C - Functions, Pointers, Arrays

Jinyang Li

Lesson Plan

- Control Flow
  - goto
- Function & variable storage
- Pointers

C’s Control flow

- Same as Java
- Conditional:
  - if ... else if... else
  - switch
- Loops: while, do while, for
  - continue
  - break

goto statements allow jump anywhere

```
while (cond) {
    ...
}
B:
...
```

Any control flow primitive can be expressed as a bunch of goto's.

```
A:
 if (cond = false) goto B;
 ...
 goto A
B:
 ...
```

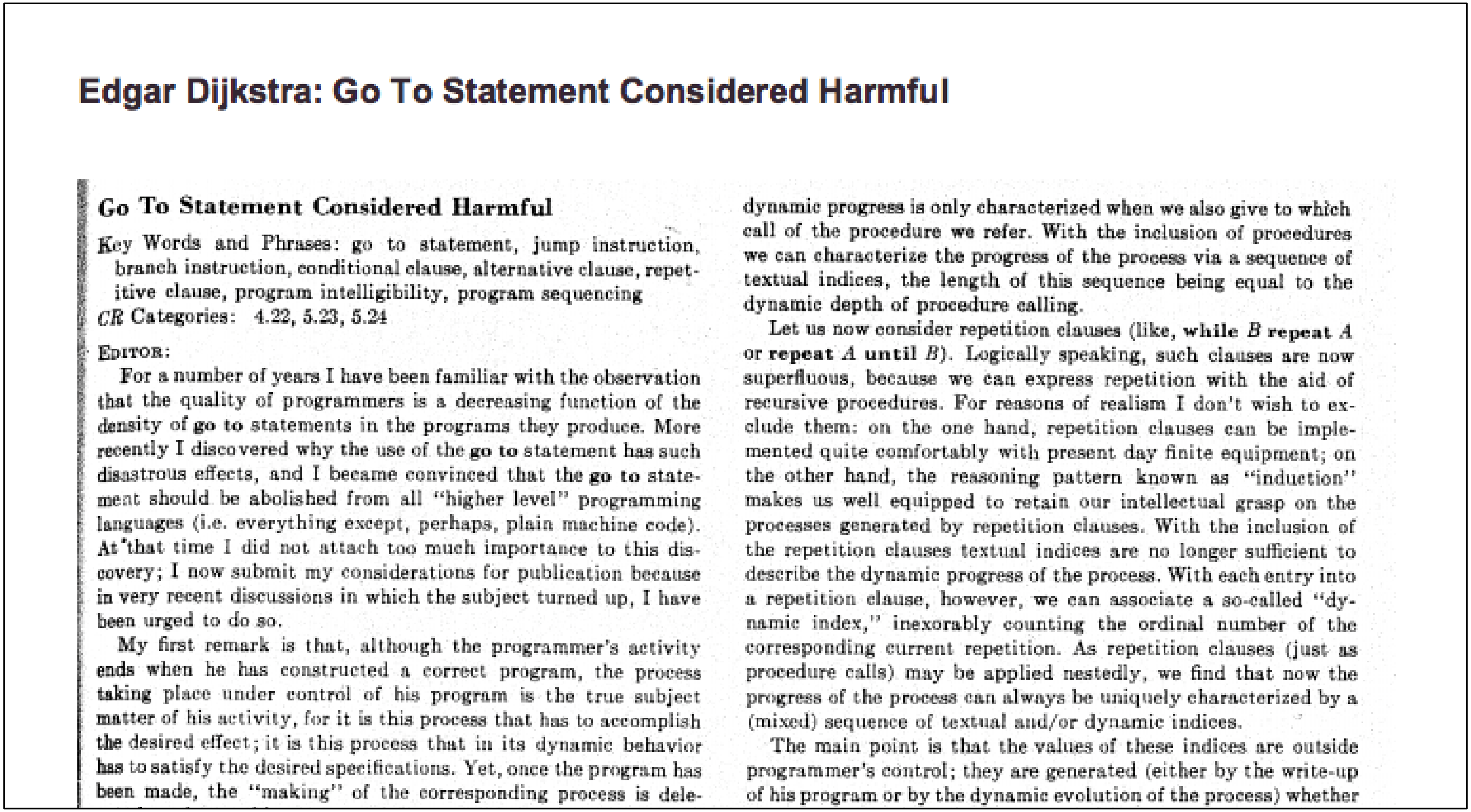
There's no goto in Java or core Python

```
for(...) {
    for(...) {
        for(...) {
            goto error
        }
    }
}

error:
    code handling error
```

