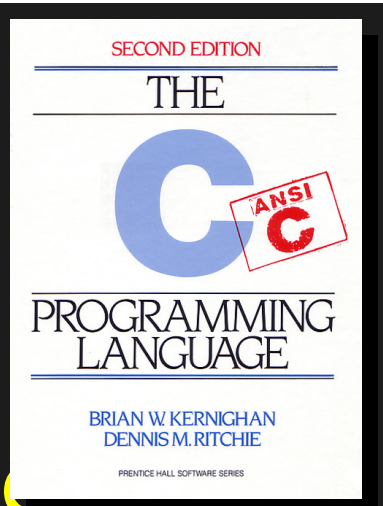


C Programming Language
Arrays and Dynamic Allocation

Math 230

Assembly Language Programming
(Computer Organization)

Thu Jan 23, 2008

Lecture 4

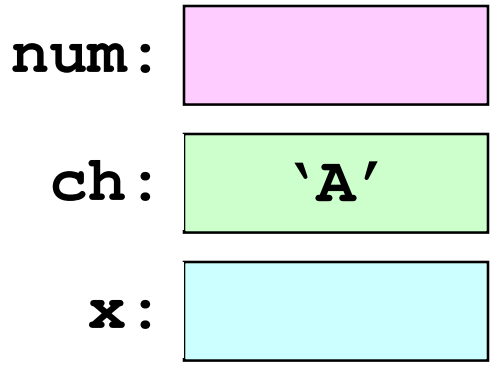
Learning Objectives

- Pointer Review, and call-by-reference
- Array Names as Pointers
  - Indirect referencing: Convert integer indexed array notation to pointer notation
- Pointer Arithmetic
  - Use pointers vs indices for array processing
- Passing and Using Array Addresses
  - pointer declaration vs standard for formal parameter
- Dynamic Memory
  - 1d, 2d

Easy Steps to Pointers

- ***Step 1***: Declare the variable to be pointed to

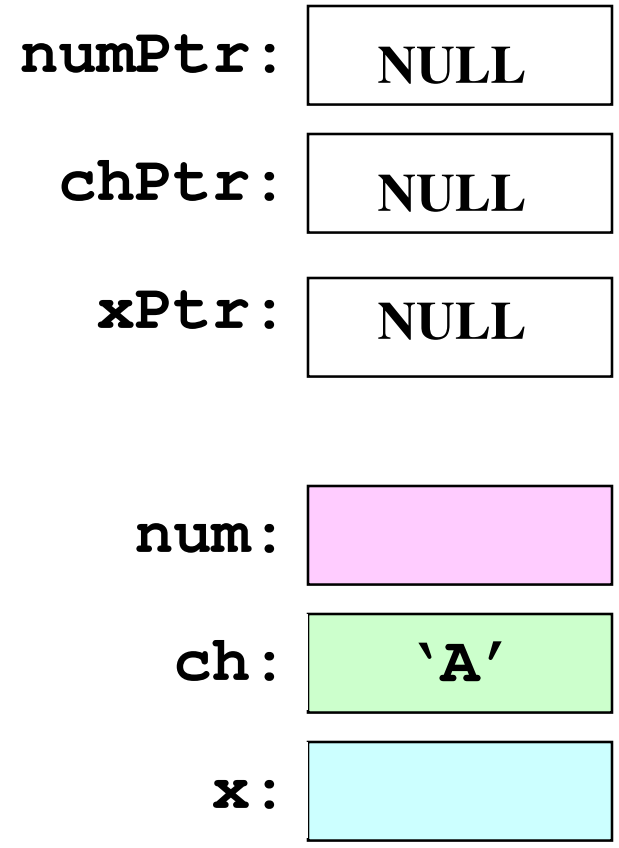
```
int   num;
char ch = 'A';
float x;
```



Easy Steps to Pointers (cont)

