Lecture 5:

Pointers

Computer Programming 2

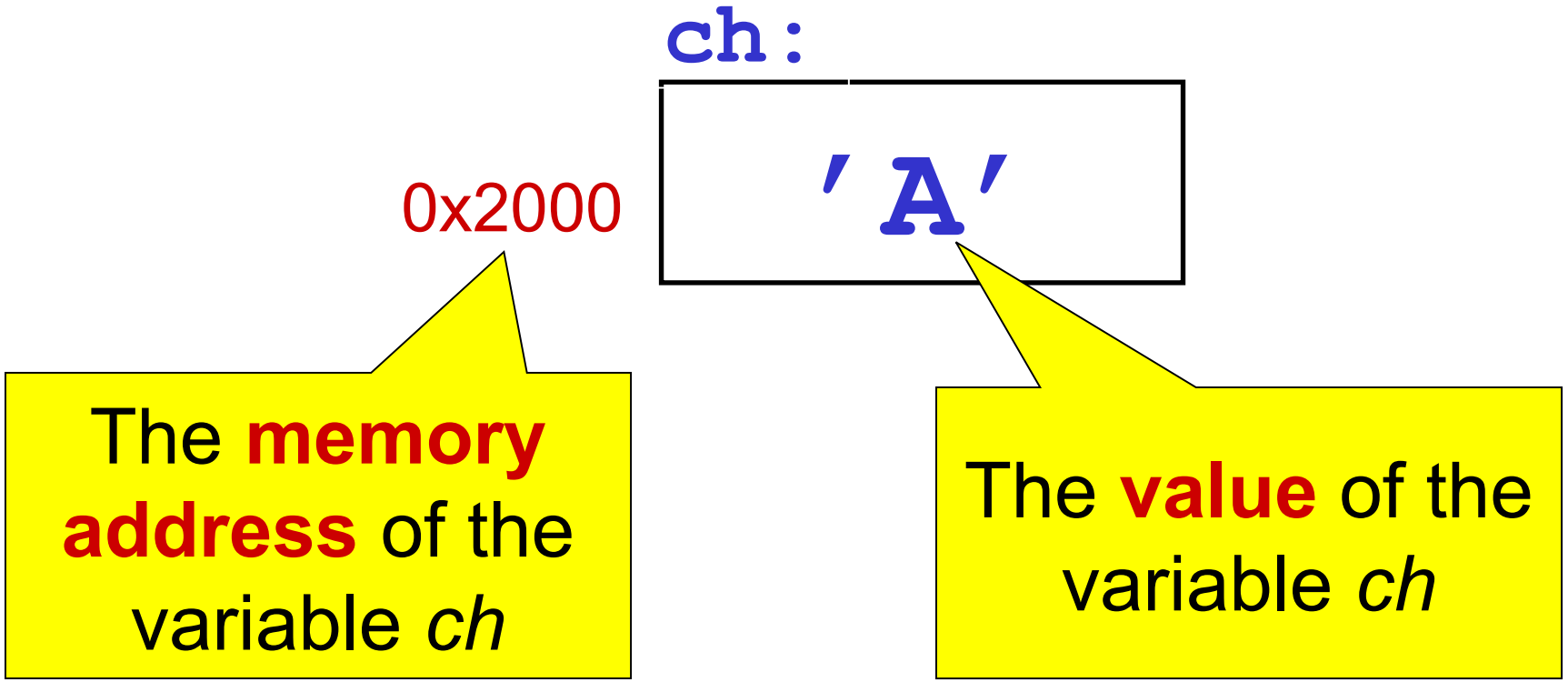
2nd Semester 2023-2024

Topics

- Introduction to pointers
- Pointers and function parameters

Memory Address of a Variable

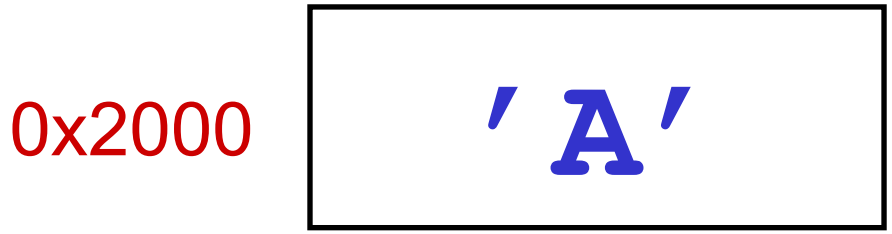
```
char ch = 'A';
```



The & Operator

- Gives the memory address of an object

```
char ch = 'A';
```



`&ch` yields the value 0x2000

- Also known as the "address operator"

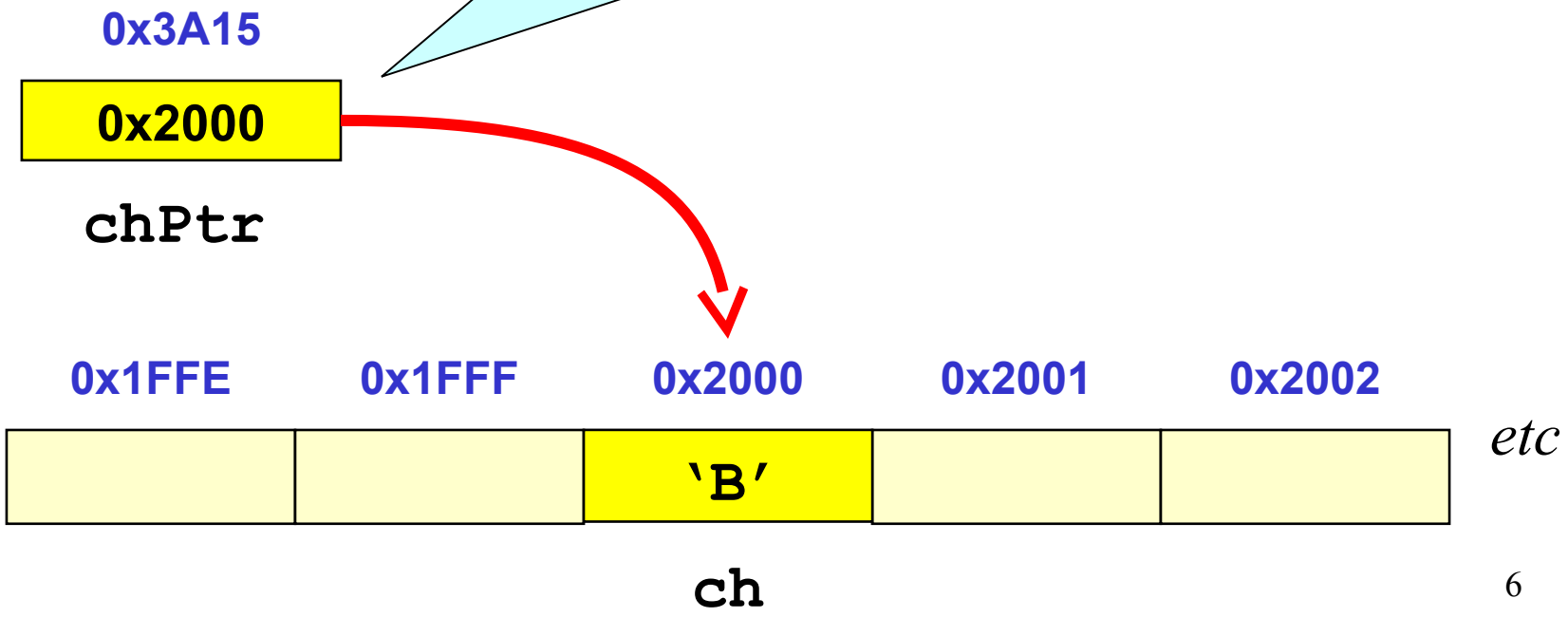
Example:

```
char ch;
printf("%p", &ch);
```

"conversion specifier" for printing a memory address

Pointers

A variable which can store the **memory address** of another variable



Pointers

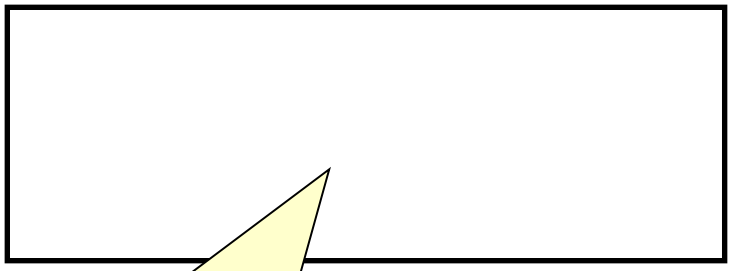
- A pointer is a variable
- Contains a memory address
- Points to a specific data type
- Pointer variables are usually named ***`varPtr`***

Example:

```
char* cPtr;
```

cPtr:

0x2004



Can store an **address** of variables of type **`char`**

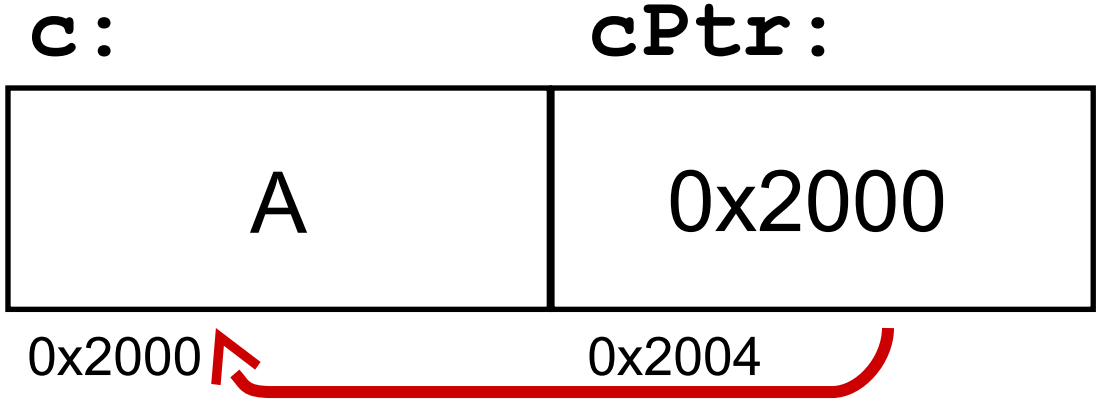
- We say *cPtr* is a ***pointer*** to char

