88
*&aPtr = 0012FF88
```

The address of **a** is the value of **aPtr**.

The ***** operator returns an alias to what its operand points to. **aPtr** points to **a**, so ***aPtr** returns **a**.

Notice how ***** and **&** are inverses

# Sort με χρήση της Swap

```
void swap( int *n1, int *n2)
{
      int temp;

      temp = *n1;
      *n1 = *n2;
      *n2 = temp;
}

int x = 10, y=20;

swap(&x,&y);
```

```
1  /* Fig. 7.15: fig07_15.cpp
2     This program puts values into an array, sorts the values into
3     ascending order, and prints the resulting array. */
4  #include <iostream>
5  #define SIZE 10
6  void bubbleSort( int *, const int );
7  void swap( int *, int * );
8  int main()
9  {
10
11    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
12    int i;
13
14    cout << "Data items in original order" << endl;
15
16    for ( i = 0; i < SIZE; i++ )
17       printf( "%4d", a[ i ] );
18
19    bubbleSort( a, SIZE );          /* sort the array */
20    cout << "Data items in ascending order" << endl;
21
22    for ( i = 0; i < SIZE; i++ )
23       cout << a[ i ] << " " );
24
25    cout << endl;
26
27    return 0;
28 }
29
30 void bubbleSort( int *array, const int size )
31 {
32
```

**sort** gets passed the address of array elements (pointers). The name of an array is a pointer.

```
33    int pass, j;
34    for ( pass = 0; pass < size - 1; pass++ )
35
36       for ( j = 0; j < size - 1; j++ )
37
38          if ( array[ j ] > array[ j + 1 ] )
39             swap( &array[ j ], &array[ j + 1 ] );
40 }
41
42 void swap( int *element1Ptr, int *element2Ptr )
43 {
44    int hold = *element1Ptr;
45    *element1Ptr = *element2Ptr;
46    *element2Ptr = hold;
47 }
```

```
Data items in original order
   2    6    4    8   10   12   89   68   45   37
Data items in ascending order
   2    4    6    8   10   12   37   45
```