- ***Step 2***: Declare the pointer variable

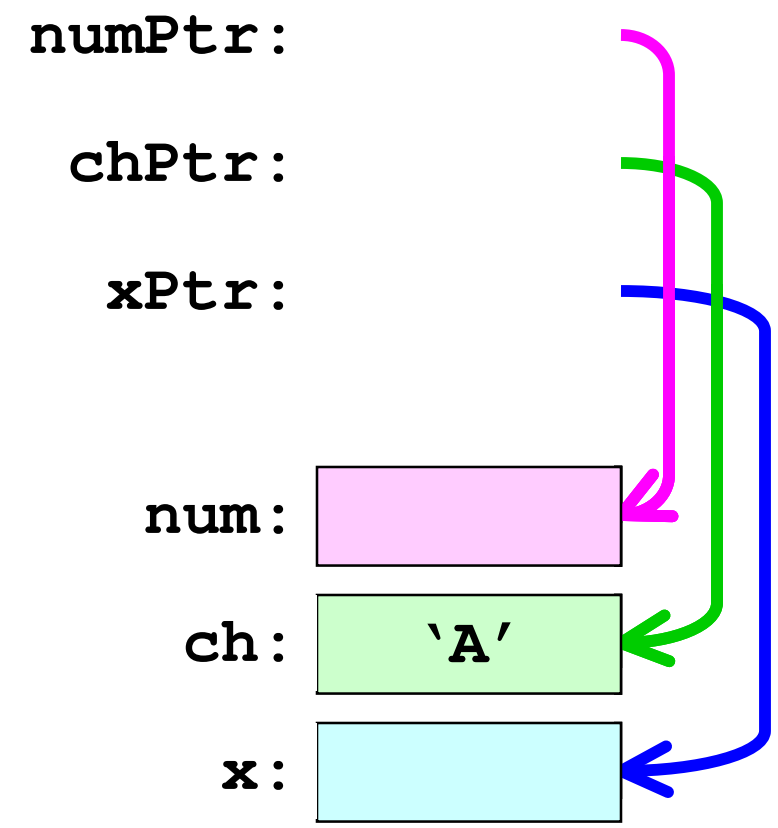
```
int   num;
char ch = 'A';
float x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

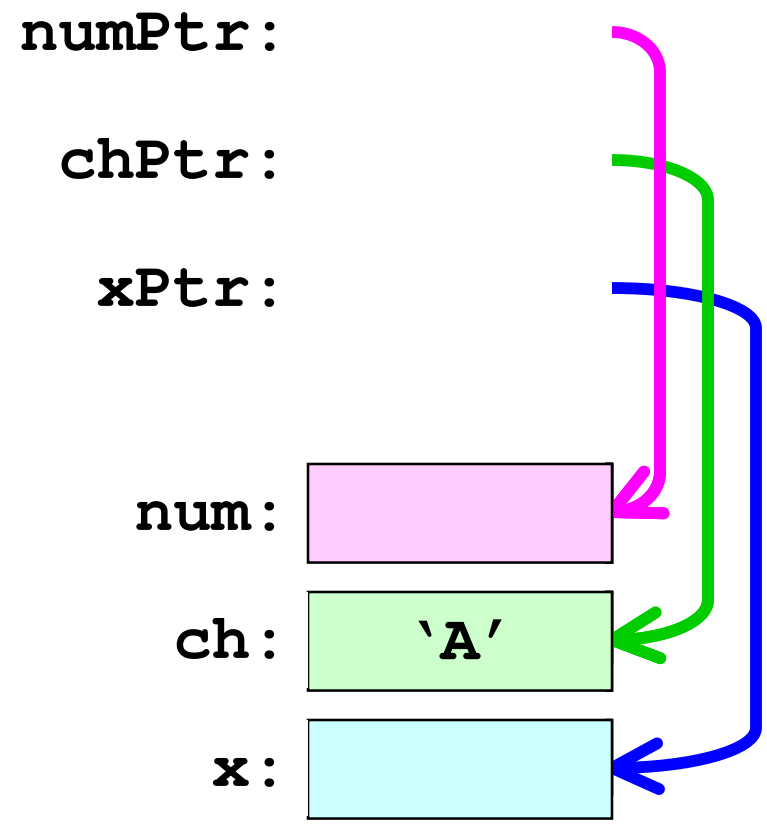
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

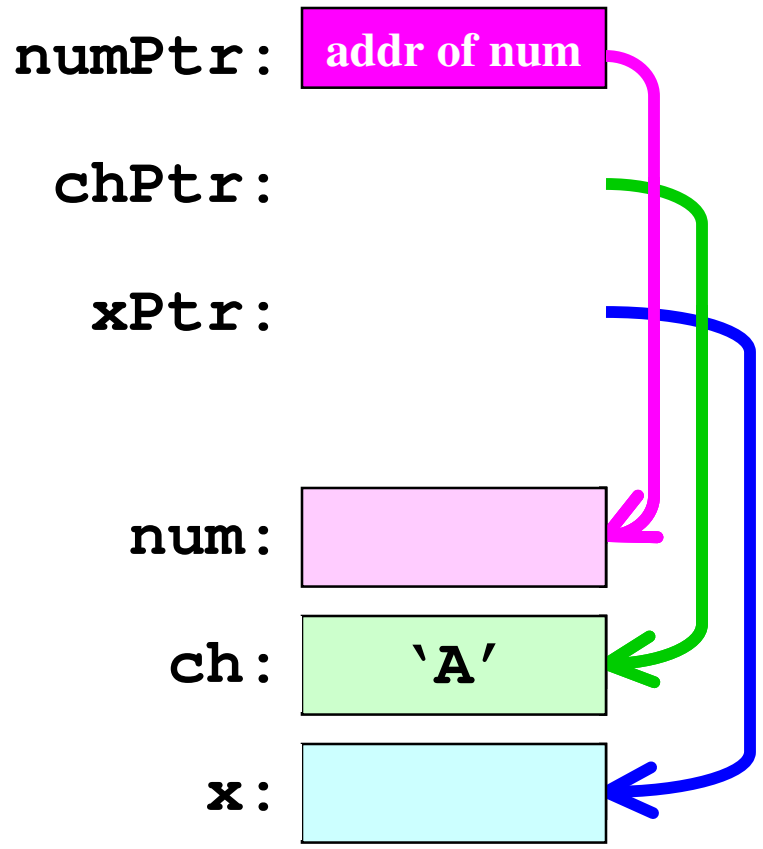
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

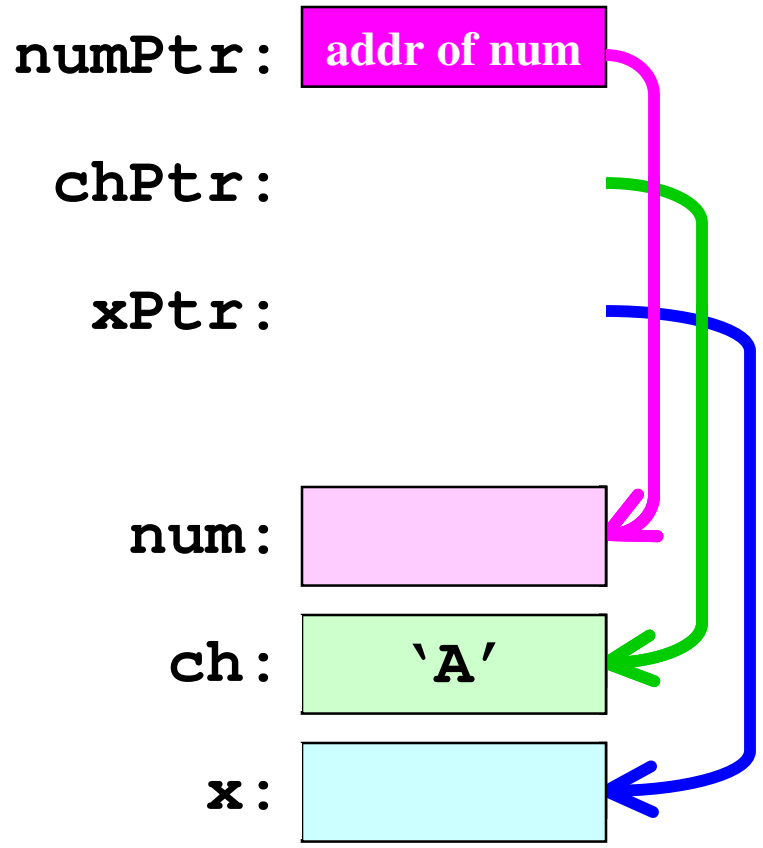
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

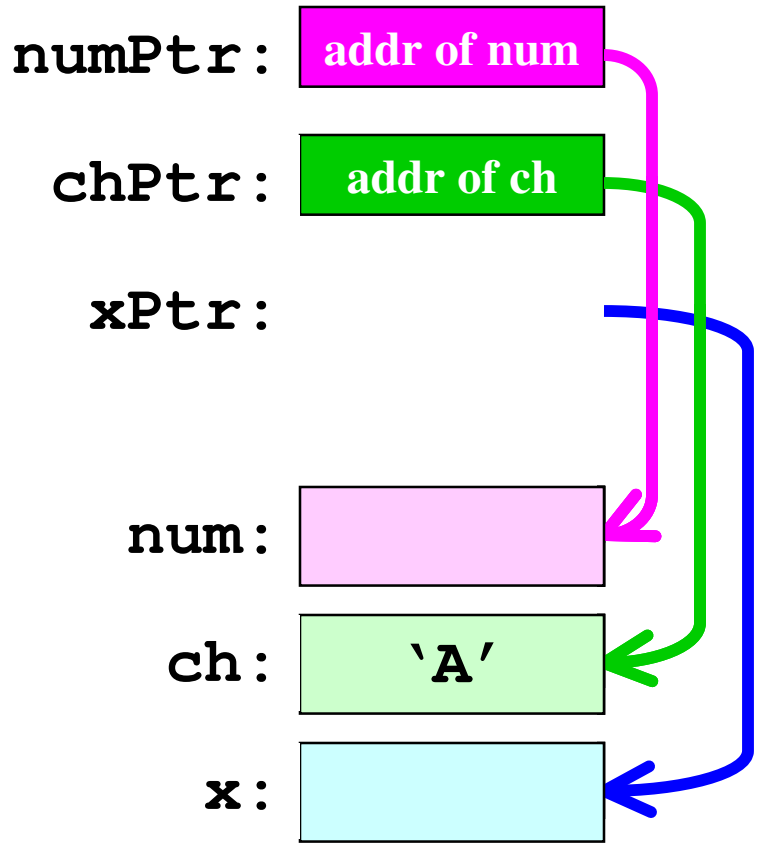