Pointers and the & Operator

Example:

```
char c = 'A';
char *cPtr;


cPtr = &c;
```

Assigns the address of c to cPtr



Notes on Pointers

- We can have pointers to any data type

Example:

```
int*   numPtr;
float*  xPtr;
```

- The * can be anywhere between the type and the variable

Example:

```
int    *numPtr;
float * xPtr;
```

Notes on Pointers (cont)

- You can assign the address of a variable to a "compatible" pointer using the **&** operator

Example:

```
int  aNumber;
int  *numPtr;

numPtr = &aNumber;
```

- You can print the address stored in a pointer using the **%p** conversion specifier

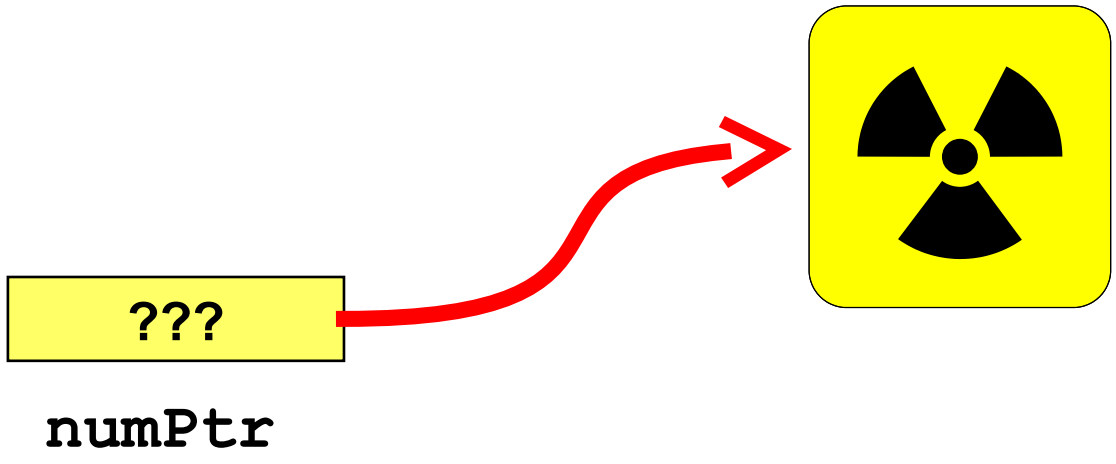
Example:

```
printf("%p", numPtr);
```

Notes on Pointers (cont)

```
int   *numPtr;
```

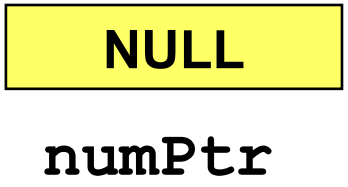
Beware of pointers which are not initialized!



Notes on Pointers (cont)

- When declaring a pointer, it is a good idea to always initialize it to **NULL** (a special pointer constant)

```
int   *numPtr = NULL;
```



The * Operator

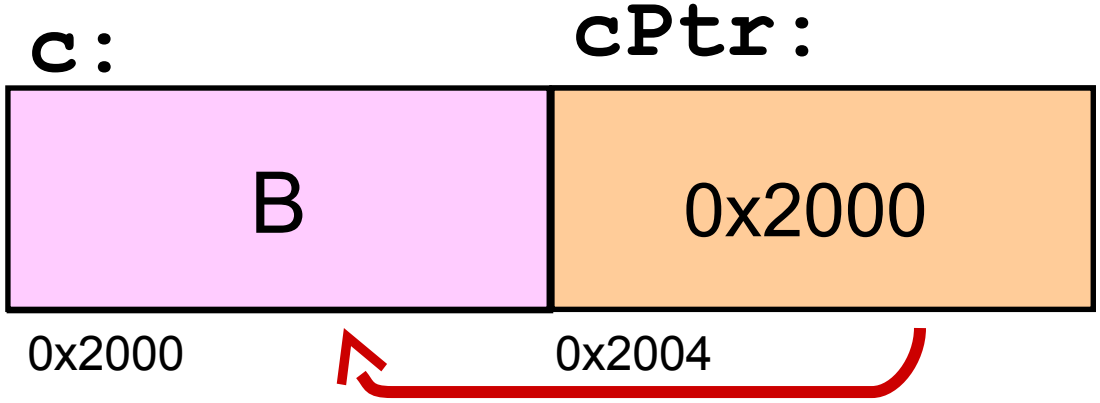
- Allows pointers to access variables they point to
- Also known as "dereferencing operator"
- Should not be confused with the * in the pointer declaration

Pointers and the * Operator

Example:

```
char c = 'A';
char *cPtr = NULL;


cPtr = &c;
*cPtr = 'B';
```

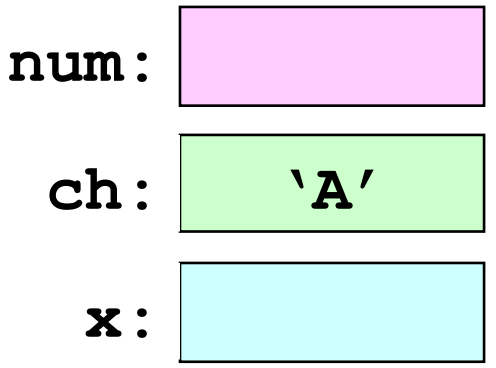
Changes the value of the variable which cPtr points to



Easy Steps to Pointers

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

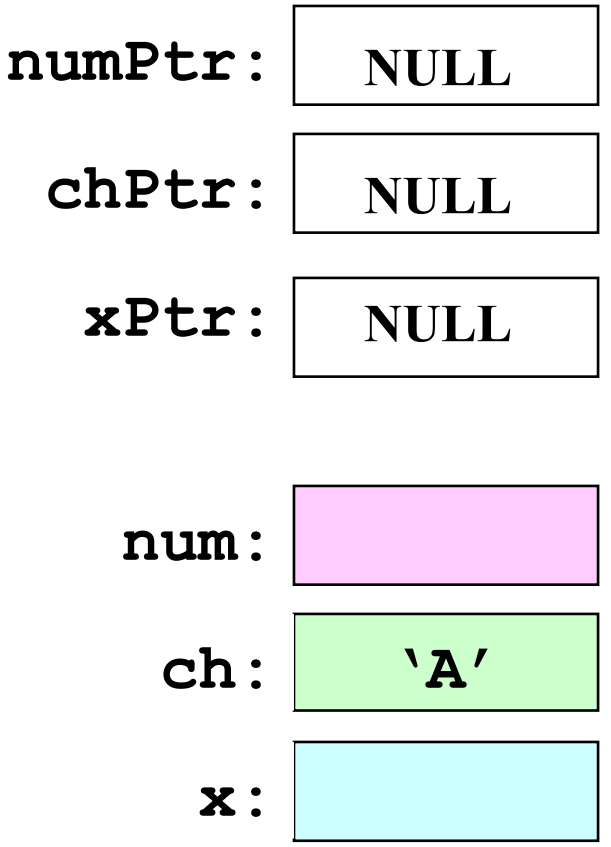
```
int  num;
char ch = 'A';
float x;
```



Easy Steps to Pointers (cont)