The only acceptable scenario for using goto

There's no try/catch or try/except in C

Avoid goto's whenever possible



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

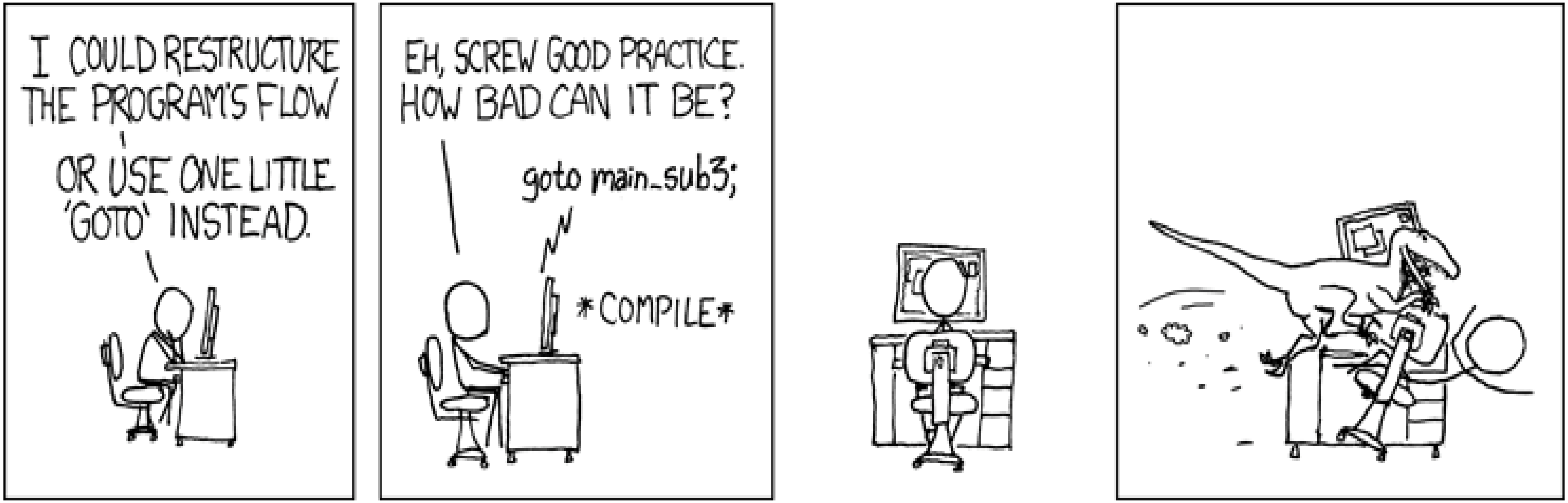
My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is dele-

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether

Avoid goto's whenever possible


I COULD RESTRUCTURE THE PROGRAM'S FLOW
OR USE ONE LITTLE 'GOTO' INSTEAD.
EH, SCREW GOOD PRACTICE. HOW BAD CAN IT BE?
goto main_sub3;
COMPILE

C is a procedural-language

- C program consists of functions
  - Also called procedures or subroutines
- Why breaking code into functions?
  - Readability and Reusability
- Keep functions short
  - General rules of thumb:
    - Small enough to keep the code in your head
    - Fits on screen without scrolling

Variable and its storage: local variable

Local variables are defined inside a function

You can view function arguments as local variables defined in function body

```
int add(int a, int b)
{
    int r = a + b;
    return r;
}
```

r’s scope is in function *add*

Local variable scope:

- Within the function/block the local variable is declared
- Local variables with the same name in different scopes are unrelated
- **Shadowing**: Nested variable “hides” the outer variable in areas where they are both in scope

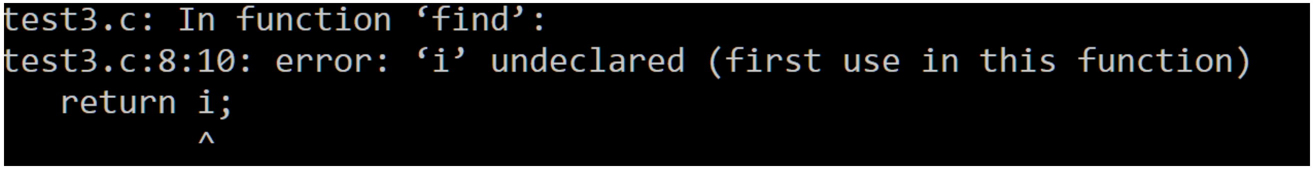
```
int add(int a, int b)
{
    int r = a;
    {
       int r;
       r += b;
    }
    return r;
}
```

What does add compute?

Variable and its storage: local variable scope

```c
int find(int start, int end)
{
    for (int i = 0; i < a; i++) {
        if ((i % 3) == 0)
           break;
    }
    return i;
}
```

```
$gcc -std=c99 test.c
```



```
test3.c: In function ‘find’:
test3.c:8:10: error: ‘i’ undeclared (first use in this function)
   return i;
          ^
```

Variable and its storage: local variable scope

```
int find(int start, int end)
{
    int r;
    for (int i = 0; i < a; i++) {
        if ((i % 3) == 0) {
            r = i;
            break;
        }
    }
    return r;
}
```

Correct?

Variable and its storage: local variable storage

Local variable storage:

- allocated upon function invocation
- deallocated upon function return

```
void add(int a, int b, int result)
{
    int result = a + b;
    return;
}

int main()
{
    int result;
    add(1, 2, result);
    printf(“r=%d\n”, result);
}
```

Program output?

Variable and its storage: global variable

Global variables are defined outside any function

```
#include <stdio.h>

int r = 0;
int sum(int a, int b)
{
    r = a + b;
}
int main()
{
    sum(1,2);
    int r = 0;
    printf(“r=%d\n”, r);
}
```

- Global variable scope:
  - Can be accessed from within any function
  - May be shadowed
- Global variable storage
  - Allocated upon program start, deallocated when entire program exits

Function invocation

C (and Java) passes arguments by value

```
void swap(int a, int b)
{
    int tmp = a;
     a = b;
     b = tmp;
}

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

    printf(“x: %d, y: %d”, x, y);
}
```

Result x: ?, y: ?