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

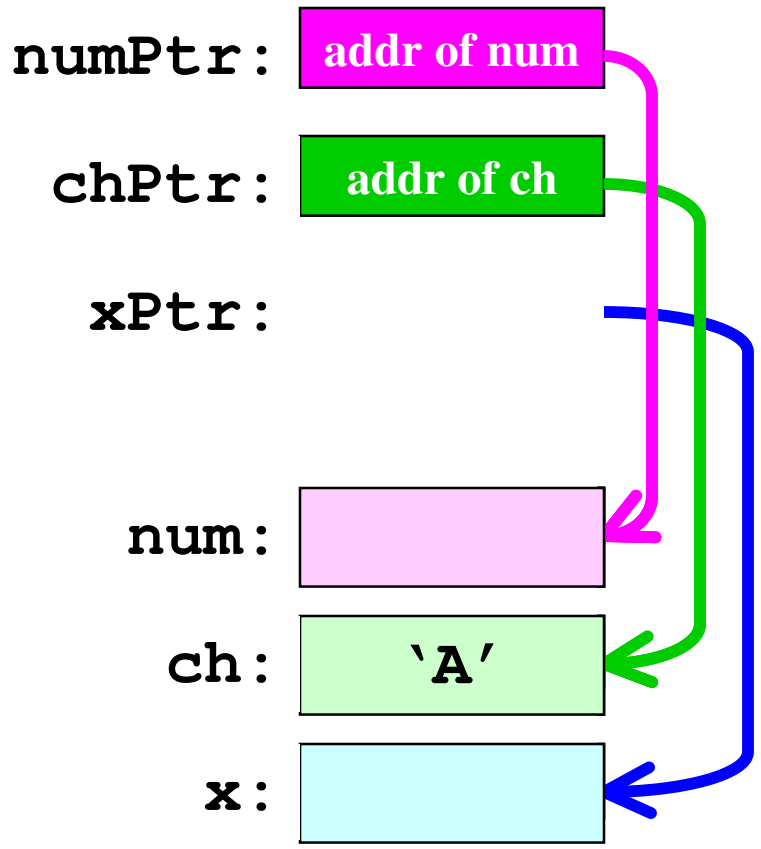
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

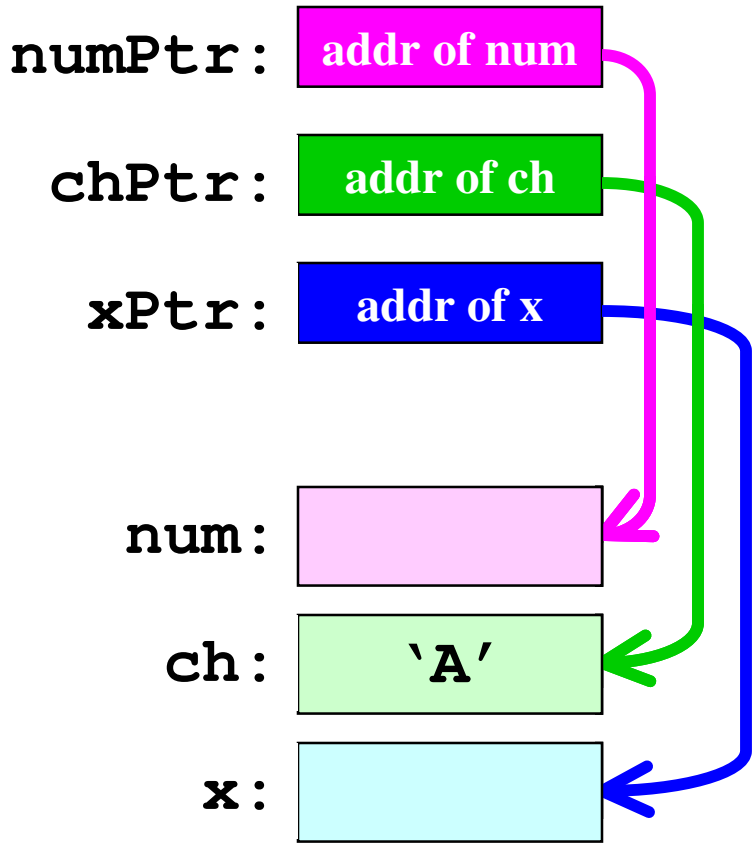
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
xPtr = &x;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

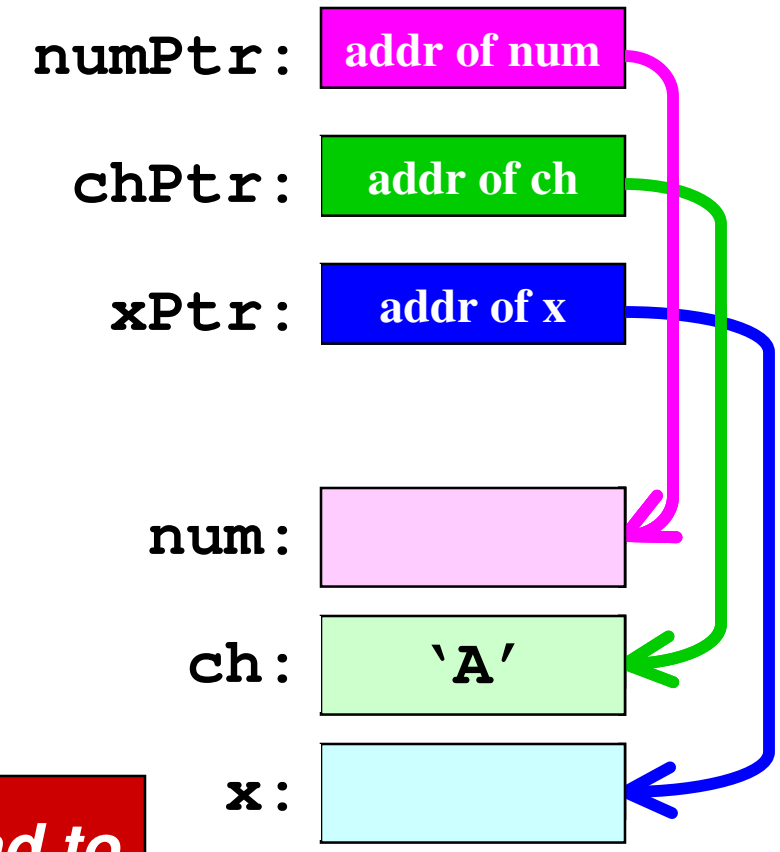
```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
xPtr = &x;
```



Easy Steps to Pointers (cont)

- ***Step 3***: Assign address of variable to pointer