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

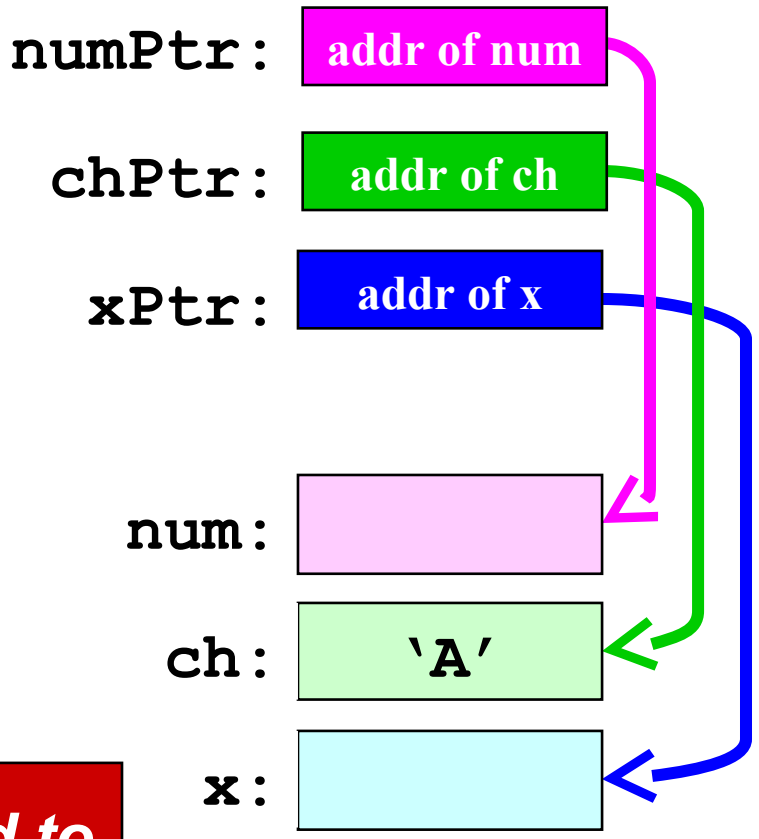
```
int  num;
char ch = 'A';
float x;

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;

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

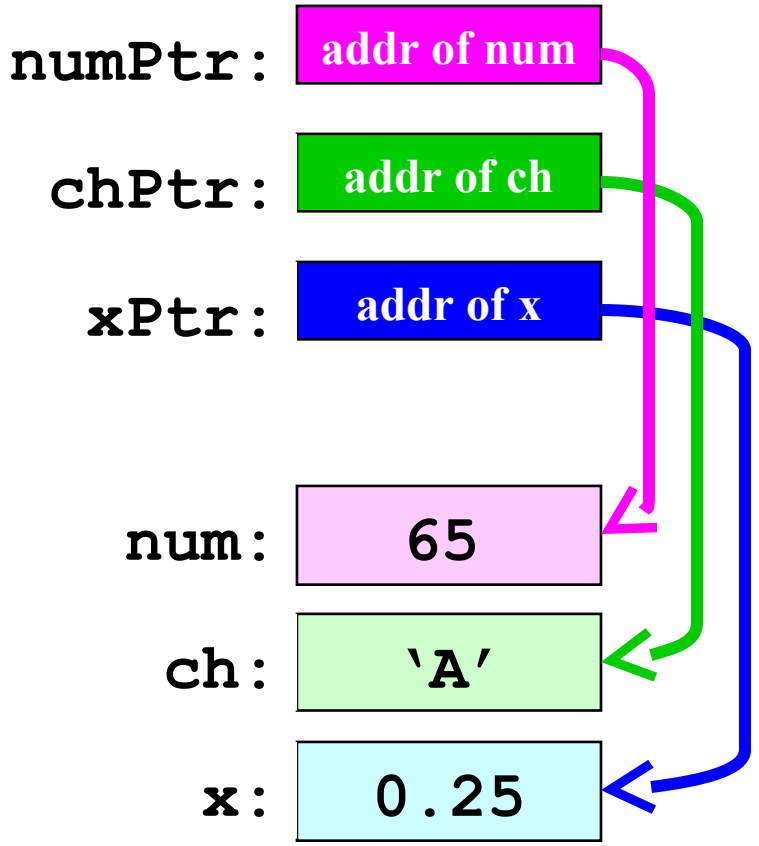


A pointer's type has to correspond to the type of the variable it points to

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;
```



Assigning to pointers

- Pointers are just normal variables, that happen to have the type “address of <some type>”
- Pointers can be assigned to the address of any variable (of the right type)
- The value of a pointer can change, just like the value of any other variable
- The value of a pointer can be manipulated, just like the value of any other variable

More on Dereferencing

- Pointers point to other variables
- We need to go *through* the pointer and find out the value of the variable it points to
- We do this by *dereferencing* the pointer, using the ***** operator
- But what is actually happening when we dereference?

Algorithm for Dereferencing

To *dereference* a pointer, e.g. use *xPtr, which means:

1. Go to xPtr
2. Take the value you find there, and use it as an address
3. Go to that address
4. Return the value you find there

Pointer examples

```
char ch;

ch = 'A';
```

0x2000

ch:

'A'

```
char* chPtr=NULL;

chPtr=&ch;
```

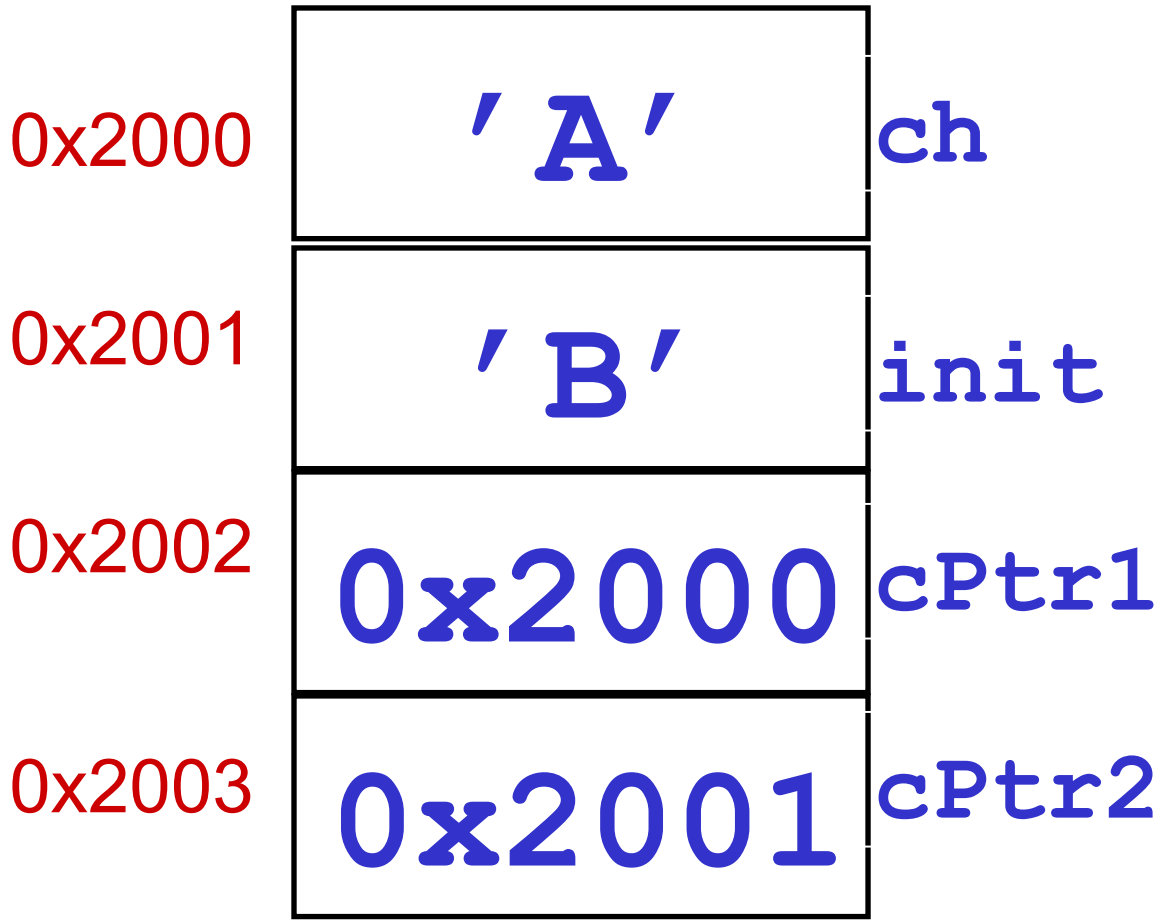
0x2004

chPtr:

0x2000

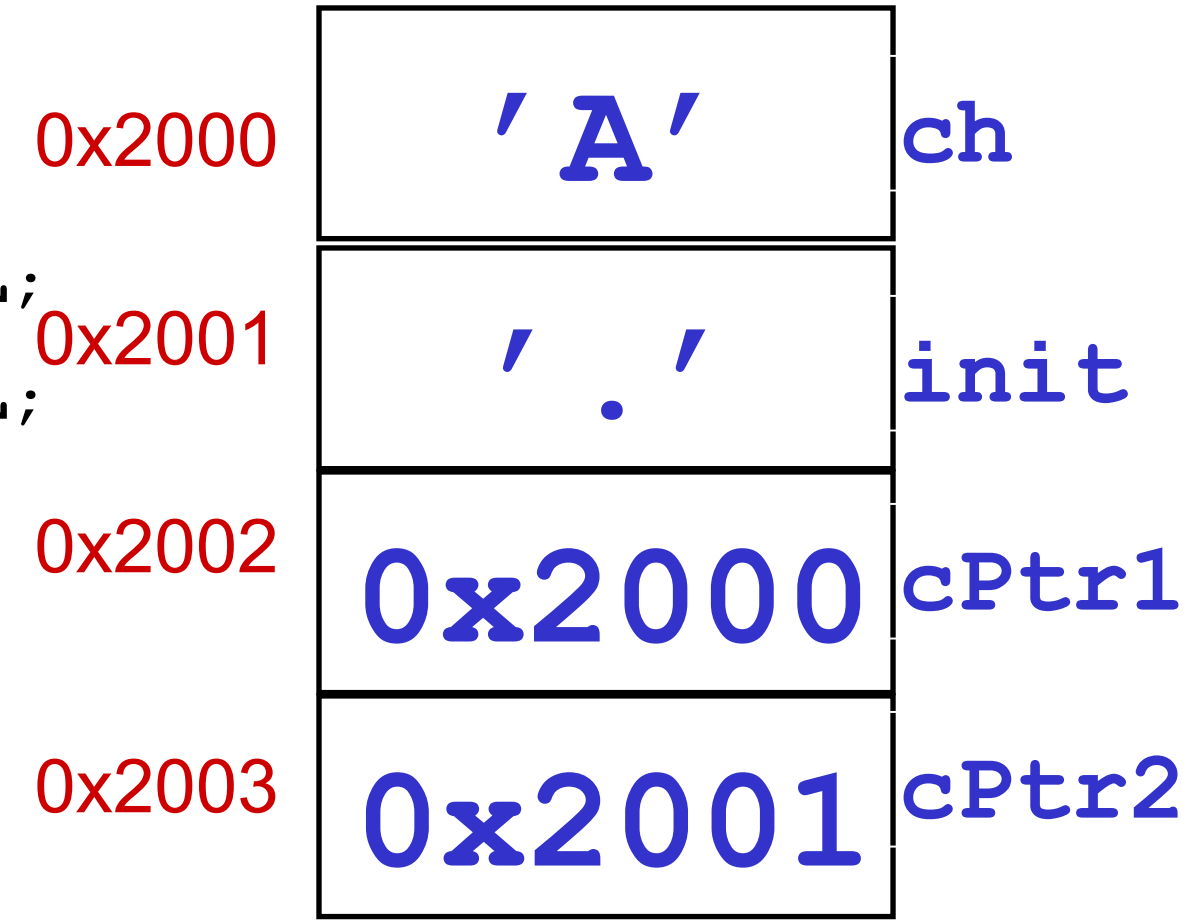
Pointer examples

```
char ch='A';
char init='B';
char* cPtr1=NULL;
char* cPtr2=NULL;
cPtr1=&ch;
cPtr2=&init;
*cPtr2='.';
```



