# Array Pitfalls in C



## Arrays in C

- No real arrays in C
- An array is just a pointe

```
#include <stdio.h>

int main () {
   char *str = "bugy";

   printf ("%c\n", *str);
   printf ("%c\n", * (str+1));
   printf ("%c\n", * (2+str));
   printf ("%c\n", str [3]);
}
```

```
Command Prompt
C:\tmp>type try.c
#include <stdio.h>

int main () {
  char *str = "bugy";

  printf ("%c\n",*str);
  printf ("%c\n",*(str+1));
  printf ("%c\n",*(2+str));
  printf ("%c\n",str[3]);
}

C:\tmp>gcc try.c

C:\tmp>a.exe
b
u
g
y

C:\tmp>_
```



## C "Arrays" are Just Pointers: No Safety

```
#include <stdio.h>
#include <string.h>

char x [4] = {'A', 'B', 'C', 'D'};
char y [4] = {'E', 'F', 'G', 'H'};
char z [4] = {'I', 'J', 'K', 'L'};
…
int main () {
   char *p;

   display();
   p = x;
   memcpy (p, "0123456789", 10);
   display ();
   x [6]= '?';
   display ();
}
```

```
void display_arr (char *name, char
   *arr, int n) {
      int k;
      printf ("%s = ", name);
      for (k = 0; k < n; k++)
     printf ("%c", arr [k]);
      printf ("\n");
}

void display () {
   display_arr ("x", x, 4);
      display_arr ("y", y, 4);
      display_arr ("z", z, 4);
      printf ("-----------\n");
}
```



## What Does the C Standard Say?

```
…
char x [4] = {'A', 'B', 'C', 'D'};
char y [4] = {'E', 'F', 'G', 'H'};
char z [4] = {'I', 'J', 'K', 'L'};
…
int main () {
   char *p;

   display();
   p = x;


   memcpy (p, '0123456789', 10);
   display ();


   x [6]= '?';
   display ();
}
```

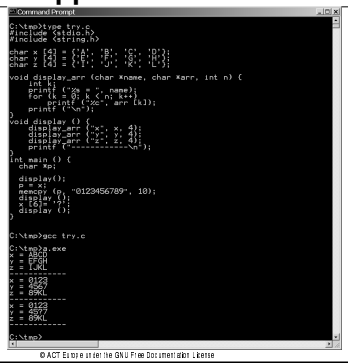
Set p to the address of x

Copy "0123456789" in the 10 consecutive memory locations pointed by p

Result is undefined since x is a pointer that points to 4 consecutive preallocated bytes in memory, not 7 compiler is free to do what it wants



## What Happens in Practice





## What Happens in Memory: Allocation

```
#include <stdio.h>
#include <string.h>

char x [4] = {'A', 'B', 'C', 'D'};
char y [4] = {'E', 'F', 'G', 'H'};
char z [4] = {'I', 'J', 'K', 'L'};
…
int main () {
   char *p;

   display();
   p = x;
   memcpy (p, '0123456789', 10);
   display ();
   x [6]= '?';
   display ();
}
```

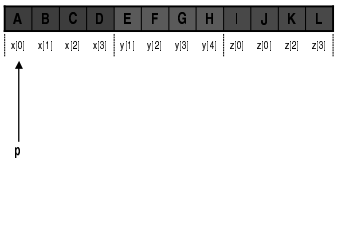
| A | B | C | D | E | F | G | H | I | J | K | L |
|---|---|---|---|---|---|---|---|---|---|---|---|
| x[0] | x[1] | x[2] | x[3] | y[1] | y[2] | y[3] | y[4] | z[0] | z[0] | z[2] | z[3] |

## Pointer Copy

```
#include <stdio.h>
#include <string.h>

char x [4] = {'A', 'B', 'C', 'D'};
char y [4] = {'E', 'F', 'G', 'H'};
char z [4] = {'I', 'J', 'K', 'L'};
...
int main () {
    char *p;

    display();
    p = x;
    memcpy (p, "0123456789", 10);
    display ();
    x [6]= '?';
    display ();
}
```



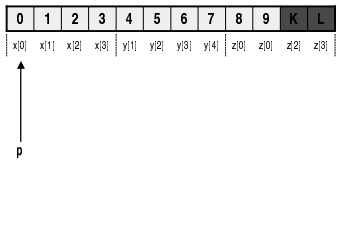



## memcpy

```
#include <stdio.h>
#include <string.h>

char x [4] = {'A', 'B', 'C', 'D'};
char y [4] = {'E', 'F', 'G', 'H'};
char z [4] = {'I', 'J', 'K', 'L'};
...
int main () {
    char *p;

    display();
    p = x;
    memcpy (p, "0123456789", 10);
    display ();
    x [6]= '?';
    display ();
}
```



