Difference between Pass-by-Reference and Pass-by-Value-Result:

1. Pass-by-value-result is more efficient in partitioned memory where a Pass-by-Reference could refer to a distant segment or page, causing a page fault. Similarly if memory is accessible through a network, it would necessitate a delay for network access.

2. Concurrent execution could give unpredictable results. Using Pass-by-Reference could end up giving values over some limit (see ticket sellers race-condition example). Using Pass-by-Value-Result could end up with an invalid value, but it would always be less than the limit.

3. Consider also the following example, where Pass-by-Value-Result differs from Pass-by-Reference:

```plaintext
PROGRAM
    VAR i,j: INTEGER;

    PROCEDURE foo(x,y)
    BEGIN
        i:=y
    END;

    BEGIN
        i:= 2; j:= 3;
        foo( i,j)
    END
```

In Pass-by-Value-Result, x is a synonym for i on the call, but the value of x is never changed, so the changed value of i is restored to the original value when returning from the function.

In Pass-by-Reference the change to i is kept back in the main program.
Adapted from March 2001 Linux magazine article on Threads and Mutexes.

Consider the following function running as a thread, for a ticket agent to sell tickets:

```c
void ticket_agent (int *total_sold)
{
    while (*total_sold < NUM_TICKETS) {
        if (sold_ticket())
            (*total_sold)++;       //retrieve and increment
    }
}
```

Assume we have the above function in a program where we also have:

```c
#define NUM_TICKETS 10000000
#define NUM_AGENTS 4
int total_sold = 0;
```

In the main program, NUM_AGENTS threads are created, which can run concurrently, selling tickets. It turns out that the program over sells tickets. Running the program 10,000 times gives the following oversold statistics:

- Over sold 0 exactly 3924 times.
- Over sold 1 exactly 4735 times.
- Over sold 2 exactly 1341 times.
- Over sold 3 exactly 0 times.

So the program only works correctly about 40% of the time. *Why is this?*

*Answer:*
Remember that these processes are running concurrently, but on a single CPU only one of the threads at a time can actually be running. Assume there is only one ticket remaining to be sold, and ticket agent Bob's thread has already executed line 3 but has not yet executed line 4, and his thread times out. Meanwhile agent Mary's thread sells the last ticket, and when Bob's thread runs again it sells one ticket over the limit.
/* constRef.c
   Show how pass by constant reference (pointer) in C++
gives pass-by-reference efficiency with pass-by-value
semantics.
*/

#include <iostream.h>

// Call by value, copy is allocated space on stack
void foo(int x)
{
    x = 5;
    printf("Value is: %d", x);
}

// Call by constant reference, only pointer is on stack
void foobar(const int &x)
{
    x = 7; // illegal
    printf("Value is: %d", x); // x implicitly dereferenced
}

int main()
{
    int number = 3;
    foo( number);
    foobar( number);
}
program main;
  var w, x, y, z : integer;
  procedure sub(a, b, c, d : integer);
    var i : integer;
    ...
  end;
  begin
    ...
    call sub (w, x, y, z); {pass w by value, x by result,
    y by value-result, z by reference}
    ...
  end;

program main

<table>
<thead>
<tr>
<th>Code</th>
<th>Stack</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>value of a</td>
<td>ref. to a</td>
</tr>
<tr>
<td>x</td>
<td>value of b</td>
<td>assign to b</td>
</tr>
<tr>
<td>y</td>
<td>value of c</td>
<td>ref. to c</td>
</tr>
<tr>
<td>z</td>
<td>address (at start)</td>
<td>assign to c</td>
</tr>
<tr>
<td></td>
<td>address (at start)</td>
<td>ref. to d</td>
</tr>
</tbody>
</table>

type MATRIX_TYPE is array (INTEGER range <>,
  INTEGER range <>) of FLOAT;

MATRIX_1 : MATRIX_TYPE (1..100, 1..20);

A function that returns the sum the elements of arrays of MATRIX_TYPE

type follows:

function SUMER (MAT : in MATRIX_TYPE) return FLOAT is
  SUM : FLOAT := 0.0;
  begin
    for ROW in MAT'range(1) loop
      for COL in MAT'range(2) loop
        SUM := SUM + MAT (ROW, COL);
      end loop;  -- for COL ...
    end loop;  -- for ROW ...
  return SUM;
  end SUMER;
void swap1 (int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

Suppose this function is called with
swap1 (c, d);

a = c       -- Move first parameter value in
b = d       -- Move second parameter value in
temp = a
a = b
b = temp

Values of c and d are unchanged.

void swap2 (int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

swap2 can be called with
swap2 (&c, &d);

The actions of swap2 can be described with
a = &c       -- Move first parameter address in
b = &d       -- Move second parameter address in
temp = *a
*a = *b
*b = temp

Values of c & d are changed.

void swap2 (int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}

C++ version using reference parameters.

procedure swap2 (var a, b: integer)
    temp : integer;
    begin
    temp := a;
    a := b;
    b := temp
    end;

Suppose the Pascal version of swap2 is called with
swap2 (i, list [i]);

In this case, the actions are described by
a = &i       -- Move first parameter address in
b = &list [i] -- Move second parameter address in
temp = *a
*a = *b
*b = temp

