CSE2031 Software Tools - Arrays and Pointers

Summer 2010

Przemyslaw Pawluk

Department of Computer Science and Engineering
York University
Toronto

May 18, 2010

What have we done last time?

Basic information about testing:

- Black- and Glass-box tests
- Random tests
- Regression
- Pre-, Post- and boundary conditions
- Assertions
- C directives #declare, #include
- modifiers extern and static

What we will do today?

1. Arrays
2. Pointers
3. Pointers and Arrays
4. Pointers and Functions
5. Multidimensional arrays, Pointers to Pointers
6. Useful tricks and complicated declarations

Plan

Array

- Data structure
- Grouping of data of the same type
- Indicated with brackets containing positive integer constant or expression following identifier
- Loops commonly used for manipulation
- Programmer sets size of array explicitly (static structure)

Declarations

Syntax

```
type name[size];
```

Examples

```
int bigArray[10];
double a[3];
char grade[10], oneGrade;
```

Declarations cnt.

- Declaration of the array allocates memory

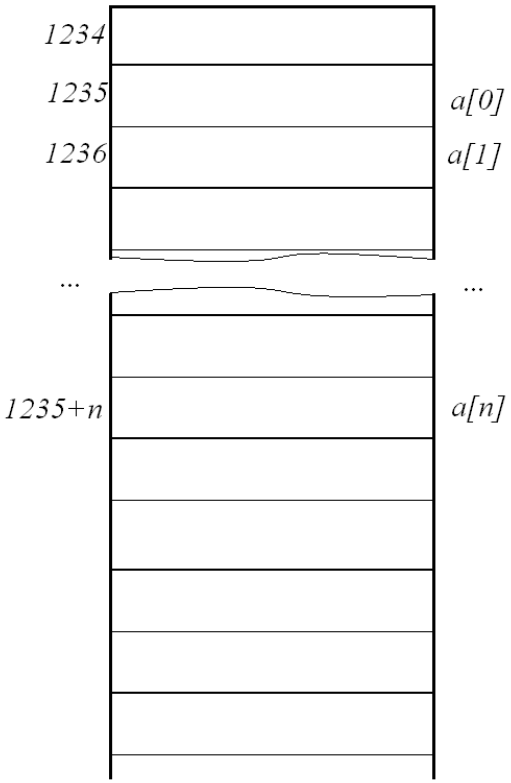
```
char mark[6];
```

- Declares array of 6 integers named "mark"
- Similar to declaring five variables:

```
char mark[0], mark[1], mark[2], mark[3], mark[4]
```

Elements, indexed (subscripted) variables
Arrays are indexed from 0 to size-1!

Arrays in memory



Initialization

Initialization enclosed in curly brackets (in declaration)

```
int a[5] = {1,2};
```

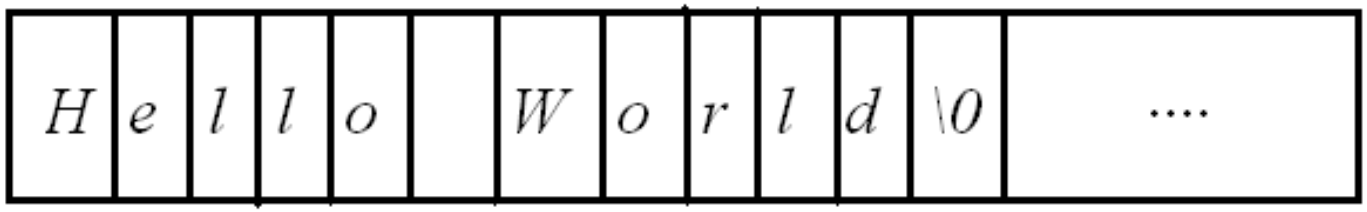
Declares array a and initializes first two elements and all remaining set to zero

```
int b[] = {5,4,3,2,1};
```

Declares array b and initializes all elements and sets the length of the array to 5

Strings=Arrays of chars

```
char msg[]="Hello world!";
```



Array access

Access to elements is through the index

Read

```
x=a[2];
```

Write

```
a[2]=2;
```

Question

What's the difference between:

```
a[i]++;
```

```
a[++i];
a[i++];
```

Example

Let's test the difference on the example - Write word in the column

Plan

Pointers

- Memory address of a variable
- Declared with data type, * and identifier

```
type * pointerVar1, * pointerVar2 ...;
```