Pointer examples

```
char ch='A';
char init='B';
char* cPtr1=NULL;
char* cPtr2=NULL;
cPtr1=&ch;
cPtr2=&init;
*cPtr2='.';
cPtr2 is ??
```

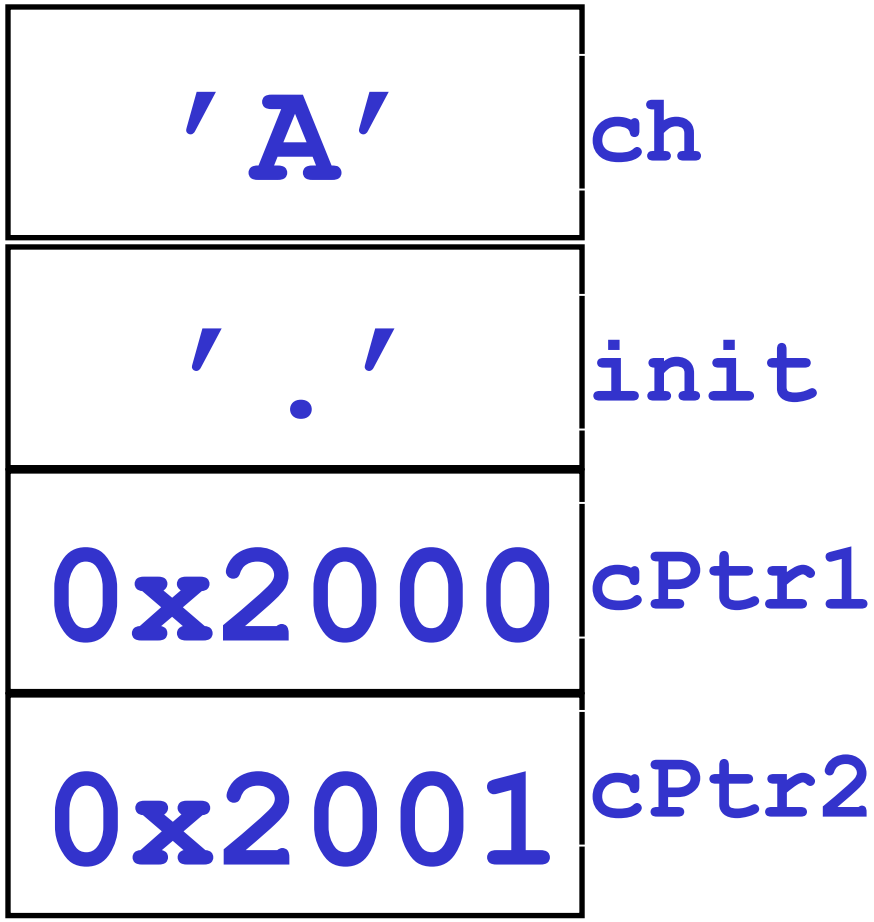


Pointer examples

```
char ch='A';
char init='B';
char* cPtr1=NULL;
char* cPtr2=NULL;
cPtr1=&ch;
cPtr2=&init;
*cPtr2='.';
```

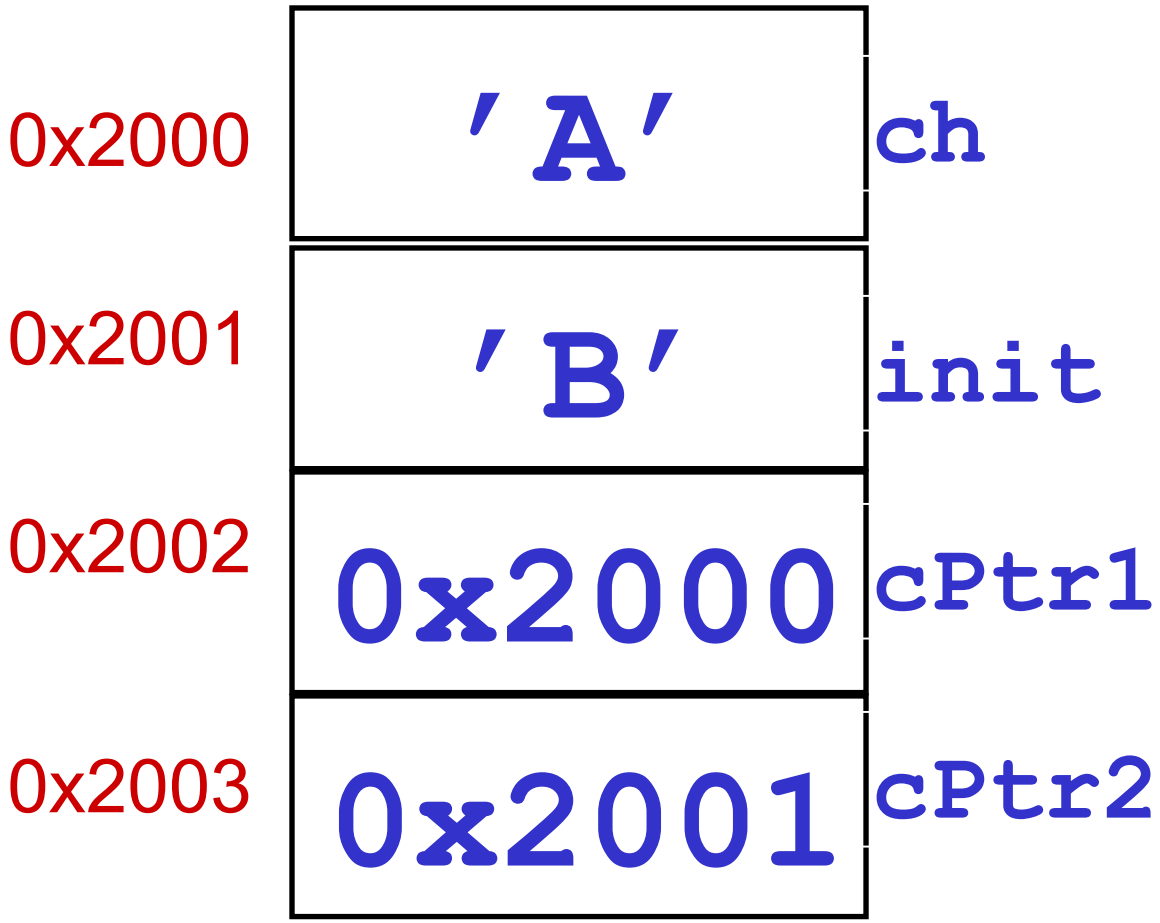
cPtr2 is 0x2001

(same as before - why would it change?)



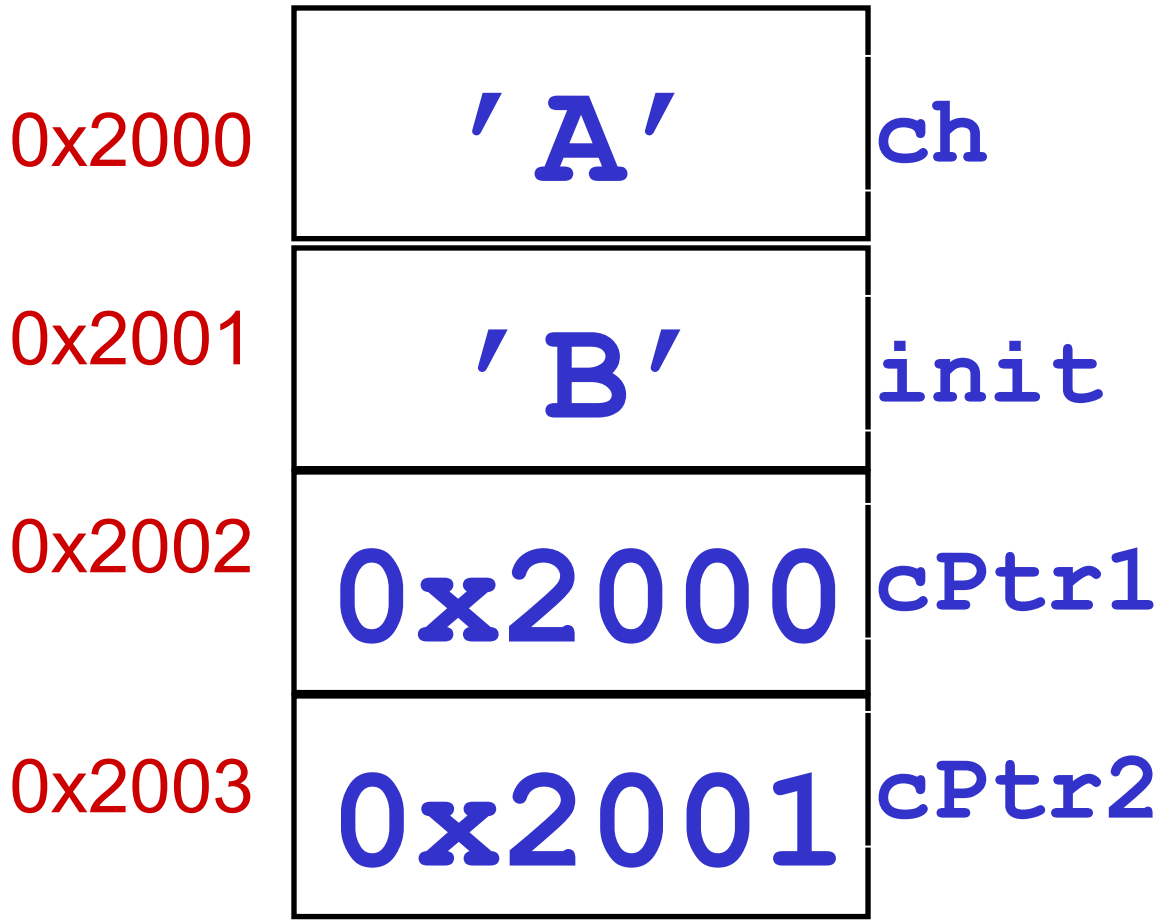
Pointer examples

```
char ch='A';
char init='B';
char* cPtr1=NULL;
char* cPtr2=NULL;
cPtr1=&ch;
cPtr2=&init;
*cPtr1 is ??
*cPtr2 is ??
 cPtr1 is ??
 cPtr2 is ??
```