Changing *a (which is i) before *b (which is list[i])
does not change the subscript i, since the address of
list[i] is computed at the time of the call.
Assume pass-by-value-result:

```plaintext
procedure swap3 (a : integer, b : integer) is
  temp : integer;
begin
  temp := a;
  a := b;
  b := temp;
end swap3;
```

Suppose swap3 is called with

```plaintext
swap3 (c, d);
```

The actions of swap3 with this call are

```plaintext
addr_c = &c      -- Move first parameter address in
addr_d = &d      -- Move second parameter address in
a = c           -- Move first parameter value in
b = d           -- Move second parameter value in
temp = a        -- Move first parameter value in
a = b           -- Move second parameter value in
b = temp
*addr_c = a     -- Move first parameter value out
*addr_d = b     -- Move second parameter value out
```

So once again, this swap subprogram operates correctly. Next, consider the call

```plaintext
swap3 (i, list [i]);
```

In this case, the actions are

```plaintext
addr_i = &i      -- Move first parameter address in
addr_listi = &list [i]  -- Move second parameter address in
a = i           -- Move first parameter value in
b = list [i]    -- Move second parameter value in
temp = a        -- Move first parameter value in
a = b           -- Move second parameter value in
b = temp
*addr_i = a     -- Move first parameter value out
*addr_listi = b -- Move second parameter value out
```

Still correct, since return addresses are computed at the time of the call rather than at the time of the return.

```plaintext
int i = 3; /* i is a global variable */
void fun (int a, int b) {
  i = b;
}
void main () {
  int list [10];
  list [i] = 5;
  fun (i, list [i]);
}
```

In fun, i and a are aliases. The actions of fun, assuming pass-by-value-result, are

```plaintext
addr_i = &i      -- Move first parameter address in
addr_listi = &list [i]  -- Move second parameter address in
a = i           -- Move first parameter value in
b = list [i]    -- Move second parameter value in
i = b           -- Move first parameter value in
*baddr_i = a    -- Move first parameter value out
*addr_listi = b -- Move second parameter value out
```

If pass-by-reference is used instead, the aliasing means that the global value will unfortunately be changed.
/* Multipurpose sorting program using function pointers */
#include <stdio.h>
#define SIZE 10

void bubble(int *, const int, int (*)(int, int));
int ascending(const int, const int);
int descending(const int, const int);

main()
{
    int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
    int counter, order;

    printf("Enter 1 to sort in ascending order,\n");
    printf("Enter 2 to sort in descending order: ");
    scanf("%d", &order);

    printf("\nData items in original order\n");
    for (counter = 0; counter <= SIZE - 1; counter++)
        printf("%4d", a[counter]);

    if (order == 1) {
        bubble(a, SIZE, ascending);
        printf("\nData items in ascending order\n");
    }
    else {
        bubble(a, SIZE, descending);
        printf("\nData items in descending order\n");
    }

    for (counter = 0; counter <= SIZE - 1; counter++)
        printf("%4d", a[counter]);

    printf("\n");
    return 0;
}

void bubble(int *work, const int size, int (*compare)(int, int))
{
    int pass, count;
    void swap(int *, int *);

    for (pass = 1; pass <= size - 1; pass++)
        for (count = 0; count <= size - 2; count++)
            if (((*compare)(work[count], work[count + 1]))
                swap(&work[count], &work[count + 1]);

    void swap(int *element1Ptr, int *element2Ptr)
    {
        int temp;
        temp = *element1Ptr;
        *element1Ptr = *element2Ptr;
        *element2Ptr = temp;
    }

    int ascending(const int a, const int b)
    {
        return b < a;
    }

    int descending(const int a, const int b)
    {
        return b > a;
    }

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
Ada generic unit to sort elements of different types

generic

type ELEMENT is private;
type VECTOR is array (INTEGER range <> ) of ELEMENT;
procedure GENERIC_SORT (LIST : in out VECTOR);
procedure GENERIC_SORT (LIST : in out VECTOR) is
begin
  TEMP : ELEMENT;
  for INDEX_1 in LIST'FIRST .. INDEX'PREDECESSOR (LIST'LAST) loop
    for INDEX_2 in INDEX'SUCCESSION (INDEX_1) .. LIST'LAST loop
      if LIST (INDEX_1) > LIST (INDEX_2) then
        TEMP := LIST (INDEX_1);
        LIST (INDEX_1) := LIST (INDEX_2);
        LIST (INDEX_2) := TEMP;
      end if;
    end loop; -- for INDEX_2 ...
  end loop; -- for INDEX_1 ...
end GENERIC_SORT;

procedure INTEGER_SORT is new GENERIC_SORT (ELEMENT => INTEGER;
                                              VECTOR => INT_ARRAY);

C# generic sort

template <class Type>

void generic_sort (Type list [], int len) {
  int top, bottom;
  Type temp;
  for (top = 0; top < len - 2; top++)
    for (bottom = top + 1; bottom < len - 1; bottom++)
      if (list [top] > list [bottom]) {
        temp = list [top];
        list [top] = list [bottom];
        list [bottom] = temp;
      }
}

An example instantiation of this template function is:

float flt_list [100];
... generic_sort (flt_list, 100);
REAL A(100)
INTEGER B(250)
COMMON /BLOCK1/ A, B

REAL C(50), D(100)
INTEGER E(200)
COMMON /BLOCK1/ C, D, E