```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
xPtr = &x;
```

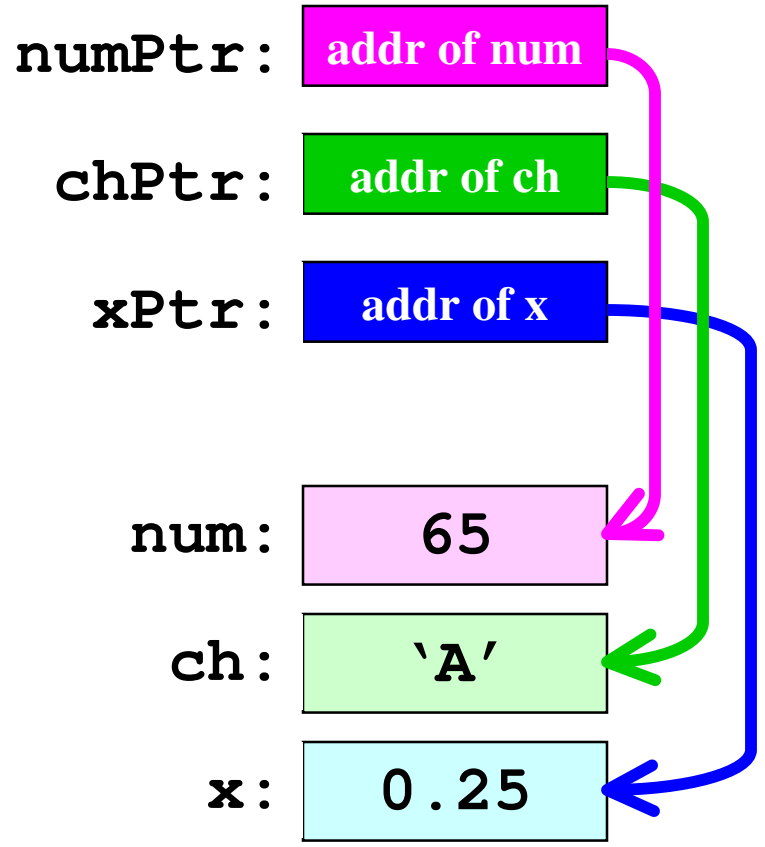


A pointer's type has to correspond to the type of the variable it points to

Easy Steps to Pointers (cont)

- ***Step 4***: De-reference the pointers

```
int    num;
char   ch = 'A';
float  x;

int*   numPtr = NULL;
char   *chPtr = NULL;
float  * xPtr = NULL;

numPtr = &num;
chPtr = &ch;
xPtr = &x;

