CS 101 Computer Programming and Utilization

Lecture 11

Passing Arrays as parameters
Dynamic allocation of arrays
Recursion

Feb 15, 2011

Prof. R K Joshi
Computer Science and Engineering
IIT Bombay
Email: rkj@cse.iitb.ac.in

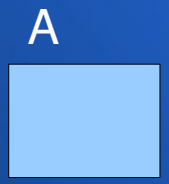
Revision: functions, arrays in parameters

- Formal parameters
- Actual parameters
- Pass by value
- Passing arrays as parameters
  - What type to use in declaration?
  - What type to use in definition
  - What name to pass as parameter in invocation?
- Improving genericity of a function
  - Passing fixed sized arrays vs.

    passing variable sized arrays
- Arrays are also passed by copy
  - But changes to elements are visible outside
  - What is passed in when an array is passed into a parameter in an invocation?

Passing Arrays as parameters .. a memory allocation view

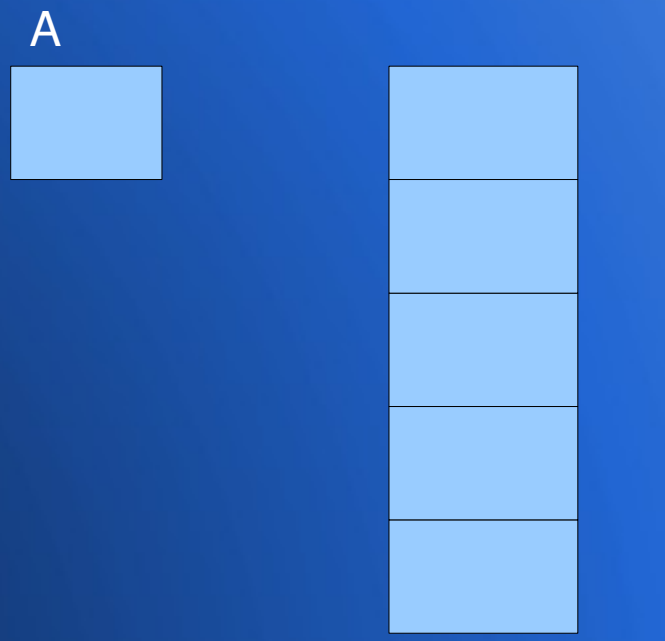
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Location A is created

Array in parameter

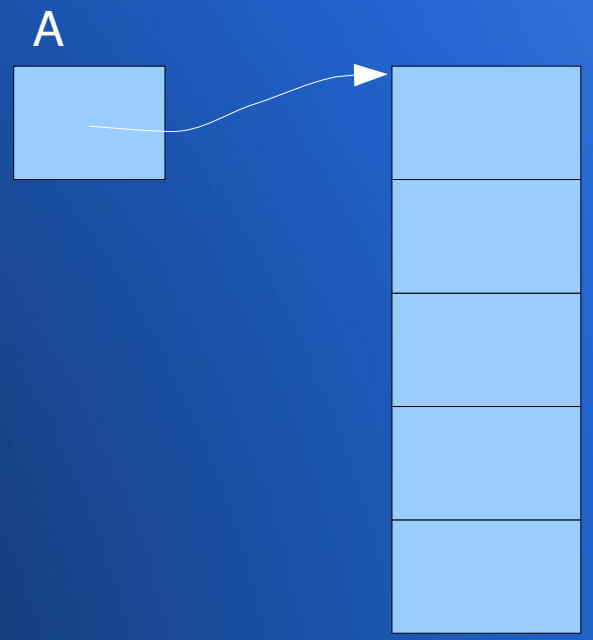
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Array strip of size 5 is created