Function invocation

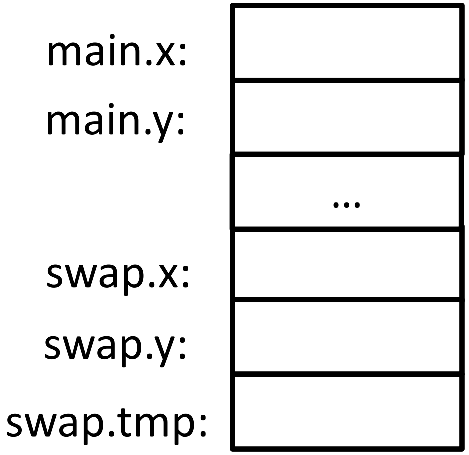
C (and Java) passes arguments by value

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

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

    printf(“x: %d, y: %d”, x, y);
}
```

Result x: ? y: ?

Function invocation

```
void swap(int a, int b)
{
    int tmp = x;
     x = y;
     y = tmp;
}

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

    printf("x: %d, y: %d", x, y);
}
```



main.x:

main.y:

...

swap.x:

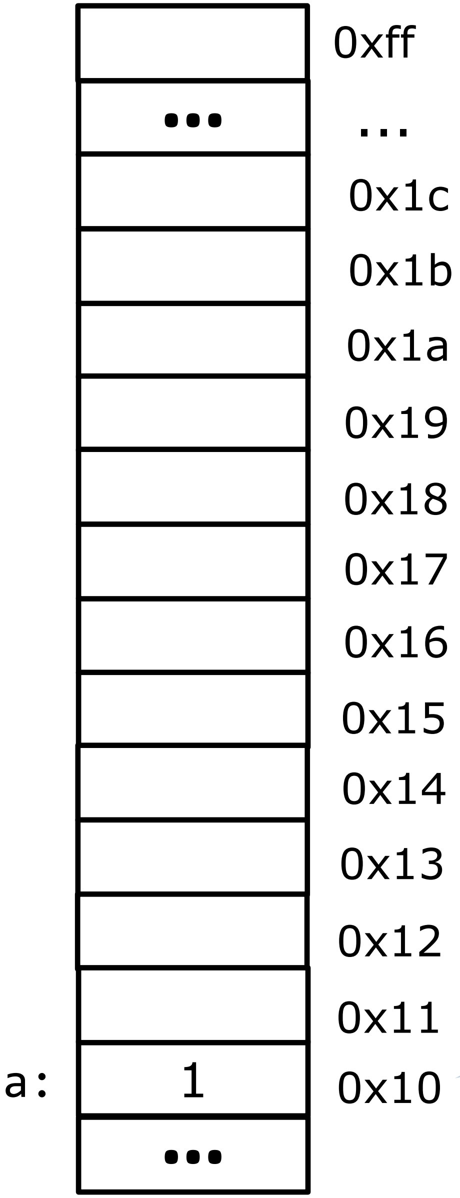
swap.y:

swap.tmp:

Pointers

Pointer is a memory address

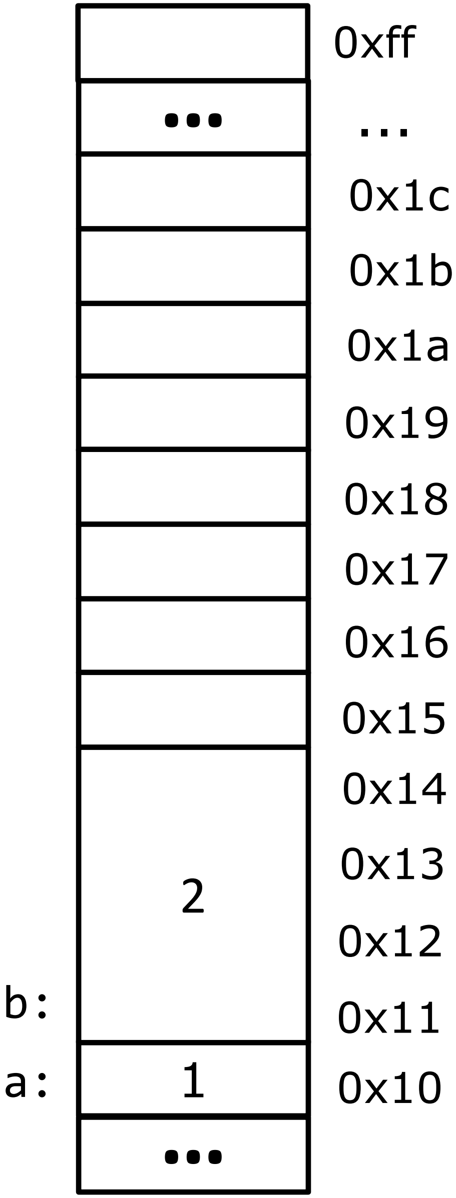
Pointer