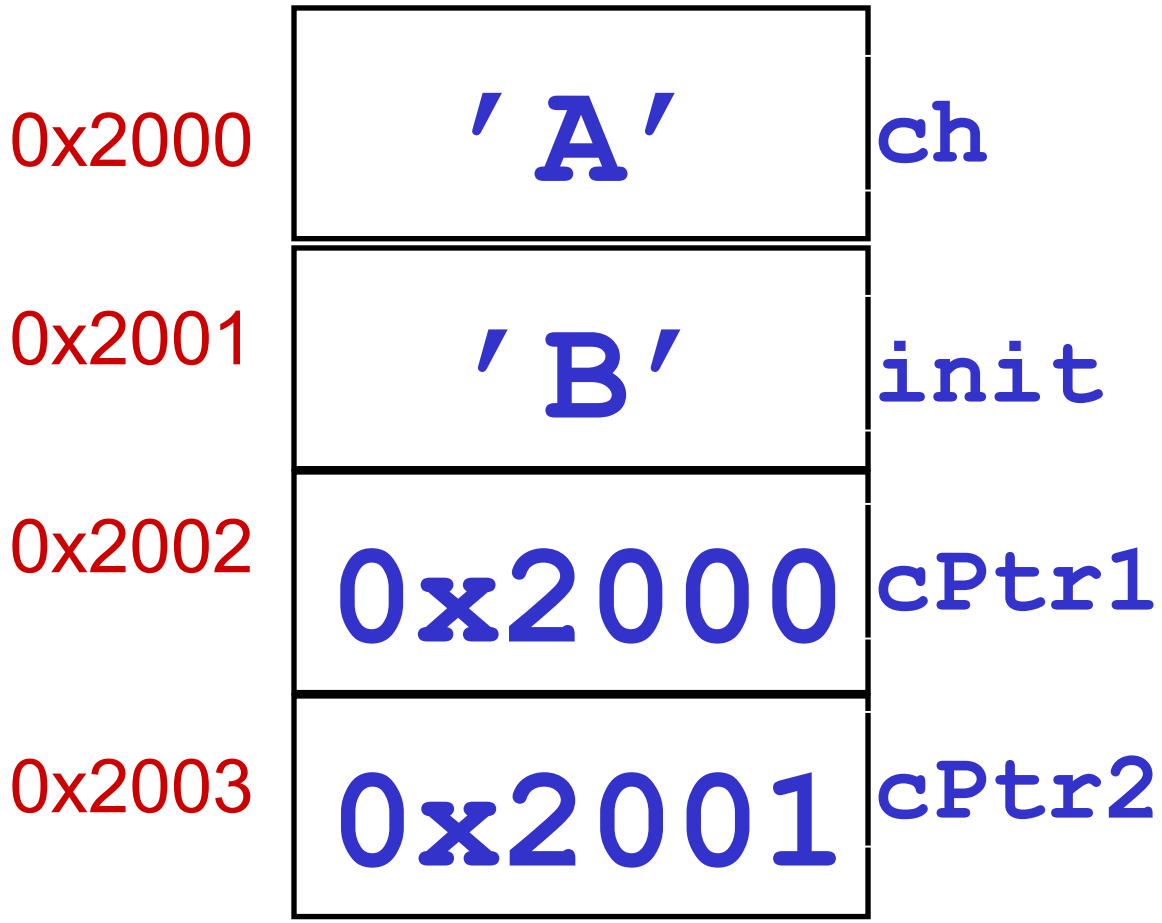
Pointer examples

```
char ch='A';

char init='B';

char* cPtr1=NULL;

char* cPtr2=NULL;

cPtr1=&ch;

cPtr2=&init;

*cPtr1 is 'A'

*cPtr2 is ??

 cPtr1 is ??

 cPtr2 is ??
```



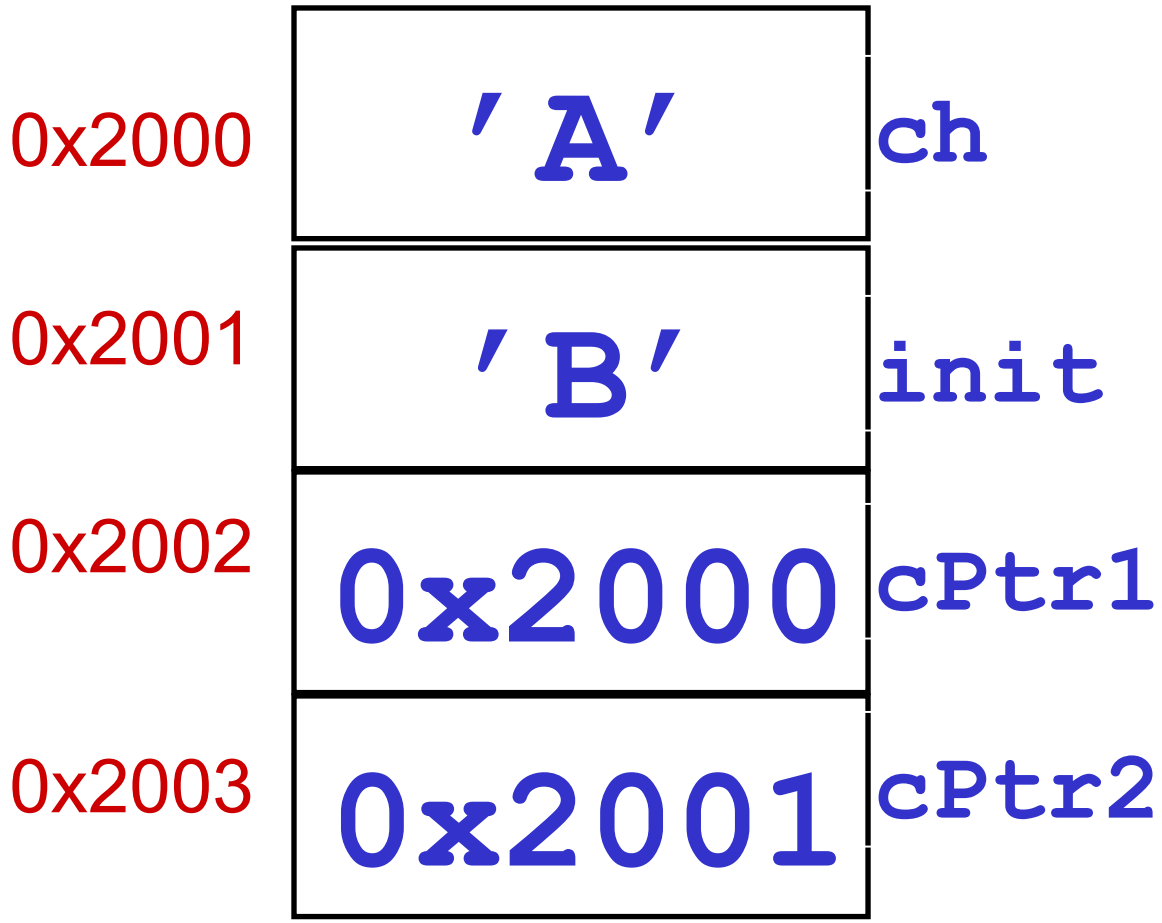
Pointer examples

```
char ch='A';

char init='B';

char* cPtr1=NULL;

char* cPtr2=NULL;

cPtr1=&ch;

cPtr2=&init;

*cPtr1 is 'A'

*cPtr2 is 'B'

 cPtr1 is ??

 cPtr2 is ??
```