*xPtr = 0.25;
*numPtr = *chPtr;
```



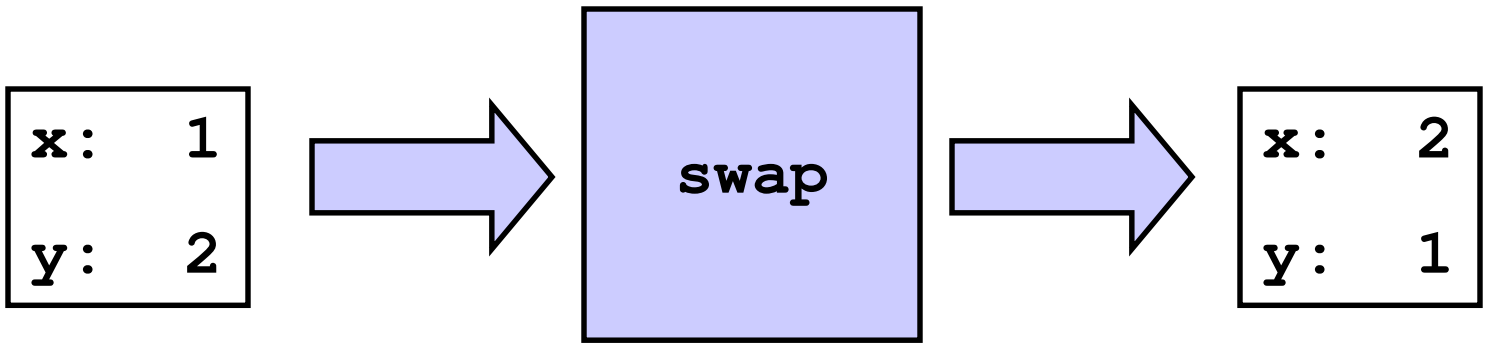
Your Turn…

C Source File

- Write a fragment of C code that does the following:
  - Declares 3 integer variables called **`a`**, **`b, and c`**.
  - Declares 3 integer pointers **`p1, p2, and p3`**.
  - Assigns the values 34, 10, and –4 to **`a`**, **`b, and c`**
  - Initializes **`p1`** with the addres of **`a`** and initializes **`p2`** with the address of **`b`**
  - Points **`p3`** to the same item pointed to by **`p2`**
  - Prints out the contents pointed to by **`p1`**, **`p2`** and **`p3`**

Pointers and Function Parameters

- ***Example:*** Function to swap the values of two variables

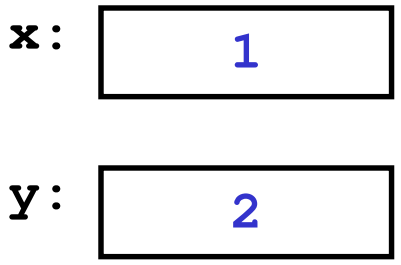


Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

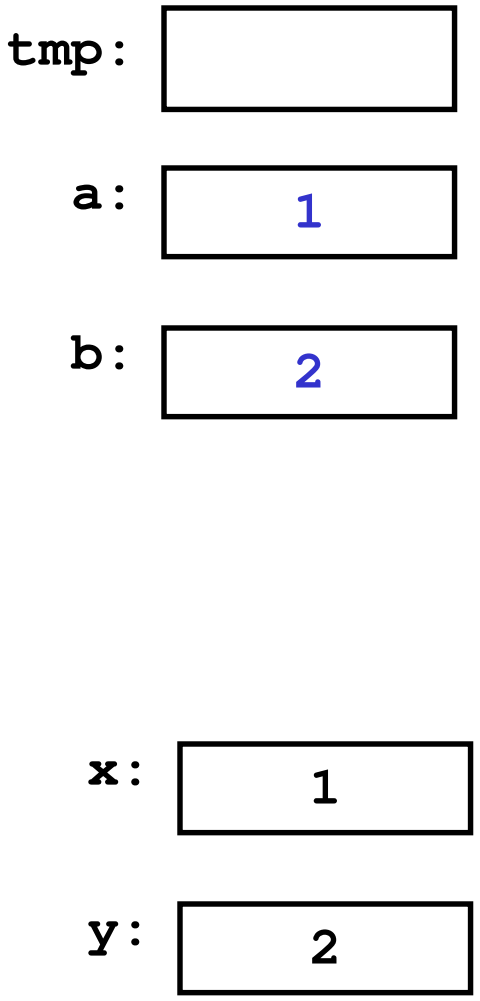
Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



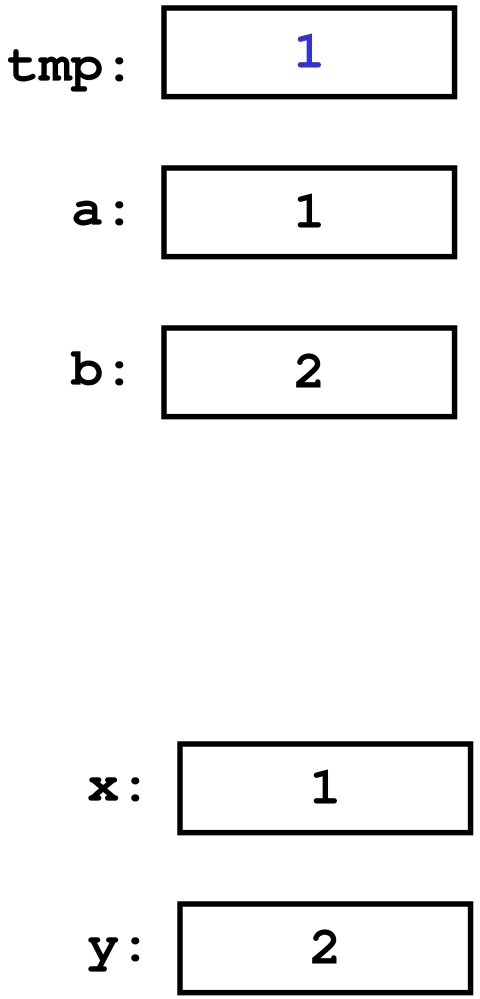
Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



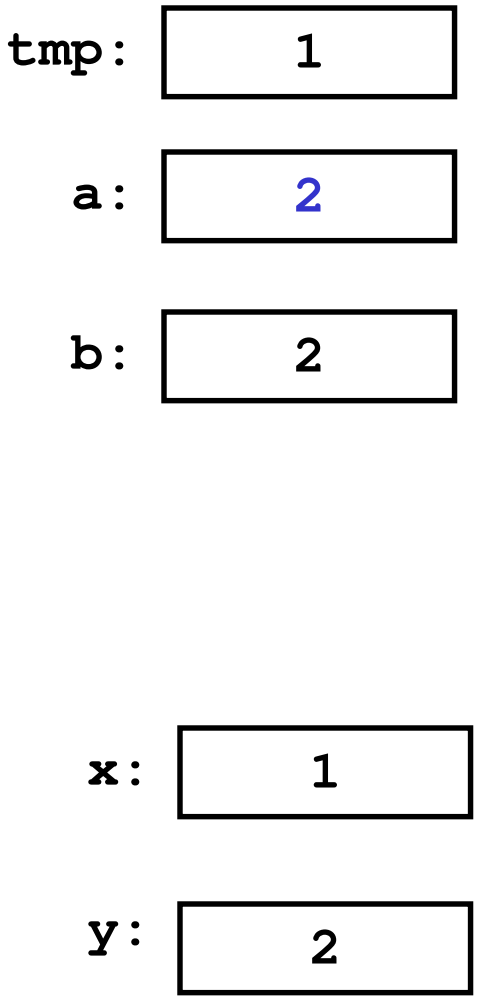
Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



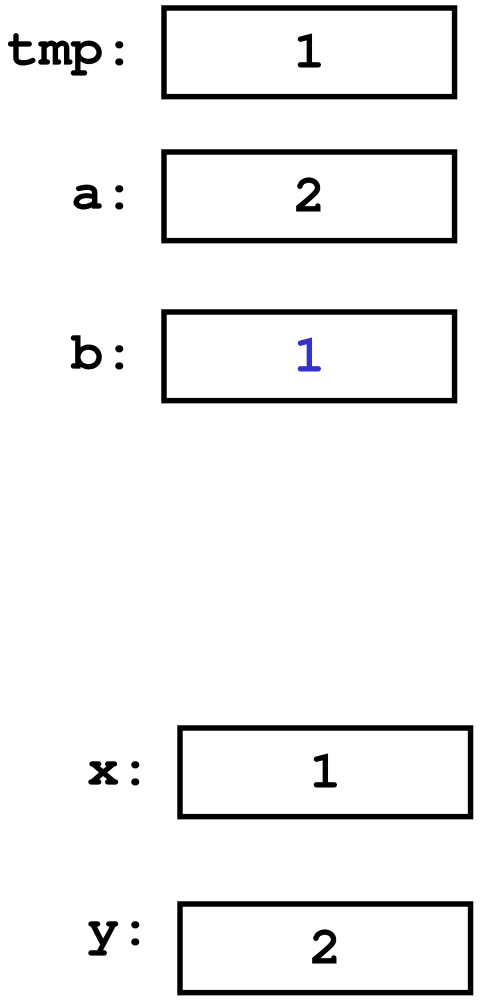
Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



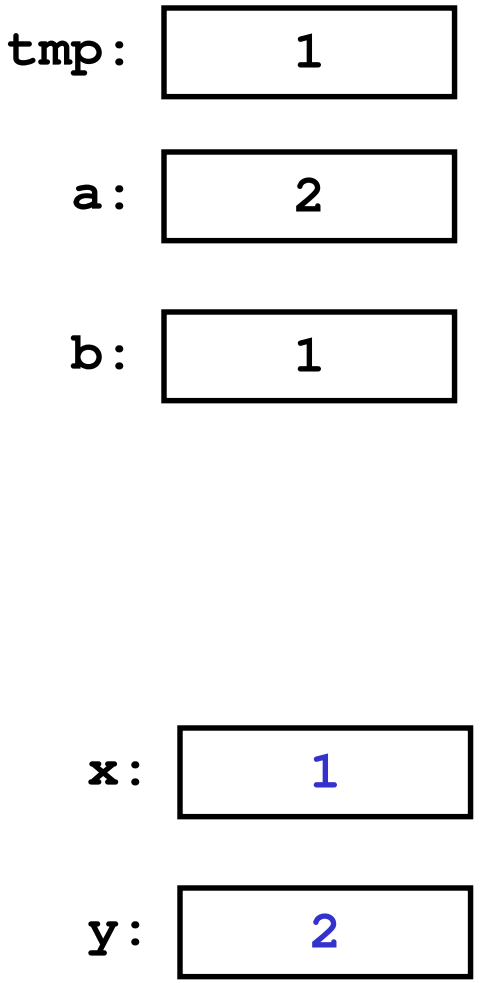
Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



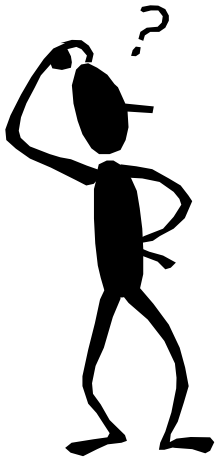
Bad swap

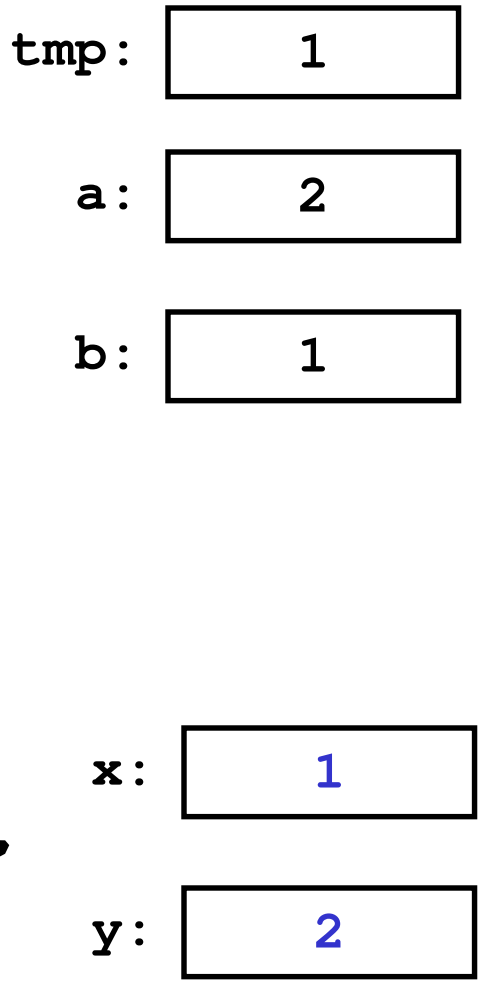
```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