```
char a = 1;
```

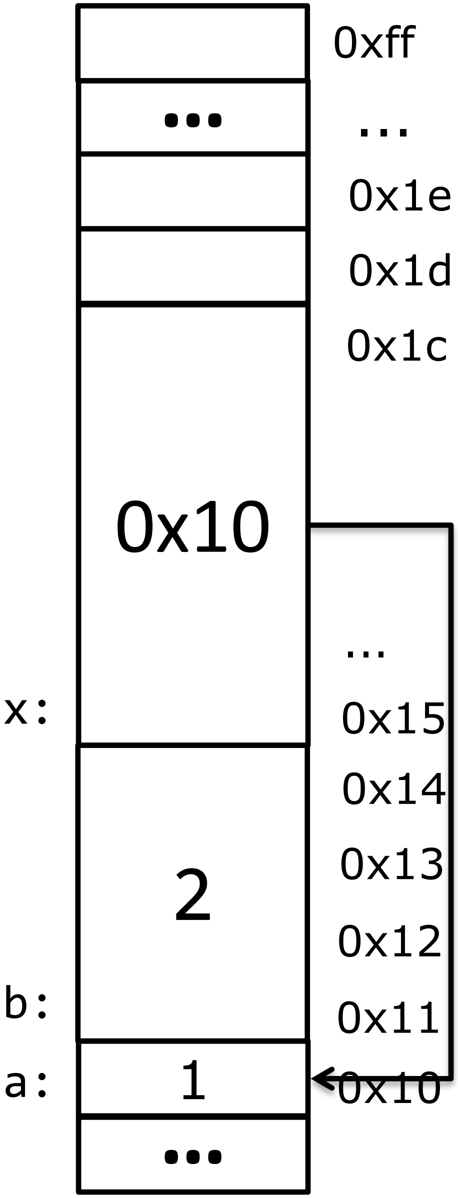
Addresses should be 8-byte long, but for the same of simplicity, picture only shows one byte

Pointer



```
char a = 1;
int b = 2;
```

Pointer



```
char a = 1;
int b = 2;
char *x;
x = &a;
```

Same as: char* x;

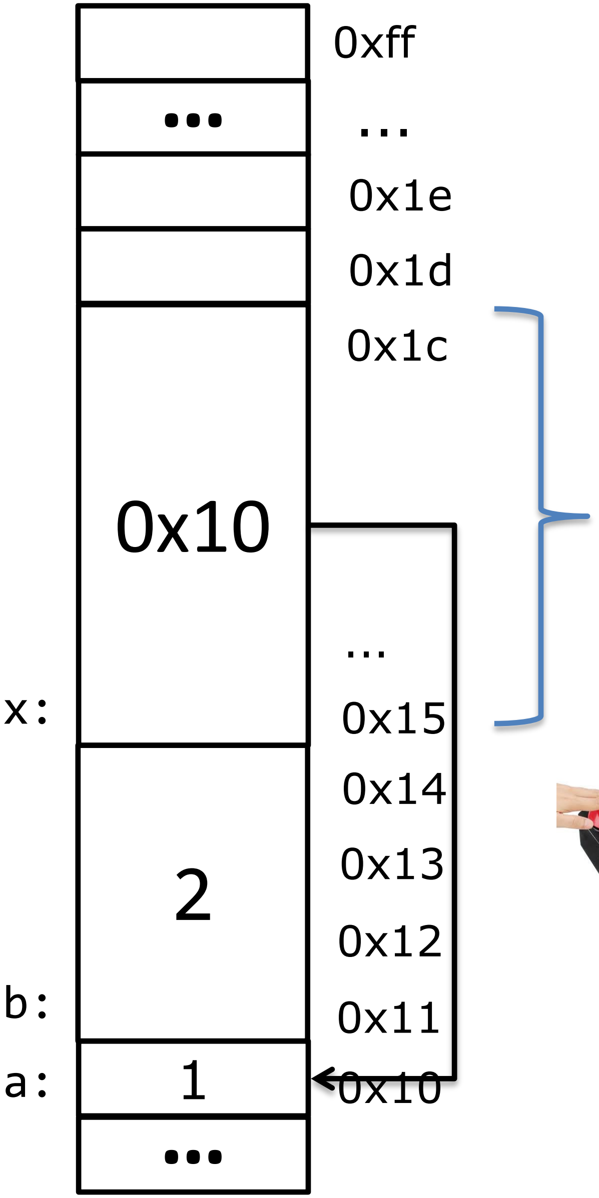
& gives address of variable

Can be combined as:
char *x = &a;

what happens if I write
char x = &a;
or
int *x = &a;

type mismatch!

Pointer



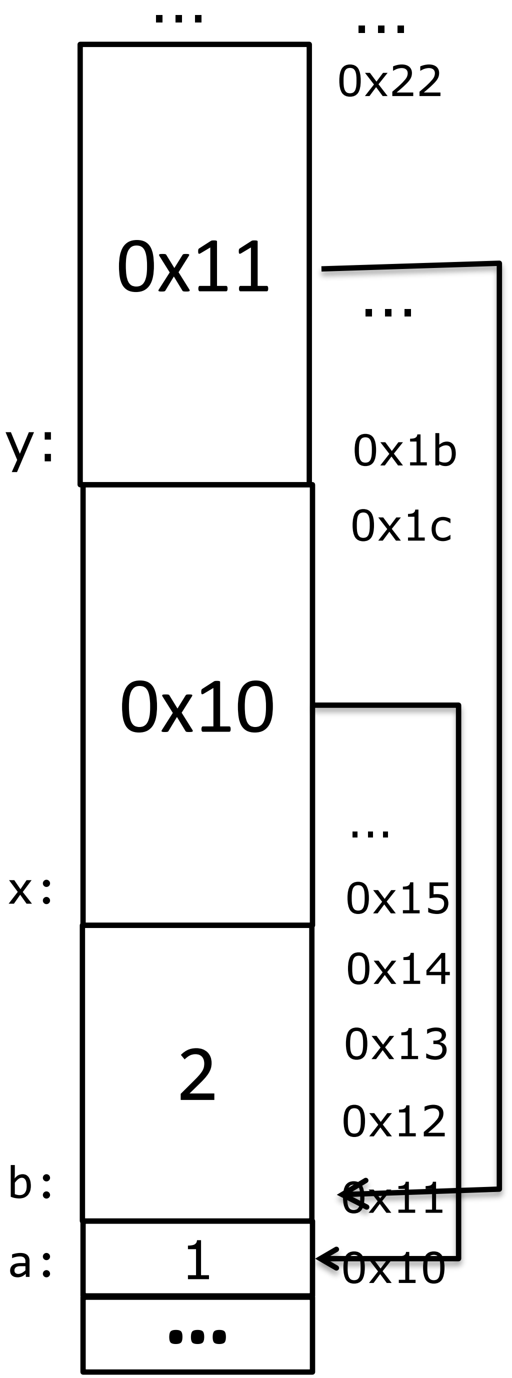
```
char a = 1;
int b = 2;
char *x = &a;
printf("x=%p\n", x);
```

You can print the value of a pointer variable with %p (leading zeros are not printed)
0x10

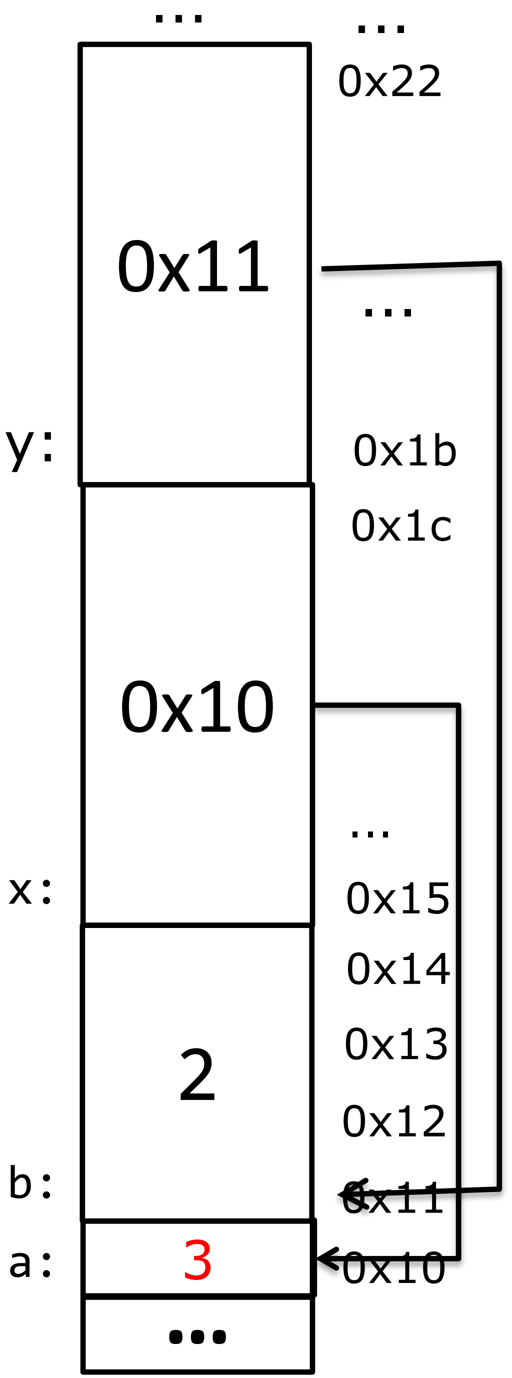
Size of pointer on a 64-bit machine?
8 bytes

Pointer