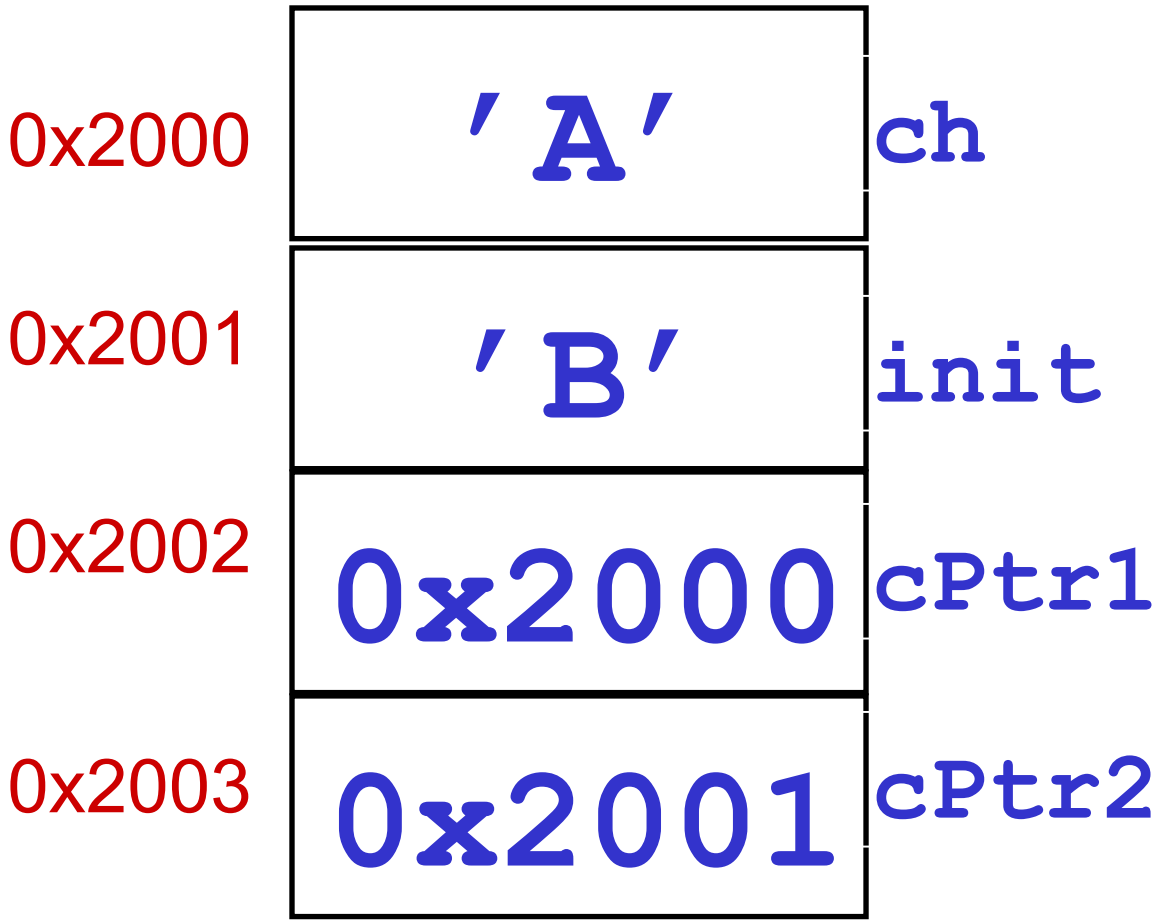
Pointer examples

```
char ch='A';

char init='B';

char* cPtr1=NULL;

char* cPtr2=NULL;

cPtr1=&ch;

cPtr2=&init;

*cPtr1 is 'A'

*cPtr2 is 'B'

 cPtr1 is 0x2000

 cPtr2 is ??
```



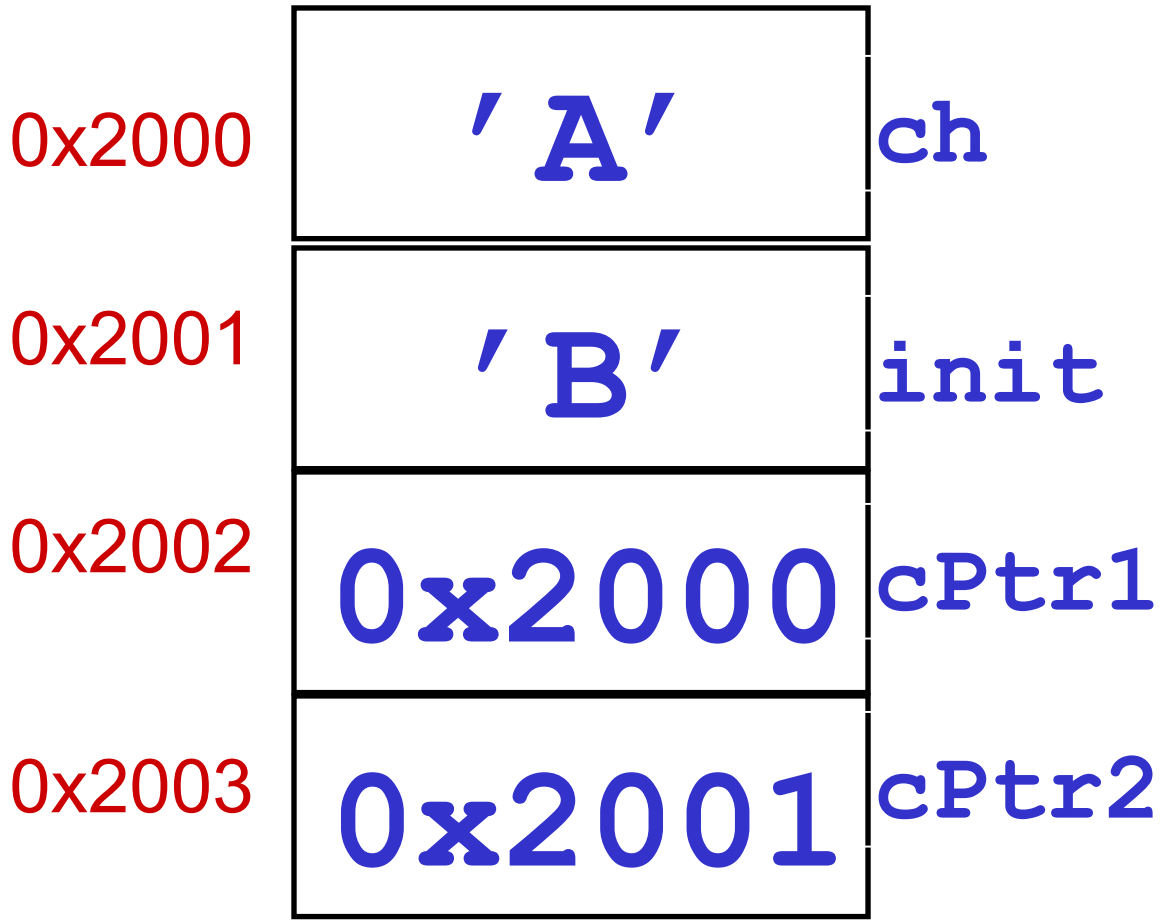
Pointer examples

```
char ch='A';

char init='B';

char* cPtr1=NULL;

char* cPtr2=NULL;

cPtr1=&ch;

cPtr2=&init;

*cPtr1 is 'A'

*cPtr2 is 'B'

 cPtr1 is 0x2000

 cPtr2 is 0x2001
```



Pointer examples

```
char ch='A';

char init='B';

char* cPtr1=NULL;

char* cPtr2=NULL;

cPtr1=&ch;

cPtr2=&init;

...

cPtr1=&init;

*cPtr1 is ??
```



Pointers as Parameters

- Recall that parameters are normally passed as copies
- **`f(x)`** takes a copy of the value of **`x`** and passes it to **`f`**
- This is called *passing by value*
- When you pass the address of a variable, you tell the function where to find the *actual* variable, not just a copy of it

Pointers as Parameters (cont)

- Passing the address means the function can go and look at the variable, and change its value if it wants to.
- This is called *passing by reference*
- If you pass by reference, you can change the variable
- If you pass by value, you can only change the copy - this has no effect on the original variable

Advantages of passing by reference

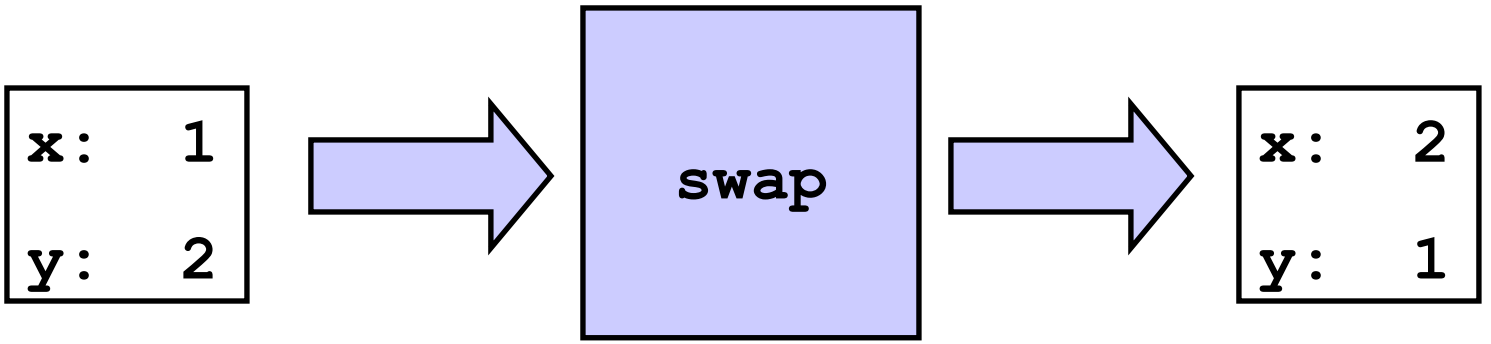
- Efficient
  - because you are not wasting space by making extra copies of variables every time you call a function
- Another way to return information from a function
  - How do you return more than one value from a function? Using parameters passed by reference!