- There has to be a * before EACH of the pointer variables
- Example.

```
double * p;
int *p1, *p2;
```

Pointers and Variables

- We can get the variable's address (pointer) using '&'

```
type *pointer_name = &variable;
```

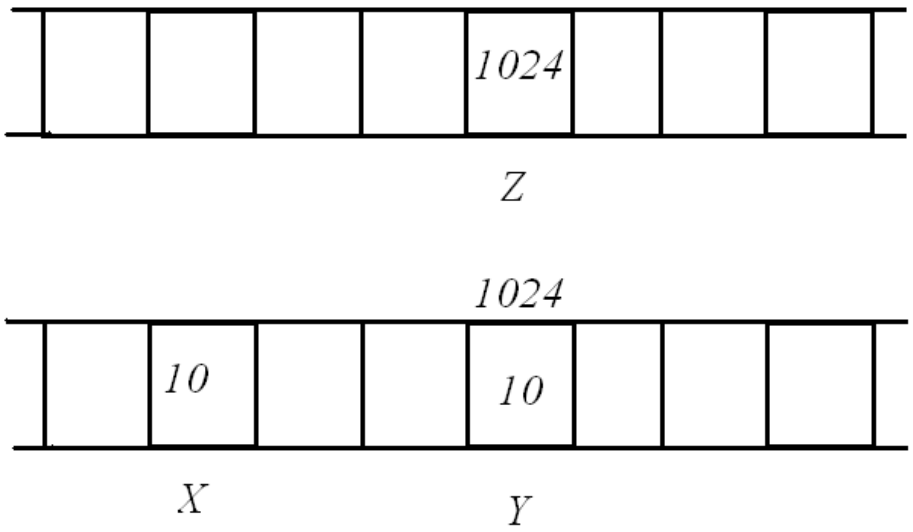
- We can get the value under the address using '*'

```
type variable = *pointer_variable;
```

Pointer Variables

```
int x,y *z;

x=10;
y=x;
z=&y;
```



Be careful!

When assigning values to the pointer variable we should use '&'. Assigning explicit value is a bad idea!

```
int *x, z;

x=0x12345A;

x=&z;
```

Explicit assignment is bad idea!

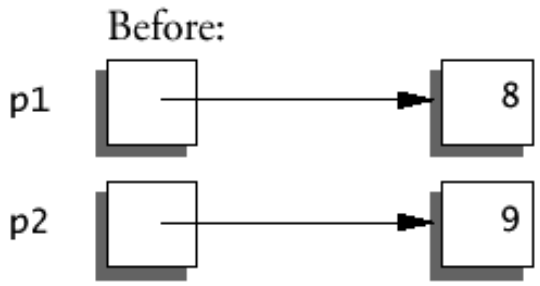
'&' gives you a makes you sure that this address exists and is valid.

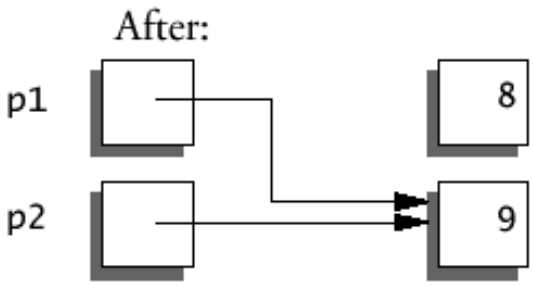
Example

Simple example with pointers

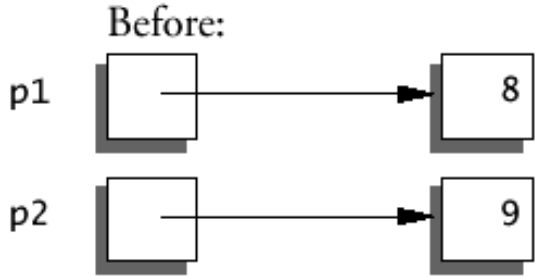
Assignments

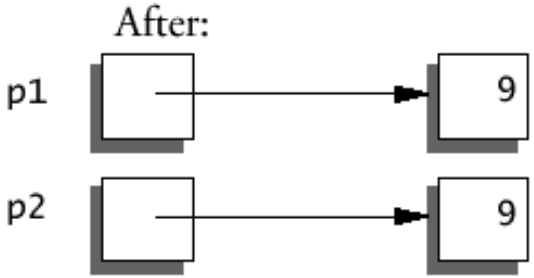
```
p1 = p2;
```






```
*p1 = *p2;
```