```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;
```



Pointer

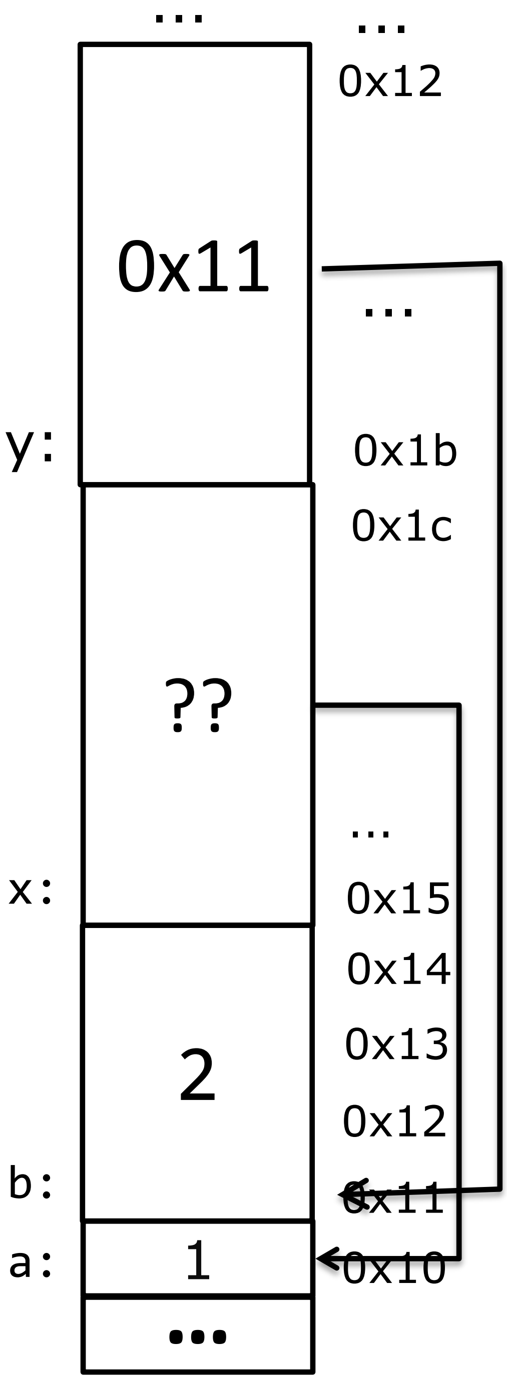