Array in parameter

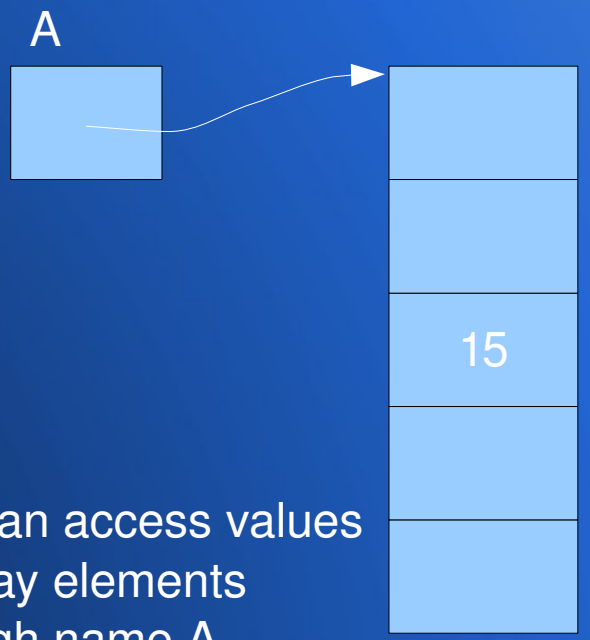
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