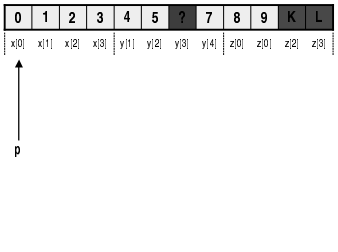



## x[6] = '?'

```
#include <stdio.h>
#include <string.h>

char x [4] = {'A', 'B', 'C', 'D'};
char y [4] = {'E', 'F', 'G', 'H'};
char z [4] = {'I', 'J', 'K', 'L'};
...
int main () {
    char *p;

    display();
    p = x;
    memcpy (p, "0123456789", 10);
    display ();
    x [6]= '?';
    display ();
}
```






## C and Arrays: Summary

- No real arrays
- Very low level
- Fundamentally unsafe mechanism
- Too many ways of expressing the same thing



## What about C++ and Java

- C++
  - Same problem as in C
  - You can redefine the '[ ]' operator in C++ for class types thereby providing your own checks and semantics. However this makes code hard to read since when you see something like s [k] it does not mean any more what you've been used to by many years of C and makes the code hard to understand if used in an uncontrolled fashion
- Java
  - Has real arrays
  - Java model inspired from Ada
  - Difference between Java and Ada is that Java arrays MUST start at index 0, Ada can have arbitrary bounds



## Arrays in Ada

- Ada has real arrays (1-dimensional and multi-dimensional)
- Ada array can have its size determined at run-time
  - Local variable length arrays are allowed in the latest C standad (C99)
- Ada array bounds can be arbitrary, lower bound does not have to start at 0

## "One-of-a-Kind" Arrays

```
procedure Compute (N : Integer) is
   A : array (1 .. N) of Float;
begin
   ...
end Compute;
```

- In Ada Arrays can have arbitrary bounds
- The bounds can be dynamic values



## Typed Arrays

```
procedure Compute (N : Integer) is
   type Arr is array (Integer range <>) of Float;

   A : Arr (1 .. N) := (others => 9);
   B : Arr := A;
   C : Arr (11 .. 20) := (1, 2, others => 0);

begin
   C := A;
   C (15 .. 18) := A (5 .. 8);
end Compute;
```

- B takes its bounds from A
- If C'Length /= A'Length then Constraint_Error is raised
- If A'Last < 8 then Constraint_Error is raised



## Arrays in Ada are Safe

- If you try to index a non-existent array position, a Constraint_Error exception is raised

```
procedure Checks is
   A : array (1 .. 100) of Integer;

begin
   A (101) := 1;
end Checks;
```

Exception raised



## Ada Arrays are Powerful: Array Bounds

```
procedure Calc is
   type Vector is array (Natural range <>) of Float;

   function Max (V : Vector) return Float is
      M : Float := Float' First;
   begin
      for K in V'Range loop
         if V(K) > M then
            M := V(K);
         end if;
      end loop;
      return M;
   end Max;

   V1 : Vector := (1.0, 2.0, 3.0);  -- V'First = 0 and V'Last = 2
   V2 : Vector (1 .. 100) := (1.0, 2.0, others => 5.0);

   X : Float := Max (V1);  -- X = 3.0
   Y : Float := Max (V2);  -- Y = 5.0
begin
   ...
end Calc;
```



Ada Core TECHNOLOGIES.INC

ACT EUROPE

# Parameter Passing in C, Java and Ada



## Two Parameter Passing Modes in C/Java

- By value
- By constant value
- In C++ there are additional modes
  - By reference
  - By constant reference

## By Value

- The parameter can be changed inside the function but the value of the original parameter is not modified

```
void copy (int x, int y) {
    x = y;
}

void try () {
    int a = 0;
    int b = 9;
    copy (a, b);

    // a == 0 here
}
```



## By Constant Value

- In C

```
void copy (const int x, const int y) {
    x = y;
}
```

Compilation error

- In Java

```
void copy (final int x, final int y) {
    x = y;
}
```

Compilation error



## Changing the Actual Parameter

- In Java
  - There is NO way to change the value of a scalar parameter
    - You have to do it yourself by hand ...
  - For a record (class) or array parameter you can only change thecomponents
  - This is a problem: Java really makes the programmer's life hardhere
- In C
  - You have to create a pointer by hand and use that
  - The programmer must perform tedious bookkeeping by specifying
    - *p and &s
  - This also makes the code hard to change

```
void foo (int *p);
...
int s;
foo (& s);
```



## Parameter Passing Modes in Ada

| | |
|---|---|
| `procedure Open_File (X : String);`<br>`procedure Open_File (X : in String);` | **in**<br>• It's the default mode, the `in` can be omitted<br>• Inside the procedure or function X is a *constant* initialized by the value of the actual parameter<br>• Functions can only have parameters of mode i |
| `function Log (X : Float) return Float;`<br>`function Log (X : in Float) return Float;` | |
| `procedure Increment (X : in out Float);` | **in out**<br>• Inside the procedure X is a *variable* initialized by the value of the actual parameter<br>• The actual parameter is updated with the last value of X when the procedure terminates. |
| `procedure Copy (Y : Float; X : out Float);` | **out**<br>• Inside the procedure X is an uninitialized *variable*<br>• The actual parameter is updated with the last value of X when the procedure terminates |