```
1  /*
2     Cube a variable using call
3     with a pointer argument */
4
5  #include <iostream>
6  using namespace std;
7  void cubeByReference( int * );    /* p
8
9  int main()
10 {
11    int number = 5; int *ptr = &number;
12
13    cout << "The original value of number is " << number << endl;
14    cubeByReference( ptr );
15    cout << "The new value of number is " << number << endl);
16
17    return 0;
18 }
19
20 void cubeByReference( int *nPtr )
21 {
22    *nPtr = *nPtr * *nPtr * *nPtr;  /* cube number in main */
23 }
```

Notice that the function prototype takes a pointer to an integer (**`int *`**).

Notice how the address of **`number`** is given - **`cubeByReference`** expects a pointer (an address of a variable).

Inside **`cubeByReference`**, **`*nPtr`** is used (**`*nPtr`** is **`number`**).

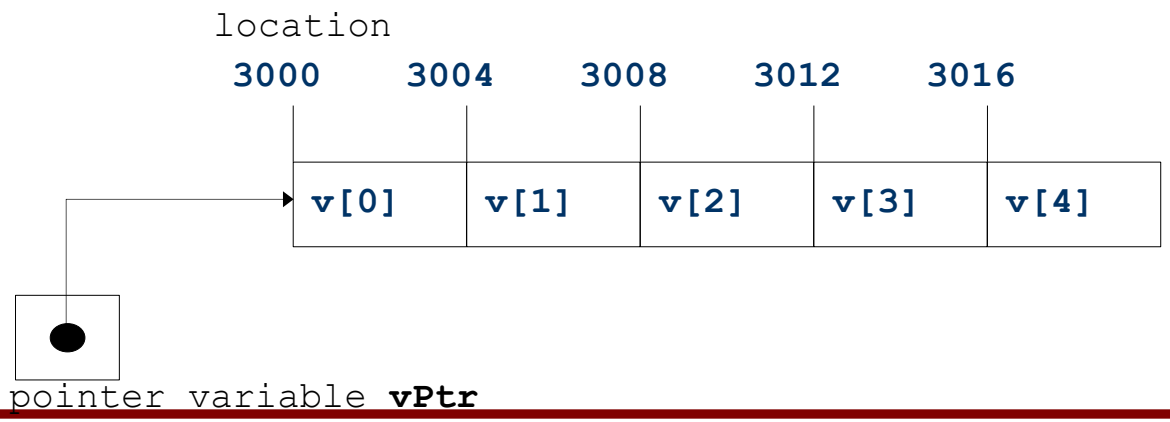
```
The original value of number is 5
The new value of number is 125
```

# Αριθμητική Δεικτών

- Στους δείκτες μπορούμε να πραγματοποιούμε αριθμητικές πράξεις:
  - **++** ή **–-**
    - Ο δείκτης δείχνει στο επόμενο ή προηγούμενο **αντικείμενο**, δηλ. αν είναι δείκτης σε ακέραιο στον επόμενο ή προηγούμενο ακέραιο
  - Αντίστοιχα **για + ή += , - ή -=**
  - Οι δείκτες μπορούν να αφαιρούνται ο ένας από τον άλλο
  - Όλες αυτές οι πράξεις δεν έχουν νόημα εκτός αν πραγματοποιούνται πάνω δείκτη που δείχνει στα στοιχεία ενός πίνακα

# Αριθμητική Δεικτών

- Πίνακας 5 στοιχείων τύπου **int** σε μηχανή με 4 byte **int**s
  - **int v[5];**
  - **int *vPtr;**
    - **vPtr** = &v[0]  ή **vPtr** = v; //δείχνει στο πρώτο στοιχείο **v[0]**
      //**vPtr** στο **3000**
  - **vPtr += 2;** //**vPtr** στο **3008**
    - **vPtr** δείχνει στο **v[ 2 ]** (αύξηση κατά 2), μια και η μηχανή έχει 4 byte **int**s, οπότε δείχνει **3008**



# Σχέση μεταξύ πινάκων και δεικτών

- Στοιχείο **b[ 3 ]**
  - Μπορούμε να το πάρουμε σαν ***( bPtr + 3 )**
  - Σαν **bPtr[ 3 ]**
    - **bPtr[ 3 ]** == **b[ 3 ]**
  - Με αριθμητική δεικτών:

    ***( b + 3 )**

# Διεύθυνση Πινάκων και Δείκτες

- int *a;
- a δείκτης σε ακέραιο
- Δεσμεύεται μνήμη μόνο για την αποθήκευση του a (της διεύθυνσης)
- Μπορούμε να αλλάξουμε την διεύθυνση που δείχνει ο a

- int b[10];
- b σταθερή διεύθυνση (δείκτης) σε ακέραιο
- Δεσμεύεται μνήμη για 10 ακέραιους
- Δεν μπορούμε να αλλάξουμε την διεύθυνση στην οποία αντιστοιχεί ο b

# Πίνακες και Δείκτες

- Μπορούμε να περνάμε δείκτες σε συναρτήσεις που περιμένουν πίνακες και αντίστροφα. Σε κάθε περίπτωση περνάμε απλώς μία διεύθυνση μνήμης. Το παρακάτω γίνεται δεκτό:

```
void f(int a[]);
void q(int *b);
int main()
{
   int *c;
   int d[10];
   f(c);
   q(d)
}
```

# Παράδειγμα - ptrarithm2.c (I)

```
int n[10] = {2, 3, 4, 5, 6, 7, 8, 9, 10, 1}, *p, c;

// First way to traverse the array
cout << "First way to index the array.“ << endl;
for (c=0; c < 10; c++)
      cout << "n[“ << c << ”] = “ << n[c] << endl;


cout << “Second way to index the array, through pointer arithmetic.“ << endl;
for (c=0; c < 10; c++)
      cout << "n[“ << c << ”] = “ << *(n+c);
```

# Παράδειγμα - ptrarithm2.c (II)

```
cout << “Third way to index the array, through pointers
arithmetic.“ << endl;
// We set the pointer to the beginning of the array
p = n;
for (c=0; c < 10; c++)
        cout << "n[“<< c << ”] = “ << *(p+c);

cout  <<“Fourth way to index the array, through
pointers arithmetic. “ << endl;
p = n;
for (c=0 ; c < 10; c++)
        cout << "n[“ << c << ”] = “ << *(p++)
        << “ ” << p;
```

# Σύγκριση pointers

- Είναι δυνατή ανεξαρτήτων που δείχνουν. Έχει νόημα όταν δείχνουν σε στοιχεία πίνακα.
- Γίνεται με τους τελεστές <, >, =, <= and >=
- Σύγκριση άσχετων μεταξύ τους pointers είναι απρόβλεπτη.