A points to the array strip

Array in parameter

```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```

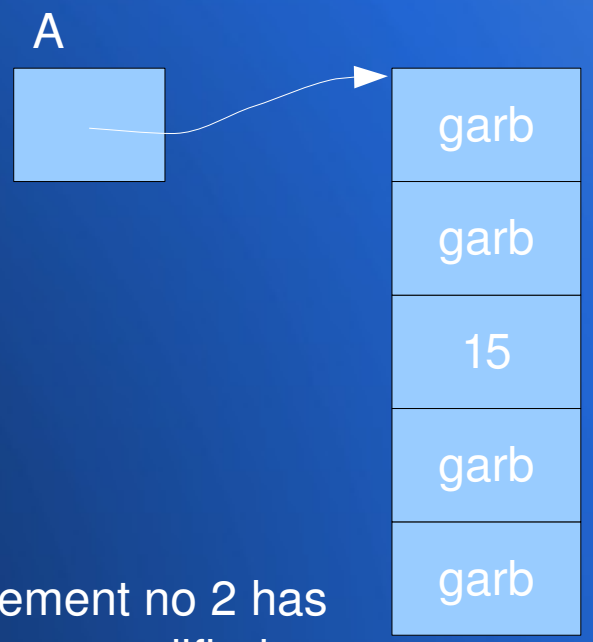


You can access values of array elements through name A

And what values do you have in other locations?

Array in parameter

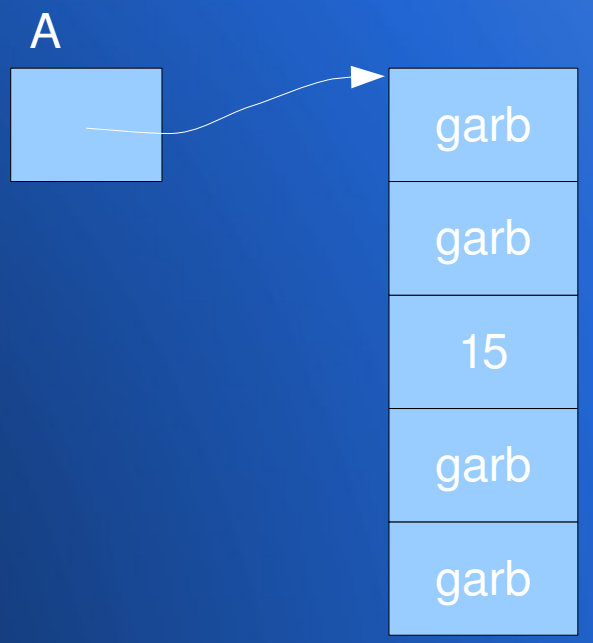
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Element no 2 has been modified

Array in parameter