```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

 *x = 3;
```

* operator dereferences a pointer, not to be confused with the * in (char *) which is part of typename

Value of variable a after this statement?

Pointer



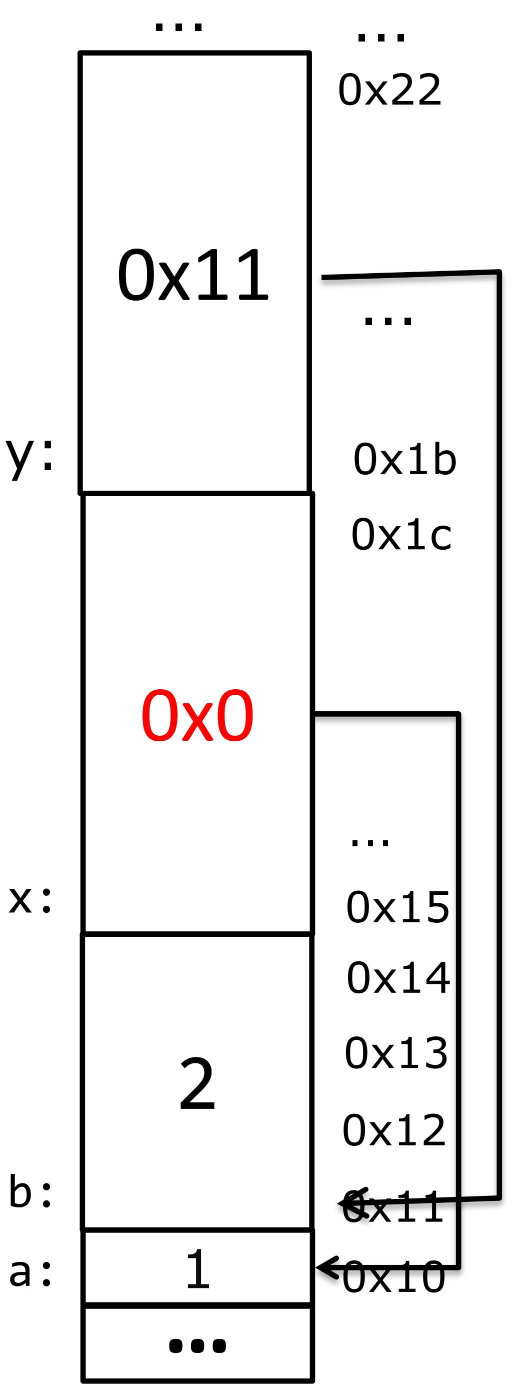
```
char a = 1;
int b = 2;
char *x ~~= &a~~;
int *y = &b;
```

what if x is uninitialized?

```
*x = 3;
```

Dereferencing an arbitrary address value may result in “Segmentation fault” or a random memory write

Pointer

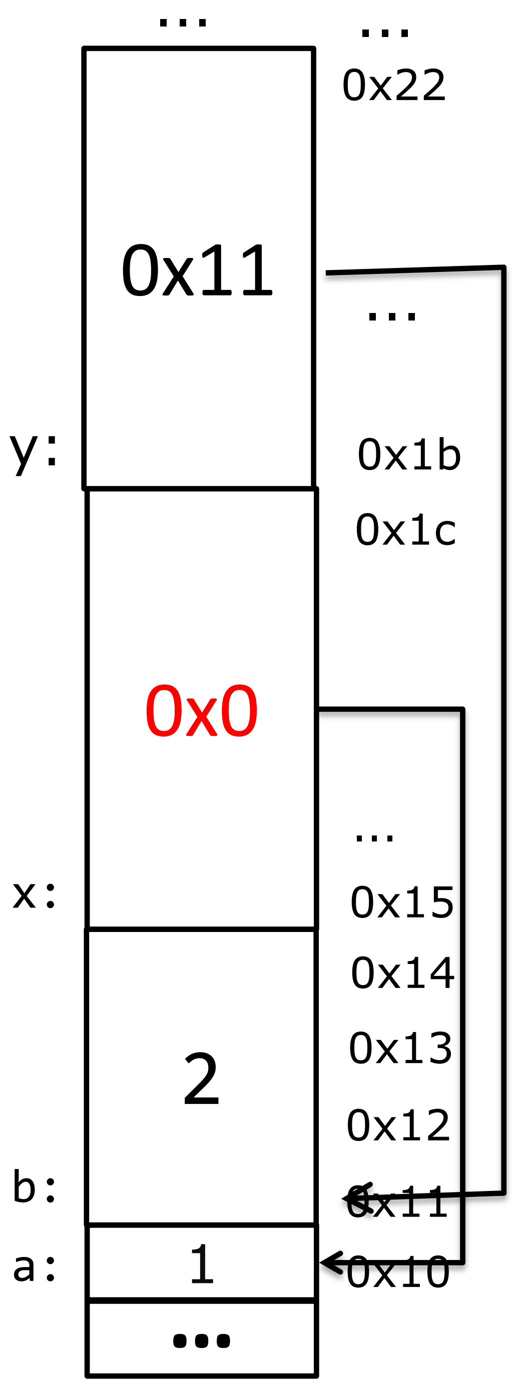