Plan

Pointers and Arrays

Identifier of an array is equivalent to the address of its first element

```
int numbers[20];
int *p;
p = numbers; /*Valid*/
numbers = p; /*Invalid*/
```

- p and numbers are equivalent and they have the same properties
- Only difference is that we could assign another value to the pointer p whereas numbers will always point to the first of the 20 integer numbers of type int

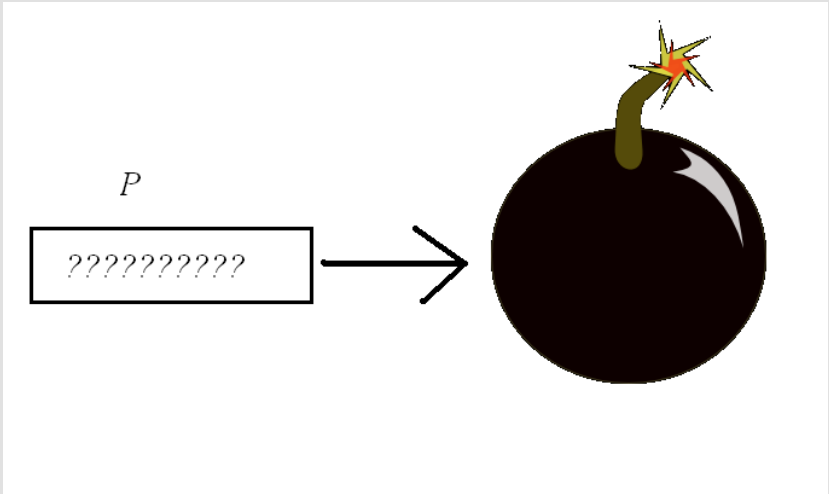
Bad pointers!

Bad Pointer

It is a pointer variable that was not initialized (has undetermined value)

What happens at runtime when the bad pointer is dereferenced?

```
int* p; /*allocate the pointer,
          * but not the pointee*/
*p = 42; /*this dereference is a
           *serious runtime error*/
```

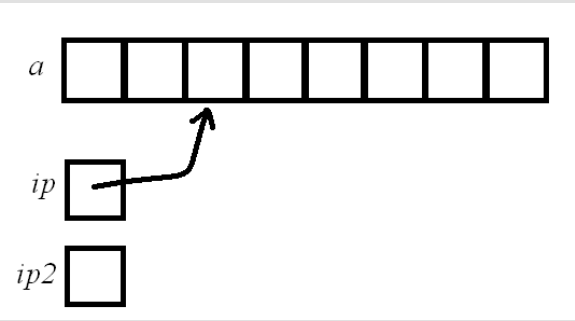


Example

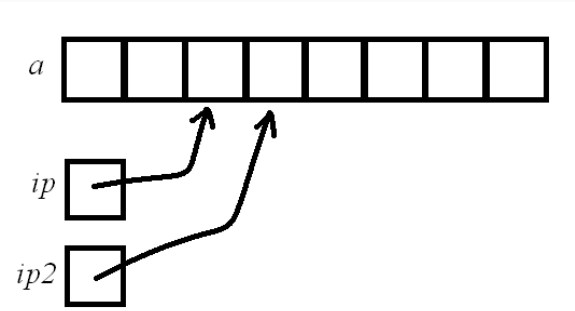
Bad pointers – segmentation fault

Pointer Arithmetic

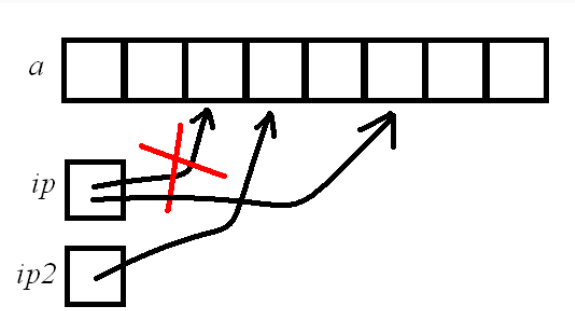
```
int *ip, *ip2;
int a[10];
ip = &a[3];
```




```
ip2 = ip + 1;
```