```
int data[100], *p1, *p2;
for (int i = 0; i <100;i++)
            data[i] = i;
p1 = &data [1];
p2 = &data [2];
if (p1 > p2)
            cout<<"\n\n p1 is greater than p2";
else
            cout<<"\n\n p2 is greater than p1";
```

# Μετατροπή Τύπων και Δείκτες

- Ο τύπος του αντικειμένου στο οποίο δείχνει ο δείκτης χρειάζεται για την σωστή αριθμητική δεικτών

- Όλοι οι δείκτες, ανεξάρτητα του αντικειμένου που δείχνουν, απαιτούν την ίδια μνήμη για την αποθήκευσή τους

- Η μετατροπή τύπων από ένα τύπο δείκτη σε άλλο τύπο δείκτη δεν αλλάζει τα περιεχόμενα του δείκτη:
  - int *aPtr;
  - float *bPtr;
  - aPtr = (int *)bPtr;

# Παραδείγματα – types.c

```
int main()
{
    int *a;
    double *b;
    float *c;
    float (*d)[10];
    float da[10];

    cout << "Size of a = “ << sizeof(a) << endl;
    cout << "Size of b = “ << sizeof(b) << endl; cout << "Size of c = “ <<
        sizeof(c) << endl;
    cout << "Size of d = “ << sizeof(d) << endl; cout << "Size of *d = “ <<
        sizeof(*d) << endl;
    cout << "Size of da = “ << sizeof(da) << endl;
    return 0;

}
```

# Παραδείγματα

```
int main()
{
    int *a,n,array[10];
    double *b;

    // Doing pointer arithmetic on a as an int *
    a = array;
    for (n=0; n < 10; n++)
        cout << "Address of a is “ << a++;

    cout << endl;

    b = (double *)array;
    for (n=0; n < 10; n++)
        cout << "Address of b is “ << b++;
    return 0;
}
```

- int A[10];
- int N;
- int * dynamicArray ;
- dynamicArray = A;
- cin >> N;
- dynamicArray = new int [N] ;

- delete [] N;

# Παράδειγμα 2Δ πίνακα - C++

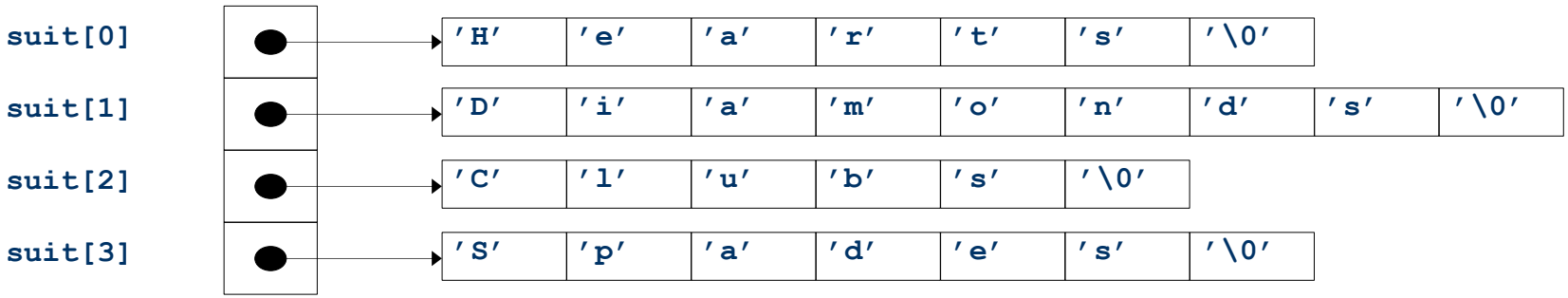
```
int **dynamicArray = 0;

//memory allocated for elements of rows.
dynamicArray = new int *[ROWS] ;
//memory allocated for elements of each column.

for( int i = 0 ; i < ROWS ; i++ )
        dynamicArray[i] = new int[COLUMNS];

//free the allocated memory
for( int i = 0 ; i < ROWS ; i++ )
        delete [] dynamicArray[i] ;
delete [] dynamicArray ;
```

# Πίνακες από δείκτες

- Ένας πίνακας μπορεί να περιέχει και δείκτες
- Παράδειγμα: ένας πίνακας από strings

  **char *suit[ 4 ] = { "Hearts", "Diamonds",**
  **"Clubs", "Spades" };**

  - Περιέχει δείκτες στον 1ο κάθε φορά χαρακτήρα
  - **char *** – κάθε στοιχείο του **suit** είναι δείκτης σε **char**
  - Τα strings στην πραγματικότητα δεν αποθηκεύονται στον πίνακα **suit**, μόνο δείκτες αποθηκεύονται στον πίνακα




  - **Το suit** έχει σταθερό μέγεθος (ως πίνακας άλλωστε), αλλά τα αλφαριθμητικά μπορούν να έχουν όποιο μέγεθος επιθυμούμε

# Παράδειγμα: Ανακάτεμα και μοίρασμα τράπουλας