```
char a = 1;
int b = 2;
char *x = NULL;
int *y = &b;

 *x = 3;
```

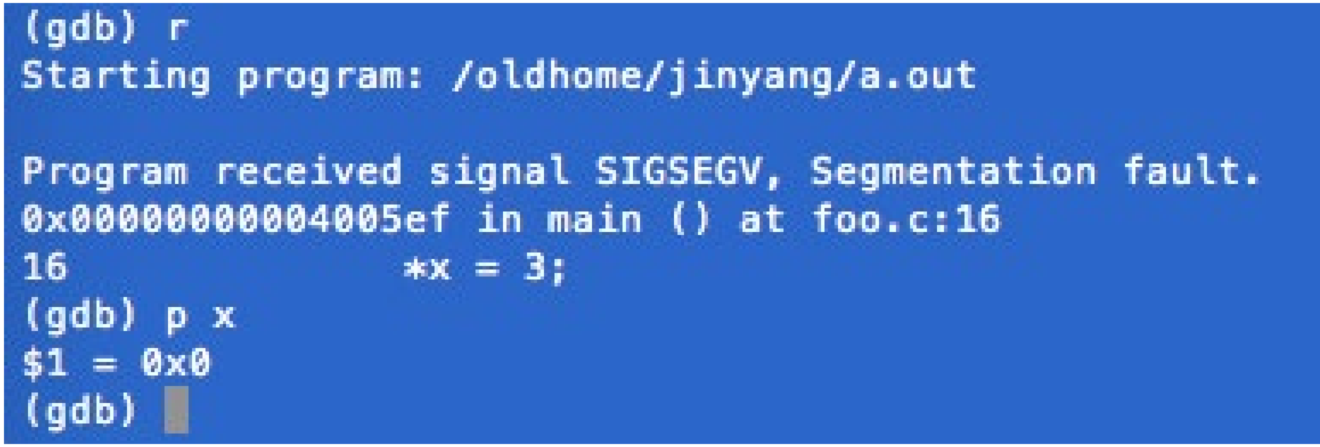
Always initialize pointers!

Dereferencing NULL pointer definitely results in “Segmentation fault”

Pointer

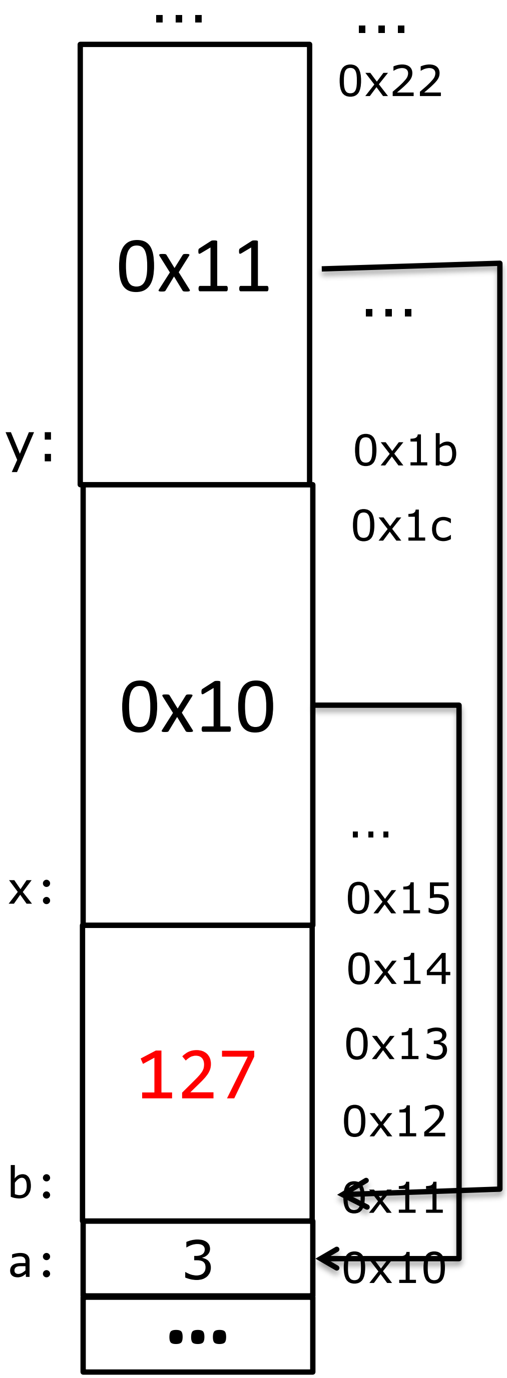


```
char a = 1;
int b = 2;
char *x = NULL;
int *y = &b;