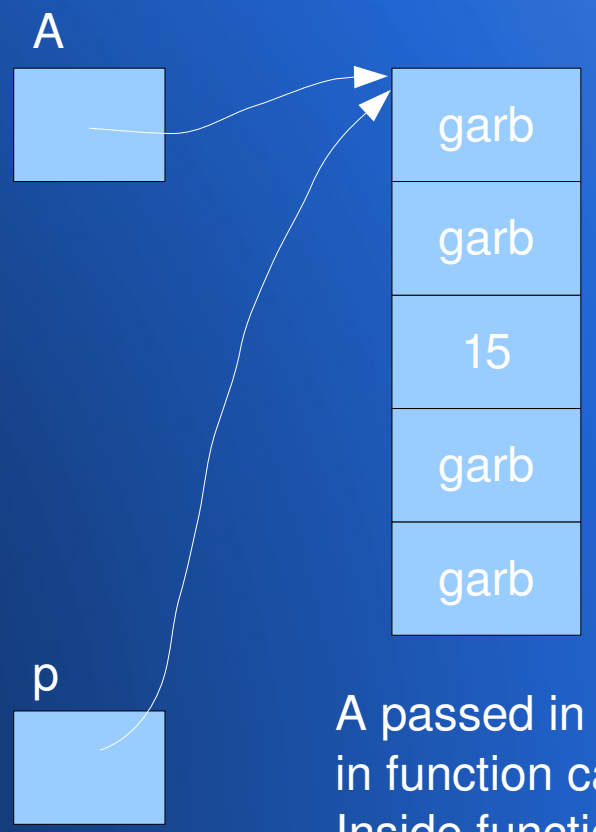
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



A passed in as parameter in function call.
Inside function invocation, Location for p is created

Array in parameter

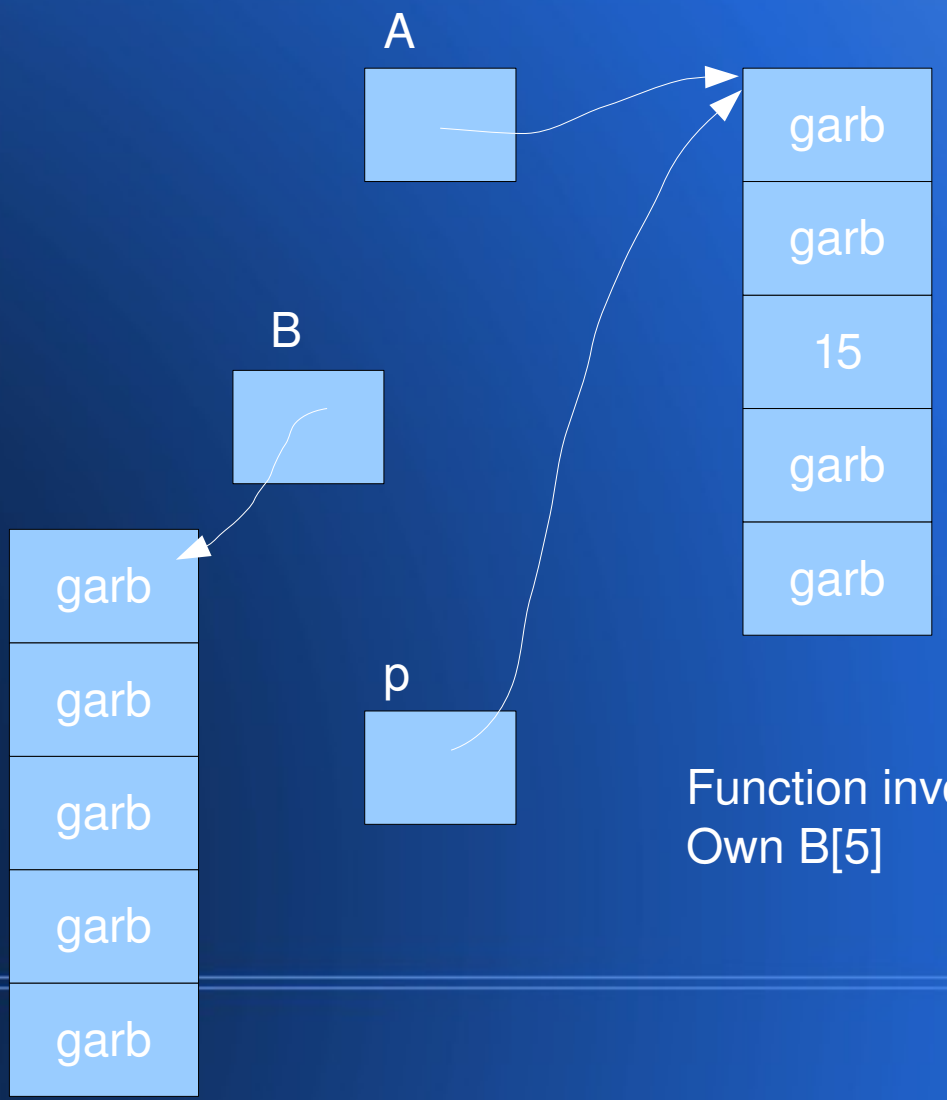
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



A passed in as parameter in function call.
Inside function invocation, Location for p is created and value of A is copied in p