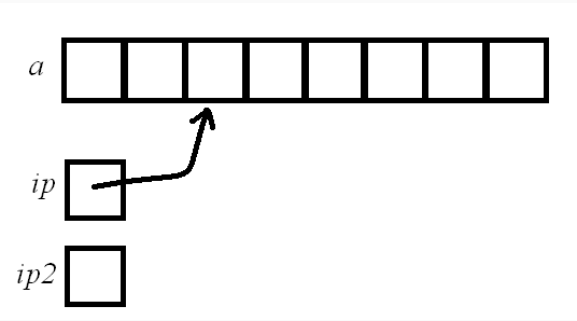



```
ip+=3;
```

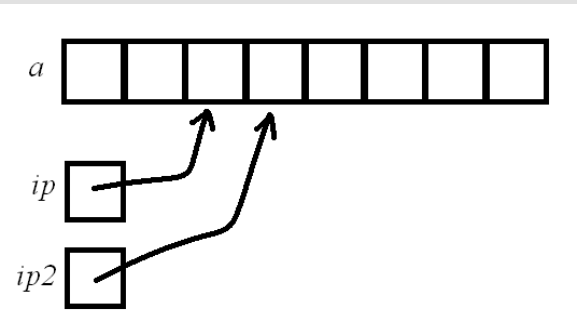


Pointer Arithmetic

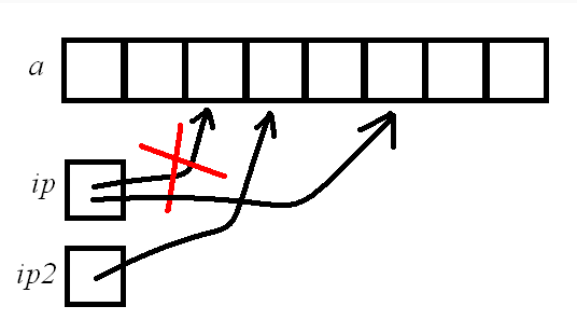
```
int *ip, *ip2;
int a[10];
ip = &a[3];
```




```
ip2 = ip + 1;
```

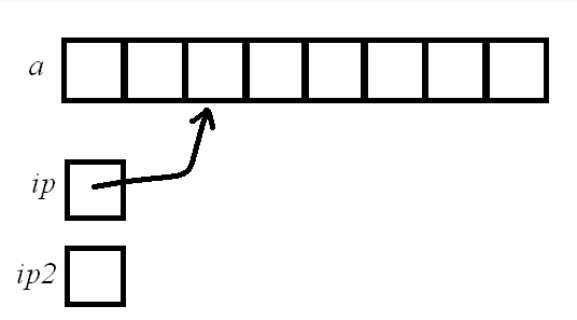



```
ip+=3;
```

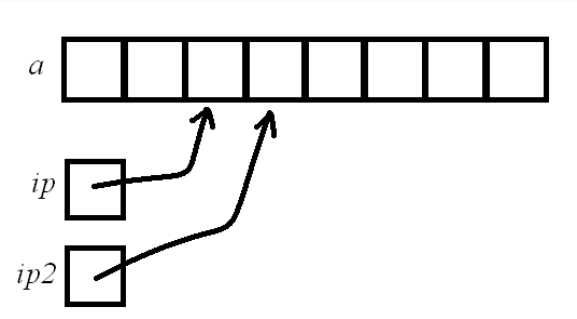


Pointer Arithmetic

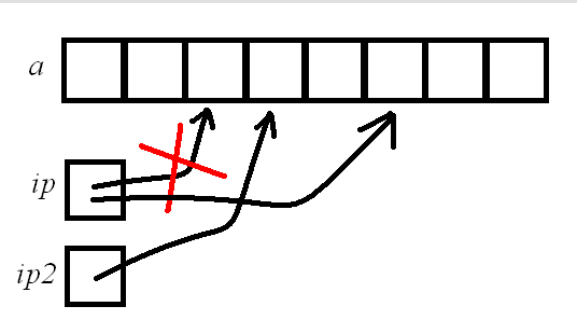
```
int *ip, *ip2;
int a[10];
ip = &a[3];
```




```
ip2 = ip + 1;
```




```
ip+=3;
```