Disadvantages of passing by reference

- Harder to keep track of where (and how) a variable changes
  - Now changes could happen anywhere in a program, not just in the function a variable was *born* in (is local to)
- Functions are less *neat*
  - a function that returns a single value is mathematically neat, one that changes other values is messier to define precisely

Pointers and Function Parameters

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

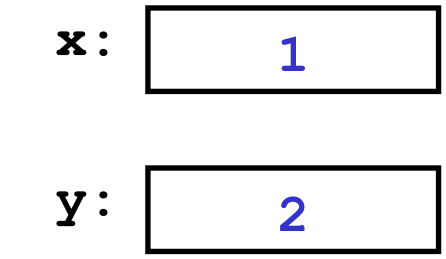


Bad swap

```
#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;
}
```

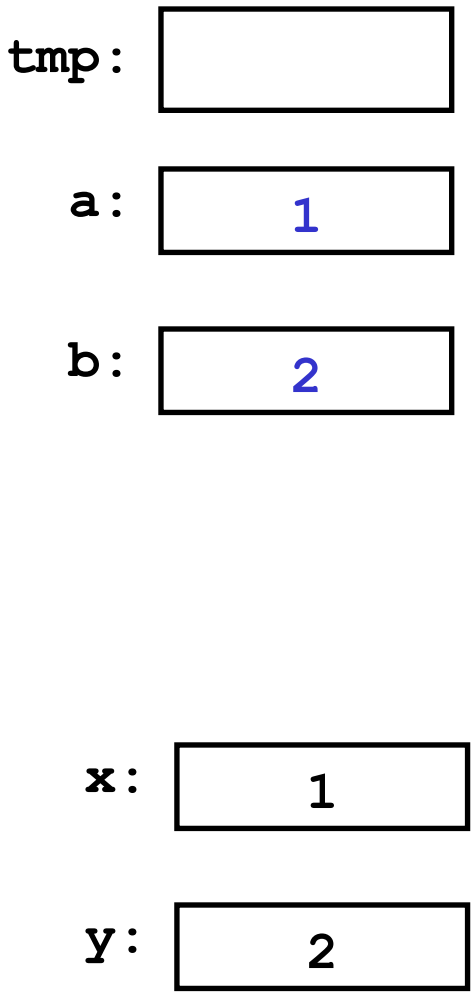
Bad swap

```
#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;
}
```



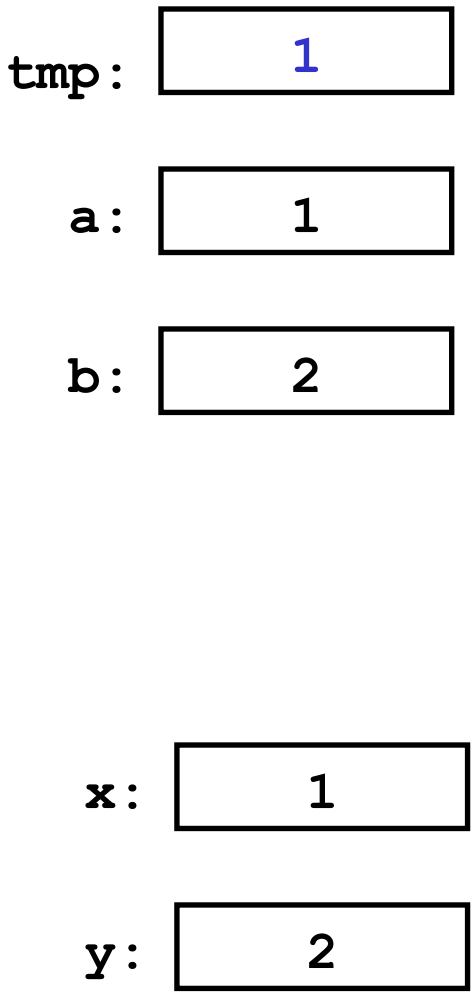
Bad swap

```
#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;
}
```



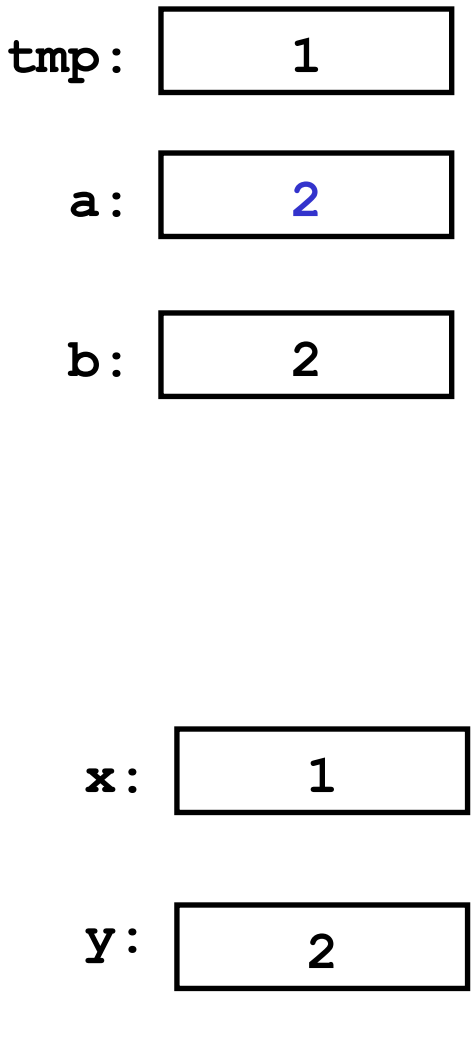
Bad swap

```
#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;
}
```



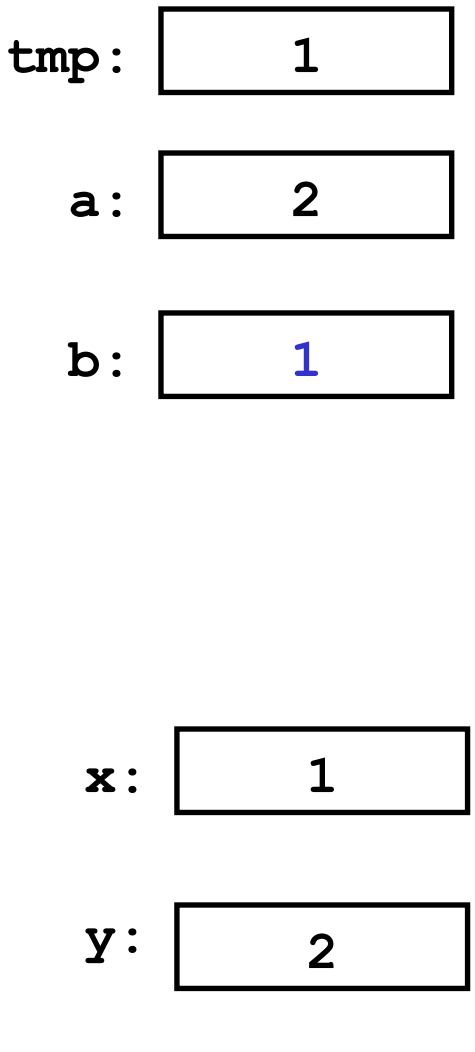
Bad swap

```
#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;
}
```



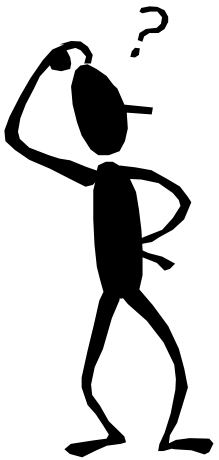
Bad swap

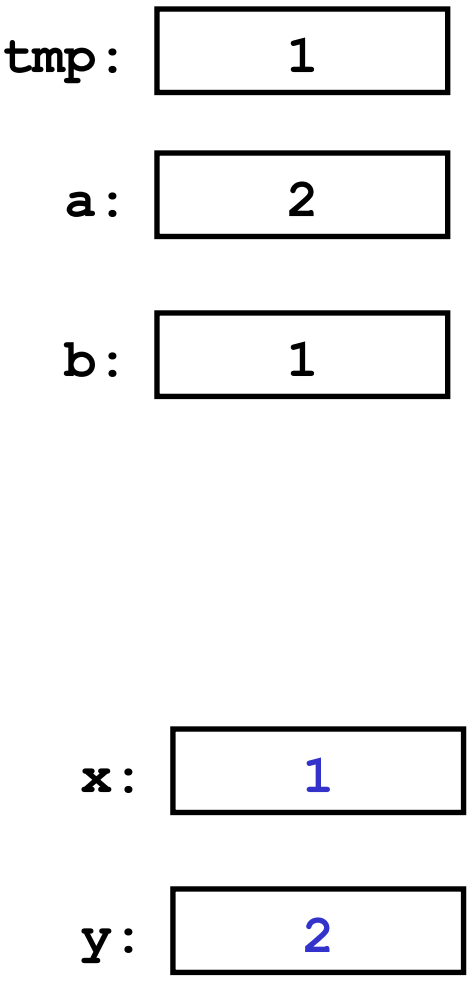
```
#include <stdio.h>

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;
}
```