Array in parameter

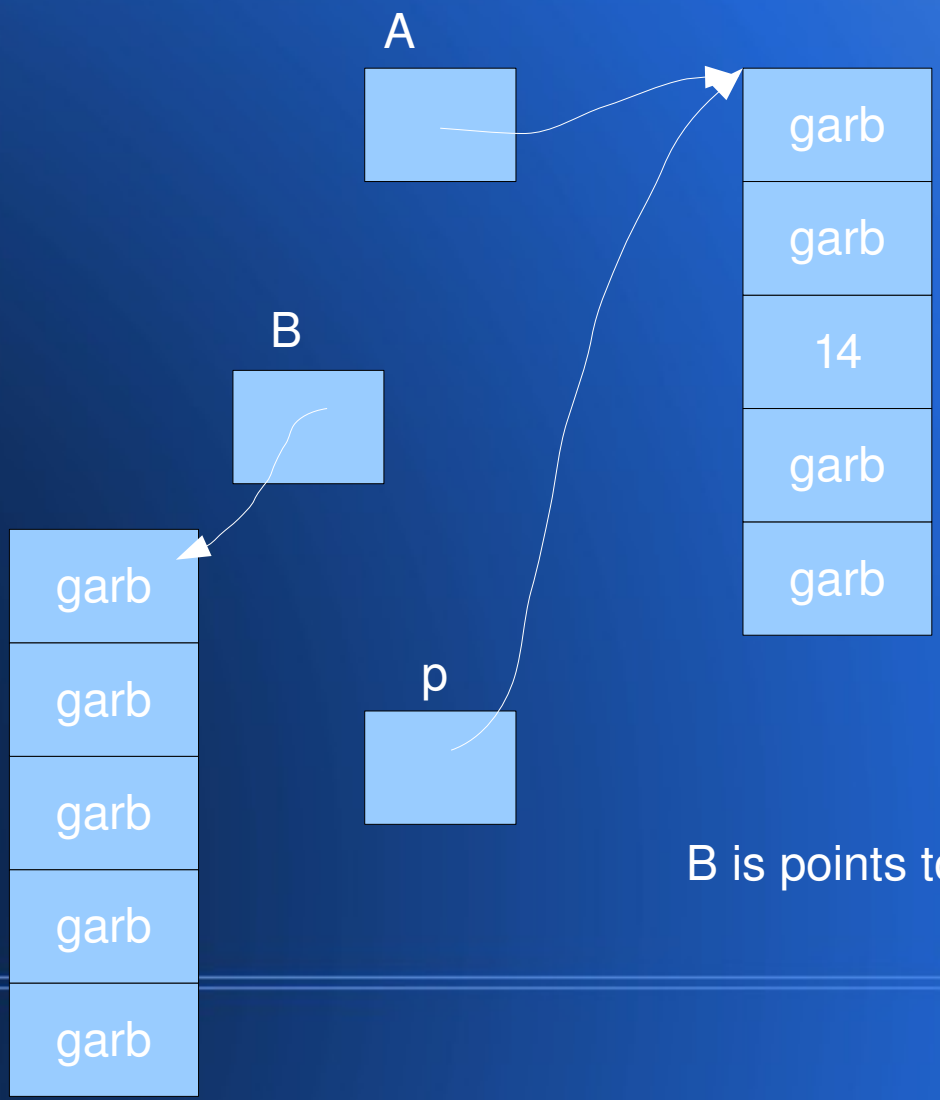
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Function invocation has its Own B[5]

Array in parameter

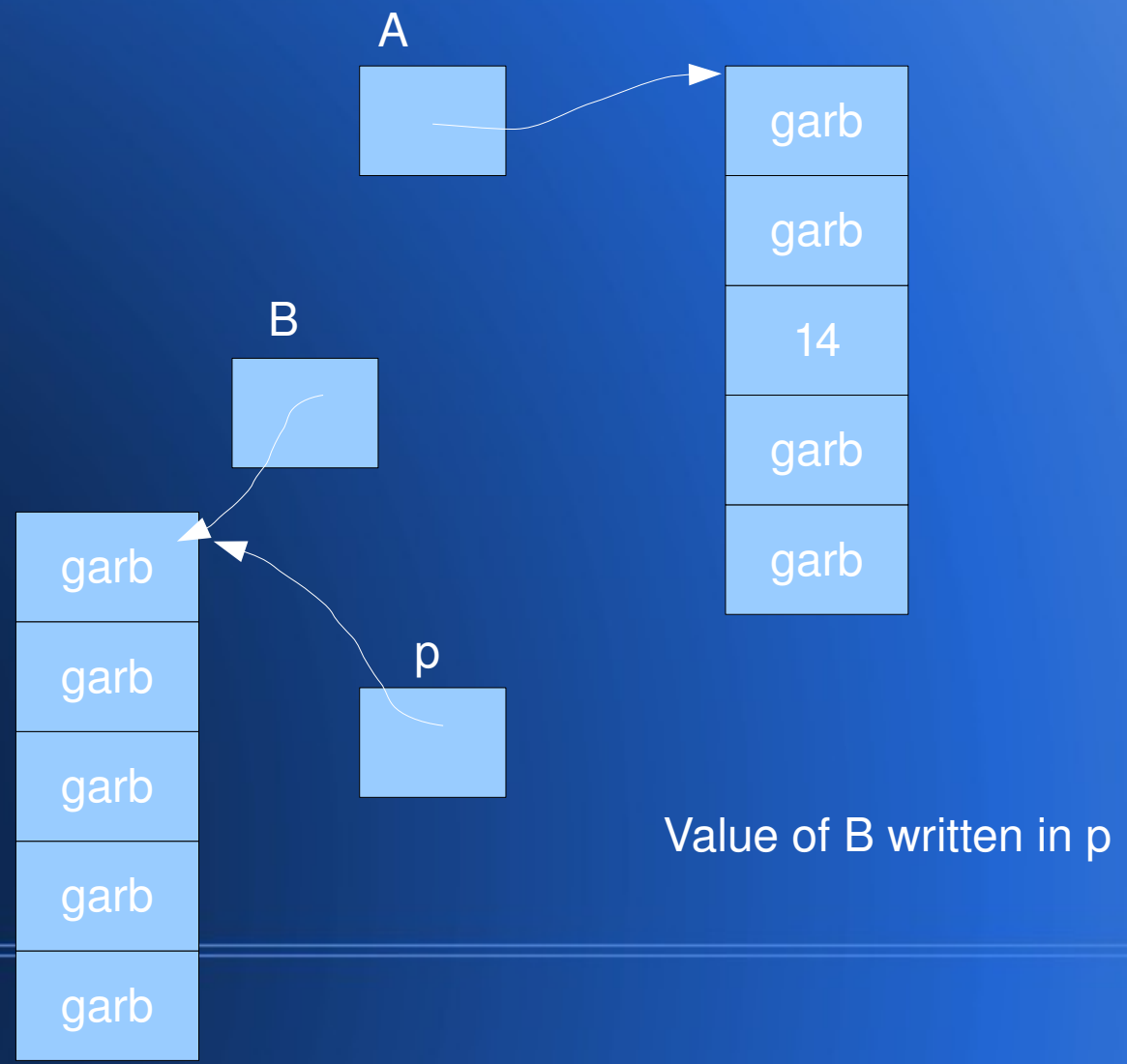
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



B is points to the new array

Array in parameter

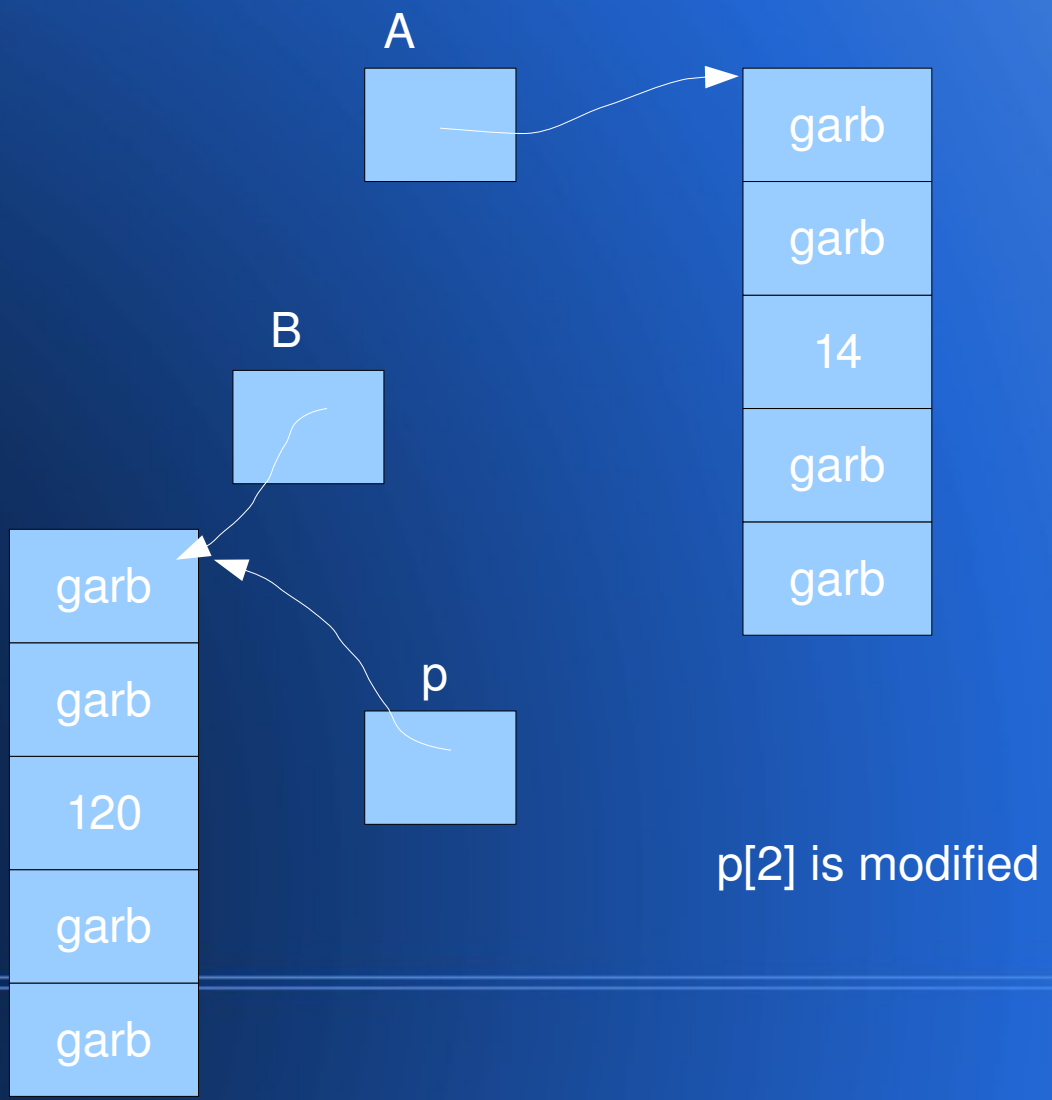
```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Value of B written in p