```c


#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void shuffle( int [][ 13 ] );
void deal( const int [][ 13 ], const char *[], const char *[] );

int main()
{
   const char *suit[ 4 ] =
      { "Hearts", "Diamonds", "Clubs", "Spades" };
   const char *face[ 13 ] =
      { "Ace", "Deuce", "Three", "Four",
        "Five", "Six", "Seven", "Eight",
        "Nine", "Ten", "Jack", "Queen", "King" };
   int deck[ 4 ][ 13 ] = { 0 };

   srand( time( 0 ) );

   shuffle( deck );
   deal( deck, face, suit );

   return 0;
}

void shuffle( int wDeck[][ 13 ] )
{
   int row, column, card;

   for ( card = 1; card <= 52; card++ ) {
```

```
33          do {
34             row = rand() % 4;
35             column = rand() % 13;
36          } while( wDeck[ row ][ column ] != 0 );
37
38          wDeck[ row ][ column ] = card;
39       }
40    }
41
42    void deal( const int wDeck[][ 13 ], const char *wFace[],
43               const char *wSuit[] )
44    {
45       int card, row, column;
46
47       for ( card = 1; card <= 52; card++ )
48
49          for ( row = 0; row <= 3; row++ )
50
51             for ( column = 0; column <= 12; column++ )
52
53                if ( wDeck[ row ][ column ] == card )
54                   printf( "%5s of %-8s%c",
55                   wFace[ column ], wSuit[ row ],
56                   card % 2 == 0 ? '\n' : '\t' );
```

The numbers 1-52 are randomly placed into the **deck** array showing the position of card wDeck[row][column].

Searches **deck** for the **card** number, then prints the **face** and **suit**.

```
  Six of Clubs          Seven of Diamonds
  Ace of Spades           Ace of Diamonds
  Ace of Hearts         Queen of Diamonds
Queen of Clubs          Seven of Hearts
  Ten of Hearts         Deuce of Clubs
  Ten of Spades         Three of Spades
  Ten of Diamonds        Four of Spades
 Four of Diamonds         Ten of Clubs
  Six of Diamonds         Six of Spades
Eight of Hearts         Three of Diamonds
 Nine of Hearts         Three of Hearts
Deuce of Spades           Six of Hearts
 Five of Clubs          Eight of Clubs
Deuce of Diamonds       Eight of Spades
 Five of Spades          King of Clubs
 King of Diamonds        Jack of Spades
Deuce of Hearts         Queen of Hearts
  Ace of Clubs           King of Spades
Three of Clubs           King of Hearts
 Nine of Clubs           Nine of Spades
 Four of Hearts         Queen of Spades
Eight of Diamonds        Nine of Diamonds
 Jack of Diamonds       Seven of Clubs
 Five of Hearts          Five of Diamonds
 Four of Clubs           Jack of Hearts
 Jack of Clubs          Seven of Spades
```

# Δείκτες σε συναρτήσεις

- Δείκτης σε συνάρτηση
  - Περιέχει τη διεύθυνση του κώδικα της συνάρτησης
  - Όπως ένας δείκτης δείχνει στο 1ο στοιχείο ενός πίνακα
- Γιατί;
  - Πολλές φορές δεν ξέρουμε τη συνάρτηση θέλουμε να καλέσουμε, μέχρι να τρέξουμε το πρόγραμμα (π.χ., αν η συνάρτηση που πρέπει να καλέσουμε καθορίζεται από την είσοδο).
  - Πολλές φορές προγραμματίζουμε ή χρησιμοποιούμε συναρτήσεις βιβλιοθήκης οι οποίες χρησιμοποιούν άλλες υποσυναρτήσεις. Οι τελευταίες μπορεί να προμηθεύονται από τον χρήστη της βιβλιοθήκης, ώστε να αλλάζουν την συμπεριφορά της συνάρτησης, χωρίς να χρειάζεται να την ξαναμεταγλωττίσουμε.
  - Απαραίτητες πολλές φορές για κατανοητό, ευανάγνωστο ή γρήγορο κώδικα. Αποφεύγουμε τις πολλές if/switch

# Δείκτες σε συναρτήσεις

- Οι δείκτες σε συναρτήσεις
  - Τα περνάμε σαν ορίσματα σε συναρτήσεις
  - Αποθηκεύονται σε πίνακες
  - Εκχωρούνται σε άλλους δείκτες
  - **Διαφορά 1 από τους υπόλοιπους δείκτες**: «πρόσβαση» στα περιεχόμενά τους: *functionPtr
    - (*functionPtr)(argument1, argument2)
    - Σε αυτήν την περίπτωση καλείται η συνάρτηση της οποίας η διεύθυνση είναι αποθηκευμένη στον δείκτη
  - **Διαφορά 2 από τους υπόλοιπους δείκτες:** ανάθεση διεύθυνσης μιας συνάρτησης και όχι μεταβλητής
    - functionPtr = &afunction;

# Δήλωση Δεικτών σε Συναρτήσεις

- int (*funPtr)(int, int)
  - Ο τύπος της μεταβλητής funPtr είναι «δείκτης σε συνάρτηση που παίρνει δύο ορίσματα τύπου int, και επιστρέφει int.
- int *funPtr(int, int)
  - Αυτό θα σήμαινε, ότι το funPtr είναι τύπου «συνάρτηση που παίρνει δύο ορίσματα τύπου int, και επιστρέφει δείκτη σε int)
- int (*funPtrarray[10])(int, int))
  - Τυπος του funPtrarray: πίνακας από 10 δείκτες σε συναρτήσεις που παίρνουν δύο ορίσματα τύπου int και επιστρέφουν int

# Παραδείγματα