Bad swap

```
#include <stdio.h>

void swap1(int a, int b)
{
    int tmp;

    tmp = a;
    a = b;
    b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap1(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```



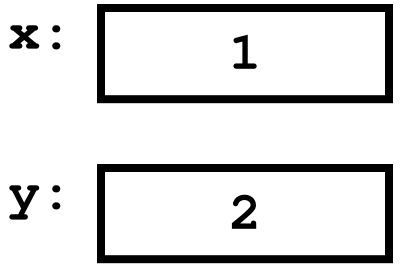


Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```

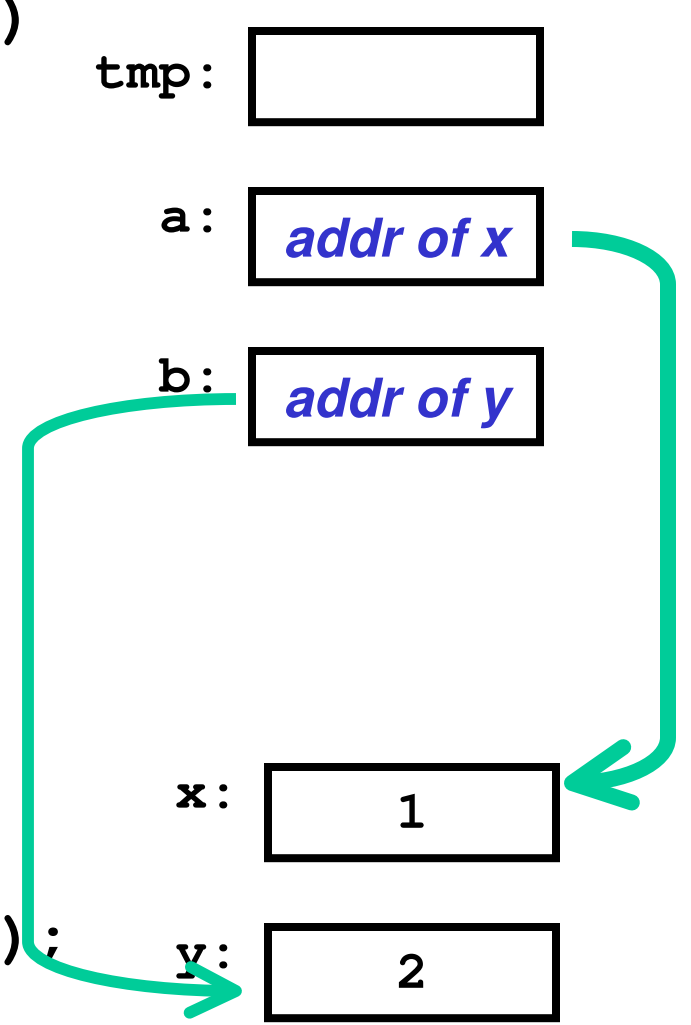
Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```



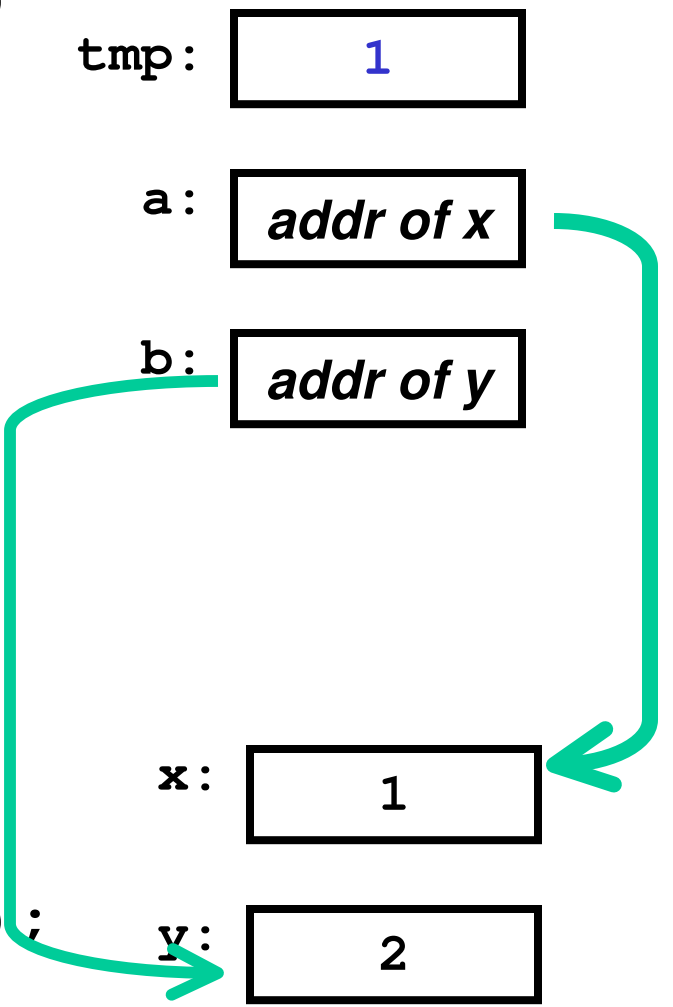
Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```



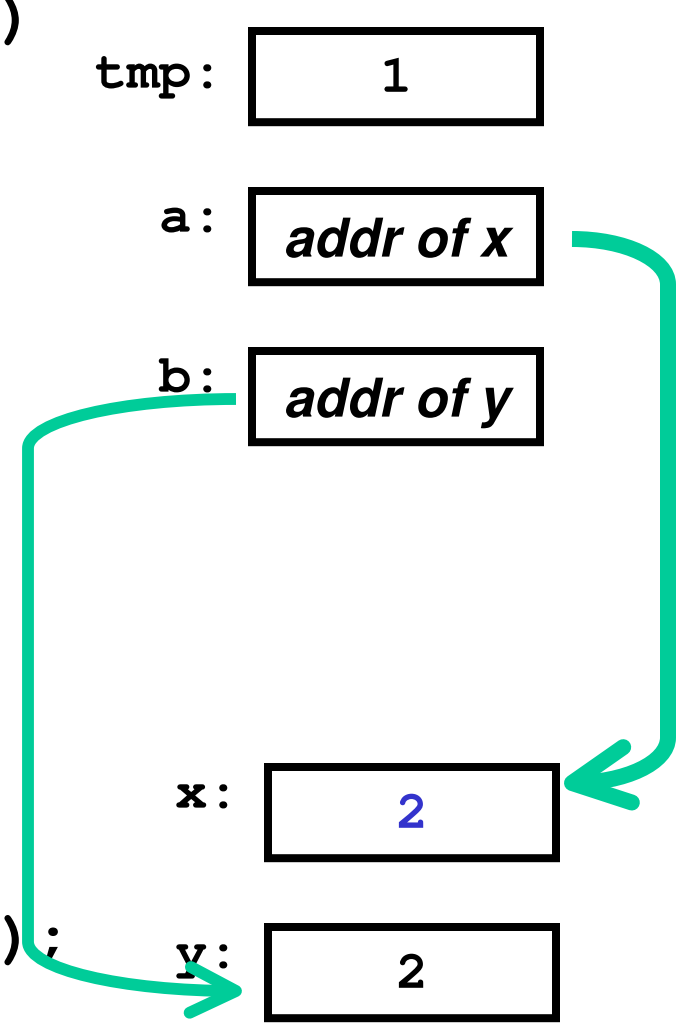
Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```



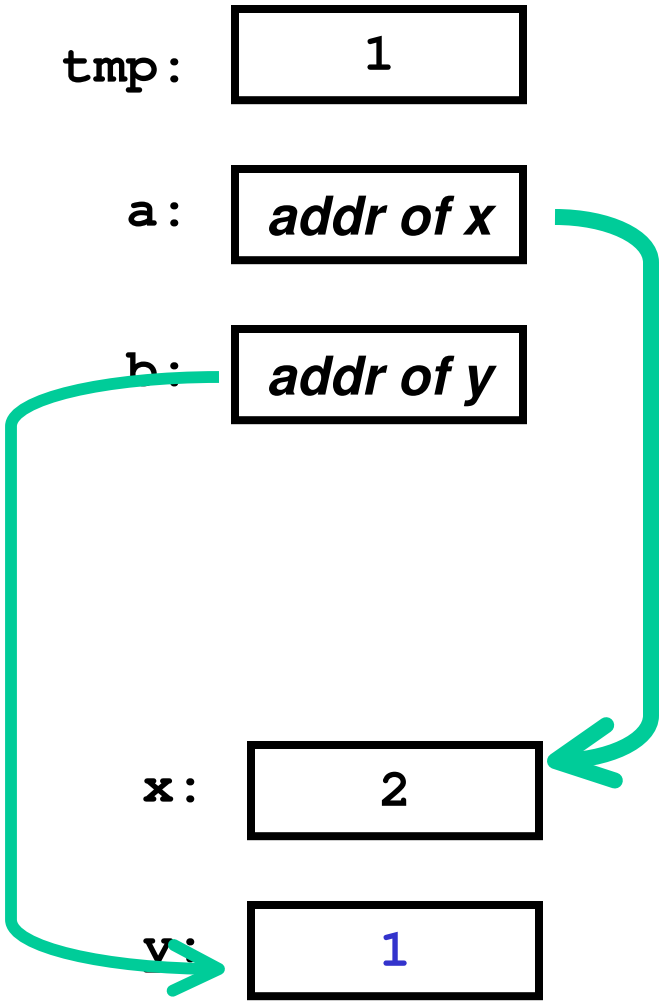
Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```



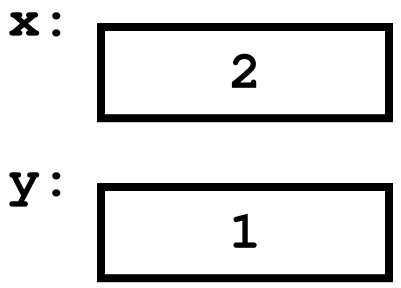
Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```



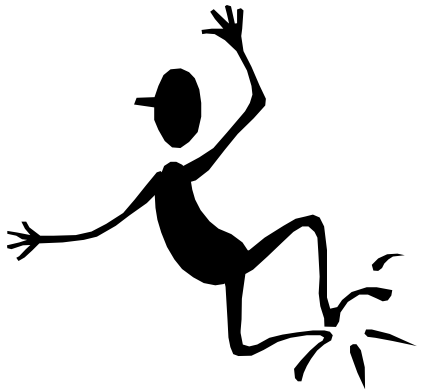
Good swap

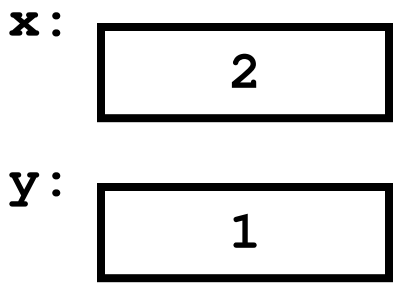
```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```