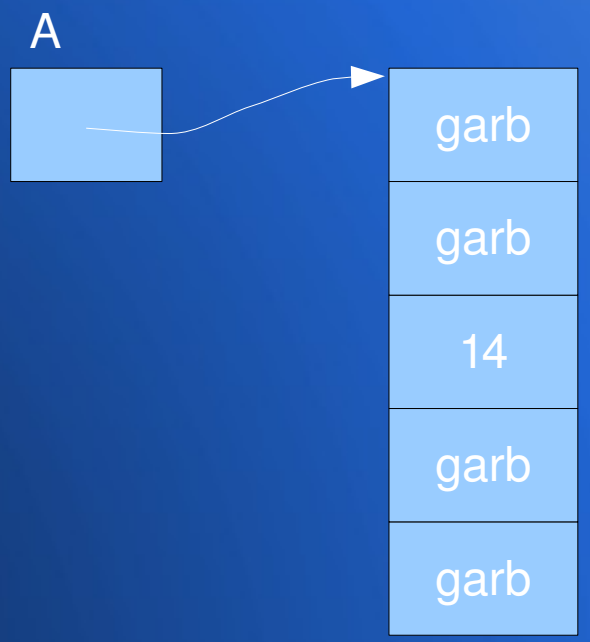
Array in parameter

```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Array in parameter