```
int function(int a, int b)
{
      return a*b;
}
int main()
{
    int (*funPtr)(int, int);
    funPtr = &function;
    cout << “Calling the function returns” << (*funPtr)(4, 5) << endl;
    return 0;
}
```

Δήλωση του δείκτη σε συνάρτηση

Ανάθεση της διεύθυνσης μιας συνάρτησης

Κλήση της συνάρτησης που είναι αποθηκευμένη στον δείκτη

# Παράδειγμα

- Θυμηθείτε ότι η BubbleSort ταξινομεί ακεραίους βασιζόμενη σε συγκρίσεις μεταξύ τους.

- Αλλάξτε την BubbleSort ώστε να παίρνει ως όρισμα την συνάρτηση σύγκρισης δύο ακεραίων και να την χρησιμοποιεί για ταξινόμηση. Καλέστε την BubbleSort έτσι ώστε να ταξινομεί ακεραίους και κατά αύξοντα σειρά και κατά φθίνουσα

```
1
2
3  #include <stdio.h>
4  #define SIZE 10
5  void bubble( int [], const int, int (*)( int, int ) );
6  int ascending( int, int );
7  int descending( int, int );
8
9  int main()
10 {
11
12    int order,
13        counter,
14        a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
15
16    cout << "Enter 1 to sort in ascending order, " <<
17            "Enter 2 to sort in descending order: ";
18    cin >> order ;
19    cout << endl << "Data items in original order\n" ;
20
21    for ( counter = 0; counter < SIZE; counter++ )
22       cout << a[ counter ];
23
24    if ( order == 1 ) {
25       bubble( a, SIZE, &ascending );
26       cout << endl << "Data items in ascending order" << endl ;
27    }
28    else {
29       bubble( a, SIZE, &descending );
30       cout << endl << "Data items in descending order\n";
31    }
32
```

Notice the function pointer parameter.

```
33    for ( counter = 0; counter < SIZE; counter++ )
34       cout << a[ counter ] ;
35
36    cout << endl;
37
38    return 0;
39 }
40
41 void bubble( int work[], const int size,
42               int (*compare)( int, int ) )
43 {
44    int pass, count;
45
46    void swap( int *, int * );
47
48    for ( pass = 1; pass < size; pass++ )
49
50       for ( count = 0; count < size - 1; count++ )
51
52          if ( (*compare)( work[ count ], work[ count + 1 ] ) )
53             swap( &work[ count ], &work[ count + 1 ] );
54 }
55
56 void swap( int *element1Ptr, int *element2Ptr )
57 {
58    int temp;
59
60    temp = *element1Ptr;
61    *element1Ptr = *element2Ptr;
62    *element2Ptr = temp;
63 }
64
```

**ascending** and **descending** return **true** or **false**. **bubble** calls **swap** if the function call returns **true**.

Notice how function pointers are called using the dereferencing operator. The * is not required, but emphasizes that **compare** is a function pointer and not a function.

```
65 int ascending( int a, int b )
66 {
67    return b < a;   /* swap if b is less than a */
68 }
69
70 int descending( int a, int b )
71 {
72    return b > a;   /* swap if b is greater than a */
          73        }
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
   2    6    4    8   10   12   89   68   45   37
Data items in ascending order
   2    4    6    8   10   12   37   45   68   89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
   2    6    4    8   10   12   89   68   45   37
Data items in descending order
  89   68   45   37   12   10    8    6    4    2
```