Arithmetic cont.

```
ip2 = ip1 + 3;
```

```
ip2 - ip1 = 3
```

```
int array1[10], array2[10];
int *ip1, *ip2 = array2;
int *ep = array1;
for(ip1 = array1; ip1 < ep; ip1++)
    *ip2++ = *ip1;
```

Arithmetic cont.

```
ip2 = ip1 + 3;
```

```
ip2 - ip1 = 3
```

```
int array1[10], array2[10];
int *ip1, *ip2 = array2;
int *ep = array1;
for(ip1 = array1; ip1 < ep; ip1++)
    *ip2++ = *ip1;
```

Arithmetic cont.

```
ip2 = ip1 + 3;
```

```
ip2 - ip1 = 3
```

```
int array1[10], array2[10];
int *ip1, *ip2 = array2;
int *ep = array1;
for(ip1 = array1; ip1 < ep; ip1++)
    *ip2++ = *ip1;
```

Example

Pointer arithmetic – copy arrays

Void*

Generic pointer

`void *` is a generic pointer. Any pointer can be cast to void * and back again without loss of information

Pointer Arithmetic Summary

- void * (pointer to a void) is the generic pointer (replacing char *)
- Legal: add/sub a pointer and an integer, subtracting and comparing 2 pointers to members of the same array, and assigning or comparing to zero.
- Illegal add, multiply or divide 2 pointers, or assign one type to another type except void * without a cast.
- Any pointer can be cast to void * and back again without loss of information (used for pointer argument).

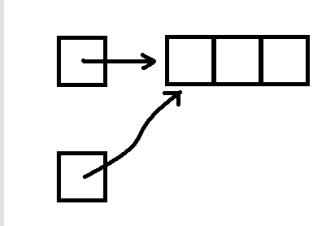
Plan

Functions

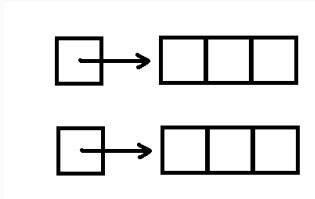
- Arrays passed to a functions are passed by reference.
- The name of the array is a pointer to its first element
- `strcpy(char dest[], char src[]);`
- Note that does not copy the array in the function call, just a reference to it.

Deep copy vs. Shallow copy

```
int a[3]={1,2,3}, *ip;
ip = a;
```

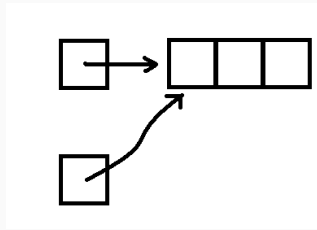


```
int a[3]={1,2,3}, ip[3];
ip[0] = a[0];
ip[1] = a[1];
ip[2] = a[2];
```

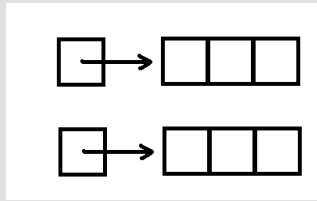


Deep copy vs. Shallow copy

```
int a[3]={1,2,3}, *ip;
ip = a;
```



```
int a[3]={1,2,3}, ip[3];
ip[0] = a[0];
ip[1] = a[1];
ip[2] = a[2];
```



Plan

Multidimensional arrays

Declaration

```
int a[3][3];
```

Declaration+Initialization

```
int a[3][3] = {
{1,2,3},
{4,5,6},
{7,8,9}};
```

Declaration+Initialization

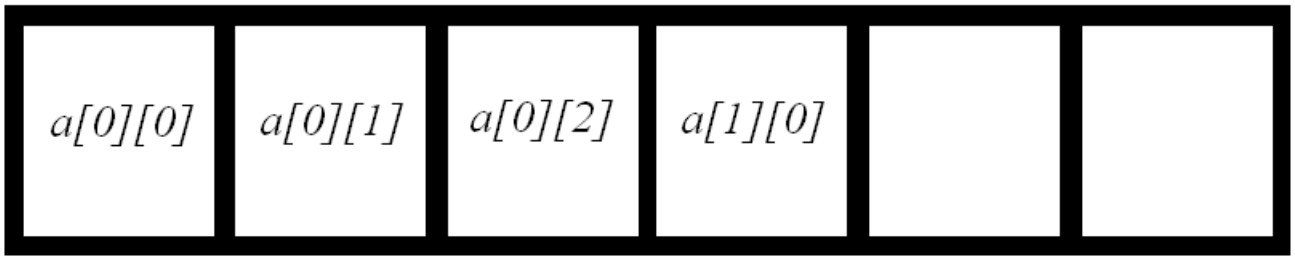
```
int a[][3] = {
{1,2,3},
{4,5,6},
{7,8,9}};
```