```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```



Function call returned everything inside the invocation cleaned up.

Array in parameter

```
int func (int [ ]);
main () {

  int A [5];

     A[2] = 15;

      func (A);
}

int func (int p[ ]] ) {

     int  B[5]
     p [2] = p[2]-1;
     p=B;
     p[2]=120
     return 0;
}
```

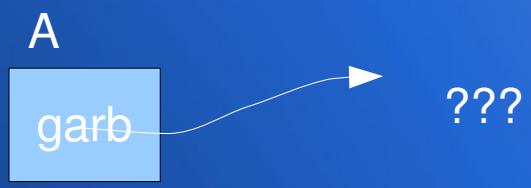
Program terminated!!
Everything inside the program cleaned up

Dynamic Arrays

- Size of the array is known during run time
- So the array declaration cannot specify the size of the array
- One cannot use an array index till the array size is known, i.e. in other words the array gets allocated
- Int A[10] // declaration specifies size
- Int A[ ]={1,2,53};   // declaration specifes size

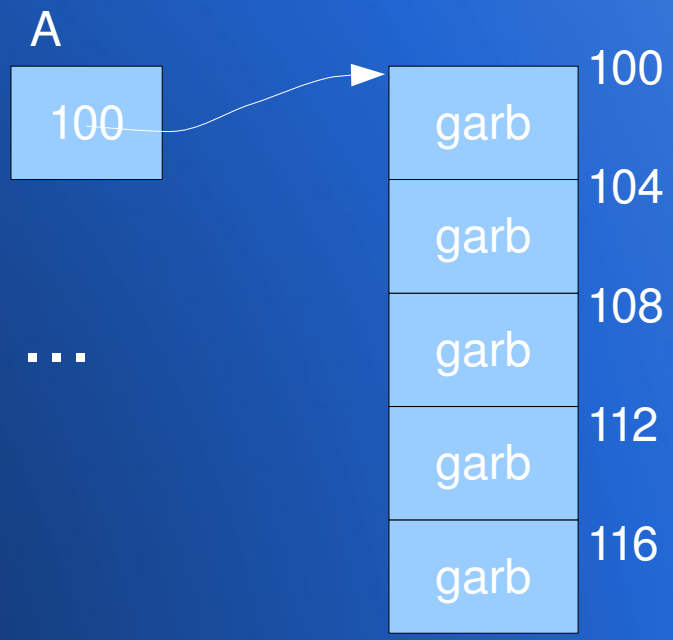
Using a pointer so that we can delay the allocation

- Int *A;
- … A is just a pointer ..
- .. it points to nothing so far ...



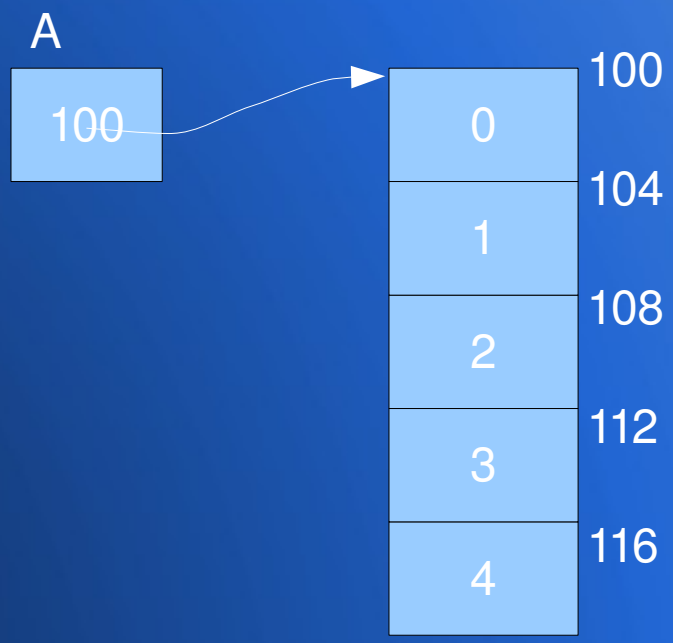
Using a pointer so that we can delay the allocation