Good swap

```
#include <stdio.h>

void swap2(int* a, int* b)
{
    int tmp;

    tmp = *a;
    *a = *b;
    *b = tmp;
    return;
}

int main()
{
    int x = 1, y = 2;

    swap2(&x, &y);
    printf("%d %d\n", x, y);
    return 0;
}
```





Array Name as A Pointer

Array Name as A Pointer

- Pointers are closely associated with array names

Array Name as A Pointer

- Pointers are closely associated with array names
- Subscripts are related to the true address of an array element
  - The array element's address is determined from the address of the first element, and the size of the element

Array Name as A Pointer

- Pointers are closely associated with array names
- Subscripts are related to the true address of an array element
  - The array element's address is determined from the address of the first element, and the size of the element
- Given `grades[4],` and that an integer is stored as four bytes, the computer calculates the address as:

Array Name as A Pointer

- Pointers are closely associated with array names
- Subscripts are related to the true address of an array element
  - The array element's address is determined from the address of the first element, and the size of the element
- Given **`grades[4],`** and that an integer is stored as four bytes, the computer calculates the address as:

```
&grades[4] = &grades[0] + (4 * (4 bytes) )
```

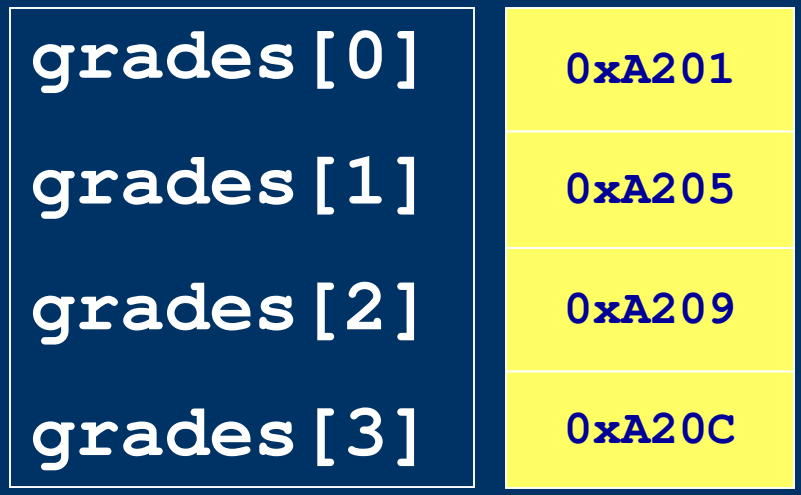
Array Name as A Pointer

- Pointers are closely associated with array names
- Subscripts are related to the true address of an array element
  - The array element's address is determined from the address of the first element, and the size of the element
- Given **`grades[4],`** and that an integer is stored as four bytes, the computer calculates the address as:

```
&grades[4] = &grades[0] + (4 * (4 bytes) )
```

address of first element
plus a 16 byte offset
(1 int = 4 bytes)

Pointer Arithmetic



Pointer Arithmetic

- Used to calculate the address of any array element

`grades[0]`	`0xA201`
`grades[1]`	`0xA205`
`grades[2]`	`0xA209`
`grades[3]`	`0xA20C`

Pointer Arithmetic

- Used to calculate the address of any array element

- If each integer takes up 4 bytes of memory, the address of **`grades[2]`** can be calculated from the address of **`grades[0]`** plus 8

`grades[0]`	`0xA201`
`grades[1]`	`0xA205`
`grades[2]`	`0xA209`
`grades[3]`	`0xA20C`

Pointer Arithmetic

- Used to calculate the address of any array element
- If each integer takes up 4 bytes of memory, the address of **`grades[2]`** can be calculated from the address of **`grades[0]`** plus 8
- The address of **`grades[3]`** is the address of **`grades[0]`** plus 12

`grades[0]`	`0xA201`
`grades[1]`	`0xA205`
`grades[2]`	`0xA209`
`grades[3]`	`0xA20C`

Using Pointers to Access Array Elements

- We can create a pointer to store the address of the first array element
  - an array of integers requires an **`int*`**
  - an array of characters requires a **`char*`**
- This allows us to access any individual element in the array using a pointer

Using Pointers to Access Array Elements

- We can create a pointer to store the address of the first array element
  - an array of integers requires an **`int*`**

```
int myArray[4];
int* arrPtr;
arrPtr = &myArray[0];
```

  - an array of characters requires a **`char*`**

- This allows us to access any individual element in the array using a pointer

Using Pointers to Access Array Elements

- We can create a pointer to store the address of the first array element
  - an array of integers requires an **`int*`**

```
int myArray[4];
int* arrPtr;
arrPtr = &myArray[0];
```

  - an array of characters requires a **`char*`**

```
char myNames[20];
char* charPtr;
charPtr = &myNames[0];
```

- This allows us to access any individual element in the array using a pointer

Walk an array: conventional

Without using pointers

```
#include <stdio.h>
int main()
{
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, grades[i]   );
  }
  return 0;
}
```

Walk an array: pointer based

Using pointers for array access

```
#include <stdio.h>
int main()
{
  int *gPtr;
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  gPtr = &grades[0];
  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, *(gPtr + i) );
  }
  return 0;
}
```

Walk an array: pointer based

Using pointers for array access

```
#include <stdio.h>
int main()
{
  int *gPtr;
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  gPtr = &grades[0];
  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, *(gPtr + i) );
  }
  return 0;
}
```

declare a pointer to an int

Walk an array: pointer based

Using pointers for array access

```
#include <stdio.h>
int main()
{
  int *gPtr;
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  gPtr = &grades[0];
  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, *(gPtr + i) );
  }
  return 0;
}
```

declare a pointer to an int

store the starting array address

Walk an array: pointer based

Using pointers for array access

```
#include <stdio.h>
int main()
{
  int *gPtr;
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  gPtr = &grades[0];
  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, *(gPtr + i) );
  }
  return 0;
}
```

declare a pointer to an int

store the starting array address

(gPtr + 1) = &grades[1]

Walk an array: pointer based

Using pointers for array access

```
#include <stdio.h>
int main()
{
  int *gPtr;
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  gPtr = &grades[0];
  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, *(gPtr + i) );
  }
  return 0;
}
```

declare a pointer to an int

store the starting array address

(gPtr + 1) = &grades[1]

(gPtr + 2) = &grades[2]

Walk an array: pointer based

Using pointers for array access

```
#include <stdio.h>
int main()
{
  int *gPtr;
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  gPtr = &grades[0];
  for (i = 0; i <= 4; ++i){
    printf("\nElement %d is %d", i, *(gPtr + i) );
  }
  return 0;
}
```

declare a pointer to an int

store the starting array address

(gPtr + 1) = &grades[1]

(gPtr + 2) = &grades[2]

(gPtr + 3) = &grades[3]

Discussion on Examples – More Pointer Math

Discussion on Examples – More Pointer Math

- The second example, shows how the computer internally accesses array elements

Discussion on Examples – More Pointer Math

- The second example, shows how the computer internally accesses array elements
- Subscripts are automatically converted to their equivalent pointer

Discussion on Examples – More Pointer Math

- The second example, shows how the computer internally accesses array elements
- Subscripts are automatically converted to their equivalent pointer
- The expression **`(gPtr + i)`** calculates the address
  - **`*(gPtr + i)`** is used to "dereference"

Discussion on Examples – More Pointer Math

- The second example, shows how the computer internally accesses array elements
- Subscripts are automatically converted to their equivalent pointer
- The expression **`(gPtr + i)`** calculates the address
  - **`*(gPtr + i)`** is used to "dereference"
- Note that we are using an integer value in the addition

Discussion on Examples – More Pointer Math

- The second example, shows how the computer internally accesses array elements
- Subscripts are automatically converted to their equivalent pointer
- The expression **`(gPtr + i)`** calculates the address
  - **`*(gPtr + i)`** is used to "dereference"
- Note that we are using an integer value in the addition
- The offset added to **`gPtr`** is automatically scaled
  - +1 = 1 integer "jump" or
  - +1 = 1 float "jump" etc.

More Pointer Math and Array Equivalency

More Pointer Math and Array Equivalency

- Are the parentheses necessary in the expression `*(gPtr + 3)`?

More Pointer Math and Array Equivalency

- Are the parentheses necessary in the expression `*(gPtr + 3)`?

- Yes. Note the difference between:
  `*(gPtr + 3)` and `*gPtr+3`
  - BIG difference. The parentheses *are* required

More Pointer Math and Array Equivalency

- Are the parentheses necessary in the expression **`*(gPtr + 3)`**?