Wrong!!!

```
int a[][] = {
{1,2,3},
{4,5,6},
{7,8,9}};
```

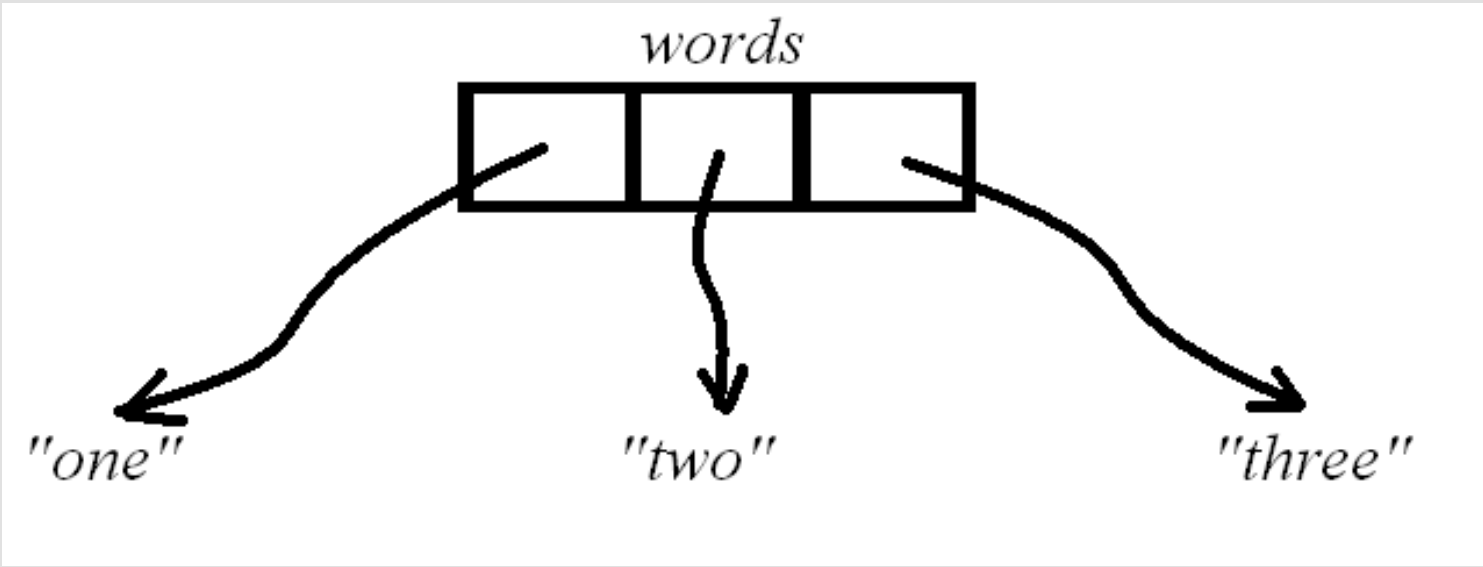
Multidimensional Arrays

- Multi-dimensional arrays are array of arrays
- For the previous example, m[0] is a pointer to the first row.



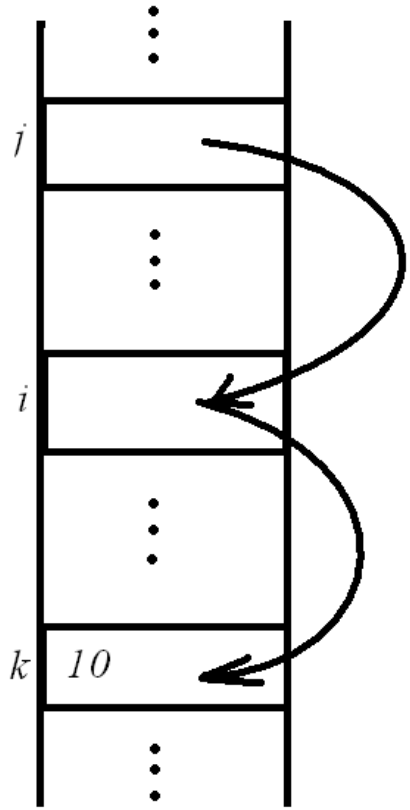
Array of pointers


```
char *words[]={"one", "two", "three"};
```



Pointers to Pointers

```
int **j;
int *i;
int k=10;
i=&k;
j=&i;
```



Pointers vs. Arrays What's the difference?