- Int *A;
- … A is just a pointer ..
- .. it points to nothing so far ...
- A = new int [5];



Using the dynamically allocated array

- Int *A;
- A = new int [5];
- For (int i=0; i<5; i++)
  - A[i] = i;

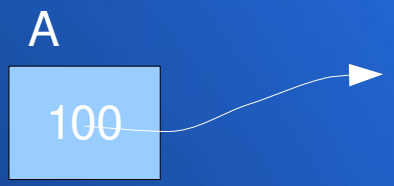


Deleting the dynamically allocated array

- Int *A;
- A = new int [5];
- For (int i=0; i<5; i++)
  - A[i] = i;
- Delete[] A;

Deleting the dynamically allocated array

- Int *A;
- A = new int [5];
- For (int i=0; i<5; i++)
  - A[i] = i;
- Delete[] A;



??? nothing here!

But you can make an assignment once again with a new array.

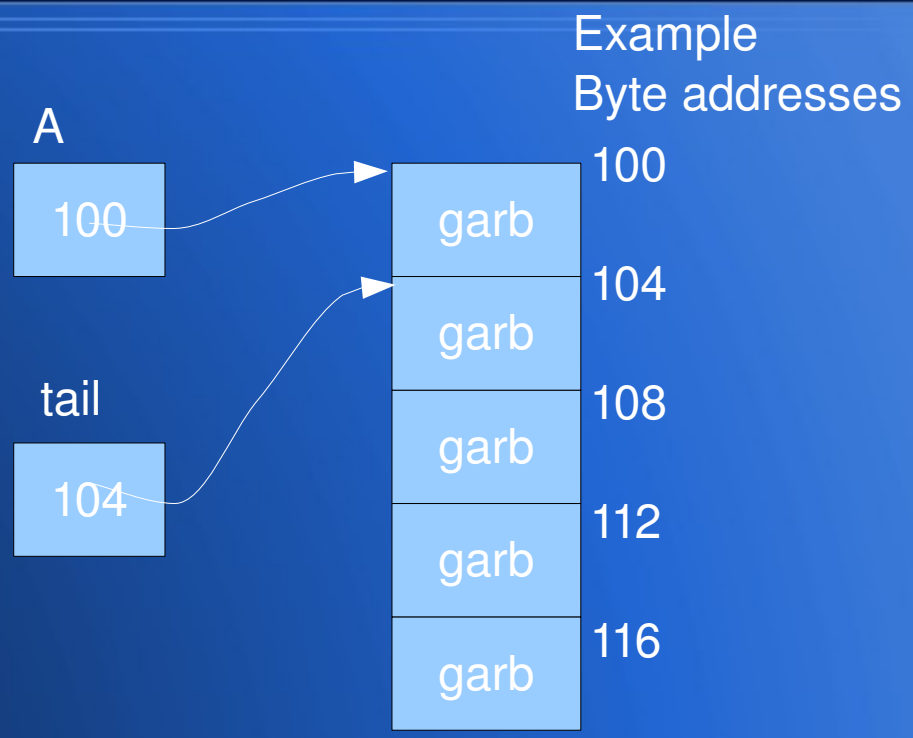
Summary of dynamic allocation of arrays

- Use a pointer variable to hold an address
- Allocate an array of required size with the 'new' operator
- The array can be indexed as usual at lval and rval
- Delete when done
- You can reuse the pointer for another array.

Pointer arithmetic with array names

- Int A[5];
- Int *tail;

  tail = A+1



Observe that size of tail is reduced by 1
Using tail without out of bound index
Is programmer's responsibility anyway!

Recursion

Recursion: *A full expansion of a function is expressed in terms of itself*

- Factorial
  - Fact (n) = n*fact(n-1)
  - Fact (0) = 1
- Fibonacci
  - Fib (n) = fib (n-1) + fib (n-2)
  - Fib (0) = 0
  - Fib (1) = 1
- Observe the terminating expansion
    - Without this, the recursion will never terminate