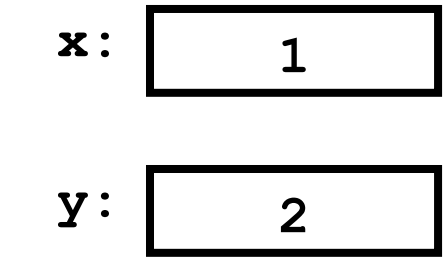


Good swap

```
#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;
}
```

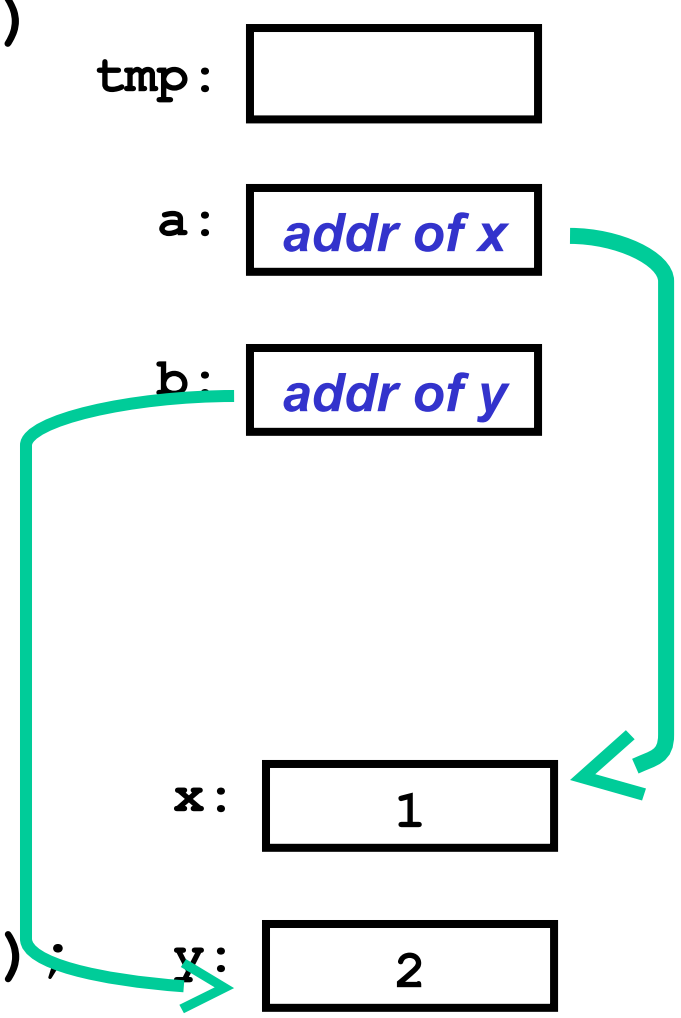
Good swap

```
#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;
}
```



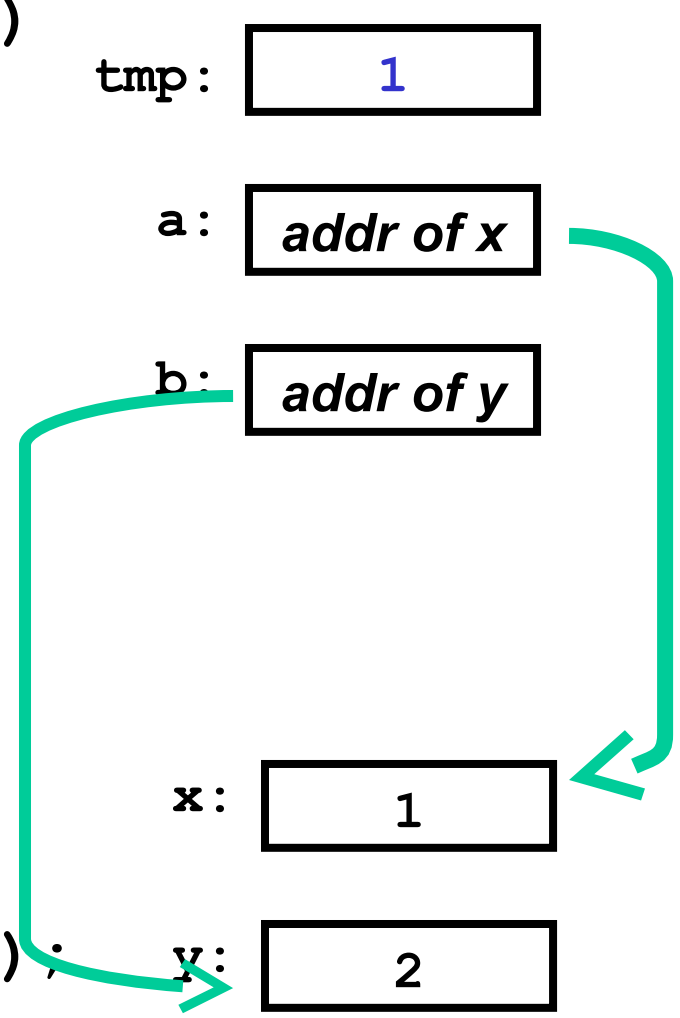
Good swap

```
#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;
}
```



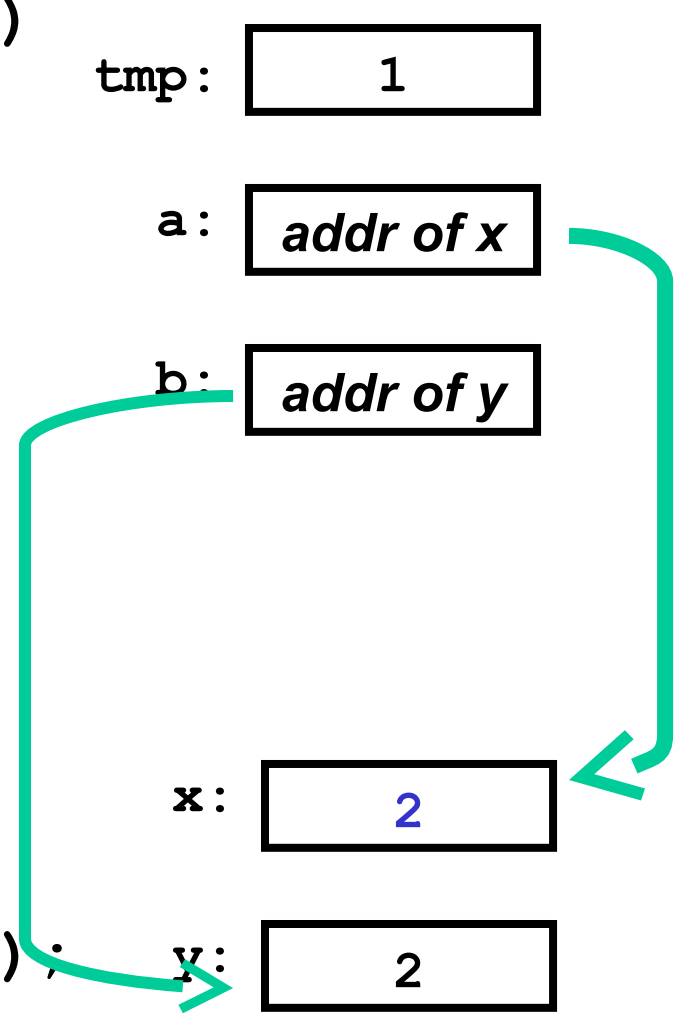
Good swap

```
#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;
}
```



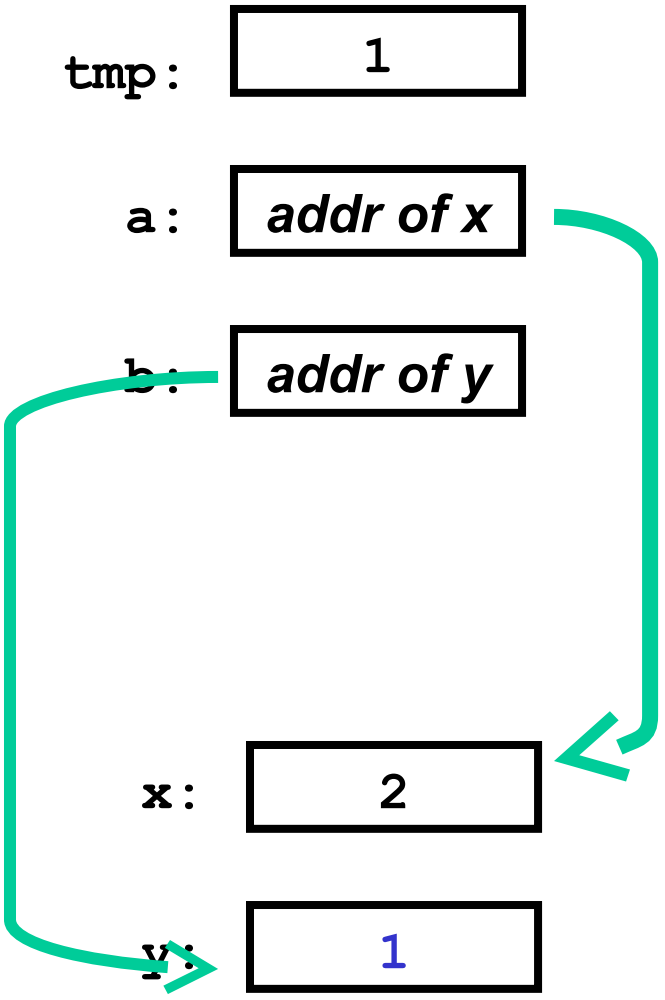
Good swap

```
#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;
}
```



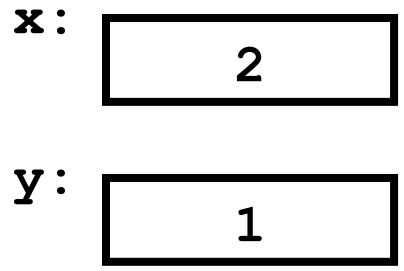
Good swap

```
#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;
}
```



Pointers and Function Arguments

- Change the value of an actual parameter variable

- **`scanf`** demystified

```
char ch;
int  numx;
float numy;
scanf("%c %d %f", &ch, &numx, &numy);
```

More pointer examples

```
int i=0;

int* myPtr=NULL;

int x=3;

myPtr=&x;   /*set myPtr to point to x*/

*myPtr=34; /*set x to be 34, using myPtr*/

myPtr=&i;   /*set myPtr to point to i*/

printf("%d",*myPtr);  /*print i using myPtr*/

printf("%p",myPtr); /*print the address of i*/
```

More pointer examples

```
float x=5.4,y=78.25;

float* xPtr=NULL;

float* yPtr=NULL;

 xPtr=&x;    /*set xPtr to point to x*/

 yPtr=&y;    /*set yPtr to point to y*/

 *xPtr=*yPtr; /*put the value of y in x using pointers*/

 *yPtr=45.0    /*put 45.0 in y using yPtr*/
```

Reading

- King
  - Chapter 11
- Deitel and Deitel
  - Chapter 7 (7.1-7.4)

End of Lecture 5