Array of pointers

```
char *words[]={"one", "two", "three"};
```

Array of arrays

```
char words[][10]={"one", "two", "three"};
```

Pointer to Whole Array

```
Char (*p2)[100];
char name[100];
char *p1;
p1=name;
p2=name; /* What is the difference?
          * Consider p1+1 and p2+1*/
```

Plan

int argc, char*argv[]

```
main(int argc, char*argv[])
```

- `argc` is the number of arguments
- `argv` is a pointer to the array containing the arguments.
- `argv[0]` is a pointer to a string with the program name

Example

Print the commandline arguments

Pointers to functions

- It is possible to assign a pointer to a function.
- That pointer could be manipulated, assigned, placed on arrays, or passed/returned to/by functions.

Pointers to functions cnt.

```
int comp(void *, void *)
```

comp is a function
that has two void* arguments
and returns int

```
int (*comp)(void *, void *)
```

comp is a pointer to a function
that has 2 void * arguments
and returns an int

Difficult Declaration

```
int *f( );
```

```
int (*daytab)[13]
```

```
int (*pf)()
```

```
char (*(*x())[]) ( )
```

```
char **argv
```

```
char (*(*x[3]) ( ) ) [5]
```

Difficult Declaration

```
int *f( );
```

f returns a pointer to int

```
int (*pf)()
```

```
char **argv
```

```
int (*daytab)[13]
```

```
char (*(*x())[]) ( )
```

```
char (*(*x[3]) ( ) ) [5]
```

Difficult Declaration

```
int *f( );
```

f returns a pointer to int

```
int (*pf)()
```

pf is a pointer to a function that returns int

```
char **argv
```

```
int (*daytab)[13]
```

```
char (*(*x())[]) ( )
```

```
char (*(*x[3]) ( ) ) [5]
```

Difficult Declaration

```
int *f( );
```

f returns a pointer to int

```
int (*pf)()
```

pf is a pointer to a function that returns int

```
char **argv
```

argv is a pointer to pointer to char

```
int (*daytab)[13]
```

```
char (*(*x())[]) ( )
```

```
char (*(*x[3]) ( ) ) [5]
```

Difficult Declaration

```
int *f( );
```

f returns a pointer to int

```
int (*pf)()
```

pf is a pointer to a function that returns int

```
char **argv
```

argv is a pointer to pointer to char

```
int (*daytab)[13]
```

daytab pointer to an array [13] of int

```
char (*(*x())[]) ( )
```

```
char (*(*x[3]) ( ) ) [5]
```

Difficult Declaration

```
int *f( );
```

f returns a pointer to int

```
int (*pf)()
```

pf is a pointer to a function that returns int

```
char **argv
```

argv is a pointer to pointer to char

```
int (*daytab)[13]
```

daytab pointer to an array [13] of int

```
char (*(*x())[]) ( )
```

x is a function returning pointer to array of pointers to function returning char

```
char (*(*x[3]) ( ) ) [5]
```

Difficult Declaration

```
int *f( );
```

f returns a pointer to int

```
int (*pf)()
```

pf is a pointer to a function that returns int

```
char **argv
```

argv is a pointer to pointer to char

```
int (*daytab)[13]
```

daytab pointer to an array [13] of int

```
char (*(*x())[]) ( )
```

x is a function returning pointer to array of pointers to function returning char

```
char (*(*x[3]) ( ) ) [5]
```

x is an array[3] of pointer to function returning pointer t array[5] of char.

What have we done today?

1. Arrays
2. Pointers
3. Pointers and Arrays
4. Pointers and Functions
5. Multidimensional arrays, Pointers to Pointers
6. Useful tricks and complicated declarations