- Yes. Note the difference between: **`*(gPtr + 3)`** and **`*gPtr+3`**
  - BIG difference. The parentheses *are* required

- Finally, the expression **`grades[i]`** can always be replaced with **`*(grades + i)`**

Pointer Constants: Array Names

Pointer Constants: Array Names

Assume the following declarations have been made:

Pointer Constants: Array Names

Assume the following declarations have been made:

```
int a[10];
```

Pointer Constants: Array Names

Assume the following declarations have been made:

```
int a[10];
int *pa;
```

Pointer Constants: Array Names

Assume the following declarations have been made:

```
int a[10];
int *pa;
```

Important difference between an array name and a pointer variable:

Pointer Constants: Array Names

Assume the following declarations have been made:

```
int a[10];
int *pa;
```

Important difference between an array name and a pointer variable:

- A pointer is a variable so it's legal to use
  - `pa = a;`
  - `pa++;`

Pointer Constants: Array Names

Assume the following declarations have been made:

```
int a[10];
int *pa;
```

Important difference between an array name and a pointer variable:

- A pointer is a variable so it's legal to use
  - `pa = a;`
  - `pa++;`

- An array name is a constant, not a variable. ILLEGAL usage
  - `a = pa;  //Incompatible types int* and int[5]`
  - `a++;  //Not allowed`
  - `pa = &a[0];  //Both types must be same`

Pointer Constants

Pointer Constants

- When arrays are created, an internal "*pointer constant*" is automatically created

Pointer Constants

- When arrays are created, an internal "*pointer constant*" is automatically created

- This pointer constant stores the *starting address* of the array, i.e., the first element

Pointer Constants

- When arrays are created, an internal "*pointer constant*" is automatically created
- This pointer constant stores the *starting address* of the array, i.e., the first element
- What happens when an array is declared?
  1. The array name becomes the name of a pointer constant
  2. The first array element is stored at the pointer's address
  3. Storage is created for the appropriate number of the indicated variable type

Init'z as array, work as pointer

```
#include <stdio.h>
int main()
{
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  for (i = 0; i <= 4; ++i)
    printf("\nElement %d is %d", i, *(grades + i) );

  return 0;
}
```

Init'z as array, work as pointer

A 'constant' pointer named **grades** is *automatically* generated

```
#include <stdio.h
int main()
{
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  for (i = 0; i <= 4; ++i)
    printf("\nElement %d is %d", i, *(grades + i) );

  return 0;
}
```

Init'z as array, work as pointer

```
#include <stdio.h
int main()
{
  int i;
  int grades[] = {98, 87, 92, 79, 85};

  for (i = 0; i <= 4; ++i)
    printf("\nElement %d is %d", i, *(grades + i) );

  return 0;
}
```

A 'constant' pointer named **grades** is *automatically* generated

Pointer notation used although no explicit pointer declared!

Pointer Arithmetic – Marching Through Arrays

- You can use pointers to index through arrays by pointing to each element in turn
- Given that p points to wilma[i], p + k points to wilma[i + k]

```
int wilma[4], i,*p, x;
wilma[4]={21, 9, 19, 6};


p = &wilma[0];
x = *p;
x = *(p+1);
p = p + 1;
p++;
```

Pointer Arithmetic – Marching Through Arrays

- You can use pointers to index through arrays by pointing to each element in turn
- Given that p points to wilma[i], p + k points to wilma[i + k]

```
int wilma[4], i,*p, x;
wilma[4]={21, 9, 19, 6};


p = &wilma[0];   address of wilma[0] assigned to p
x = *p;
x = *(p+1);
p = p + 1;
p++;
```

Pointer Arithmetic – Marching Through Arrays

- You can use pointers to index through arrays by pointing to each element in turn
- Given that p points to wilma[i], p + k points to wilma[i + k]

```
int wilma[4], i,*p, x;
wilma[4]={21, 9, 19, 6};


p = &wilma[0];
x = *p;
x = *(p+1);
p = p + 1;
p++;
```

```
address of wilma[0] assigned to p
wilma[0] is assigned to x
```

Pointer Arithmetic – Marching Through Arrays

- You can use pointers to index through arrays by pointing to each element in turn
- Given that p points to wilma[i], p + k points to wilma[i + k]

```
int wilma[4], i,*p, x;
wilma[4]={21, 9, 19, 6};


p = &wilma[0];      address of wilma[0] assigned to p
x = *p;             wilma[0] is assigned to x
x = *(p+1);         wilma[1] is assigned to x
p = p + 1;
p++;
```

Pointer Arithmetic – Marching Through Arrays

- You can use pointers to index through arrays by pointing to each element in turn
- Given that p points to wilma[i], p + k points to wilma[i + k]

```
int wilma[4], i,*p, x;
wilma[4]={21, 9, 19, 6};


p = &wilma[0];      address of wilma[0] assigned to p
x = *p;             wilma[0] is assigned to x
x = *(p+1);         wilma[1] is assigned to x
p = p + 1;          &wilma[1] is assigned to p
p++;
```

Pointer Arithmetic – Marching Through Arrays

- You can use pointers to index through arrays by pointing to each element in turn
- Given that p points to wilma[i], p + k points to wilma[i + k]

```
int wilma[4], i,*p, x;
wilma[4]={21, 9, 19, 6};


p = &wilma[0];      address of wilma[0] assigned to p
x = *p;             wilma[0] is assigned to x
x = *(p+1);         wilma[1] is assigned to x
p = p + 1;          &wilma[1] is assigned to p
p++;                p now points to wilma[2]
```

Marching Through Arrays – an example

- *p++ can be used to walk through the array pointed to by p
- Given `int wilma[4], i,*p, x;`
  - The same output is achieved by both of the following

Marching Through Arrays – an example

- *p++ can be used to walk through the array pointed to by p
- Given `int wilma[4], i,*p, x;`
  - The same output is achieved by both of the following

```
p = wilma;
for (i = 0; i < 4; i++)
  printf("%d\n", *p++);
```

Marching Through Arrays – an example

- *p++ can be used to walk through the array pointed to by p
- Given `int wilma[4], i,*p, x;`
  - The same output is achieved by both of the following

```
p = wilma;
for (i = 0; i < 4; i++)
  printf("%d\n", *p++);
```

```
for (i = 0; i < 4; i++)
  printf("%d\n", wilma[i]);
```

Array Marching – Stop Based on Address

```
#include <stdio.h>
int main()
{
  int nums[5] = {16, 54, 7, 43, -5};
  int total = 0, *nPtr;

  nPtr = nums;

  while (nPtr  <=  nums + 4)
    total += *nPtr++;

  printf("The total of the array elements is %d", total);

  return 0;
}
```

Array Marching – Stop Based on Address

```
#include <stdio.h>
int main()
{
  int nums[5] = {16, 54, 7, 43, -5};
  int total = 0, *nPtr;

  nPtr = nums;

  while (nPtr  <=  nums + 4)
    total += *nPtr++;

  printf("The total of the array elements is %d", total);

  return 0;
}
```

store address of nums[0] in nPtr

Array Marching – Stop Based on Address

```
#include <stdio.h>
int main()
{
  int nums[5] = {16, 54, 7, 43, -5};
  int total = 0, *nPtr;

  nPtr = nums;

  while (nPtr  <=  nums + 4)
    total += *nPtr++;

  printf("The total of the array elements is %d", total);

  return 0;
}
```

store address of nums[0] in nPtr

compare addresses

Functions, Pointers and Using Array Addresses

- The array address is the only actual item passed
  - What constitutes an array's address?
- The following examples study the passing of arrays and pointers

findMax routine

```
int findMax(int vals[], int numEls)
{
  int i, max = vals[0];

  for (i = 1; i < numEls; ++i)
    if (max < vals[i])
      max = vals[i];

  return(max);
}
```

pgm 8.6c findMax routine-changes

```
int findMax(int *vals, int numEls)
{
  int i, max = *vals;

  for (i = 1; i < numEls; ++i)
    if (max < *(vals + i) )
      max = *(vals + i);

  return(max);
}
```

int vals[]

max = &vals[0]

max = vals[i]