 *x = 3;
```

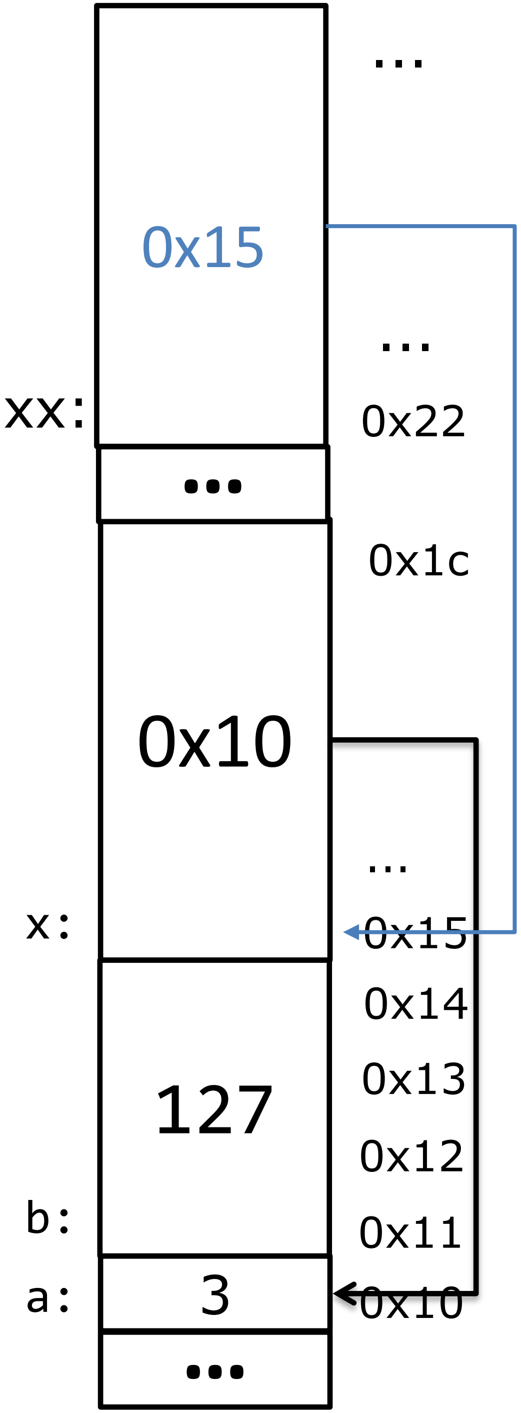


```
(gdb) r
Starting program: /oldhome/jinyang/a.out

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005ef in main () at foo.c:16
16              *x = 3;
(gdb) p x
$1 = 0x0
(gdb) 
```

Pointer



```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

  *x = 3;
  *y = 127;
```



Double Pointer

```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

*x = 3;
*y = 127;

char **xx = &x;
```

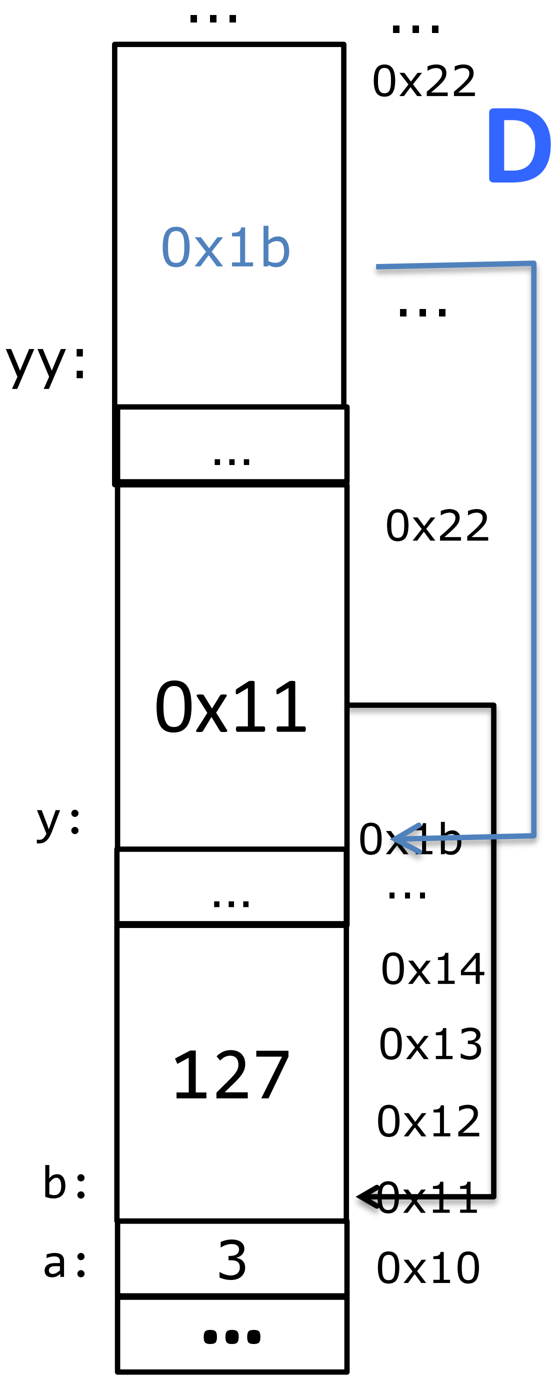
Same as:
char **xx;
xx = &x;

char **xx is the same as char** xx;

what if I write
char* xx;
xx = &x;

printf("xx=%p *xx=%p **xx=%d\n", xx, *xx, **xx);

Double Pointer



```
char a = 1;
int b = 2;
char *x = &a;
int *y = &b;

 *x = 3;
 *y = 127;

 char **xx = &x;
 int **yy = &y;
```

printf(“yy=%p *yy=%p **yy=%d\n”, yy, *yy, **yy);

Common confusions on *

* has two meanings!!

1. part of a pointer type name, e.g. char *, char **, int *
2. the deference operator.

```
char a = 1;
char *p = &a;
*p = 2;

char *b, *c;
char **d,**e;

char *f=p, *g=p;
char **m=&p, **n=&p;
```

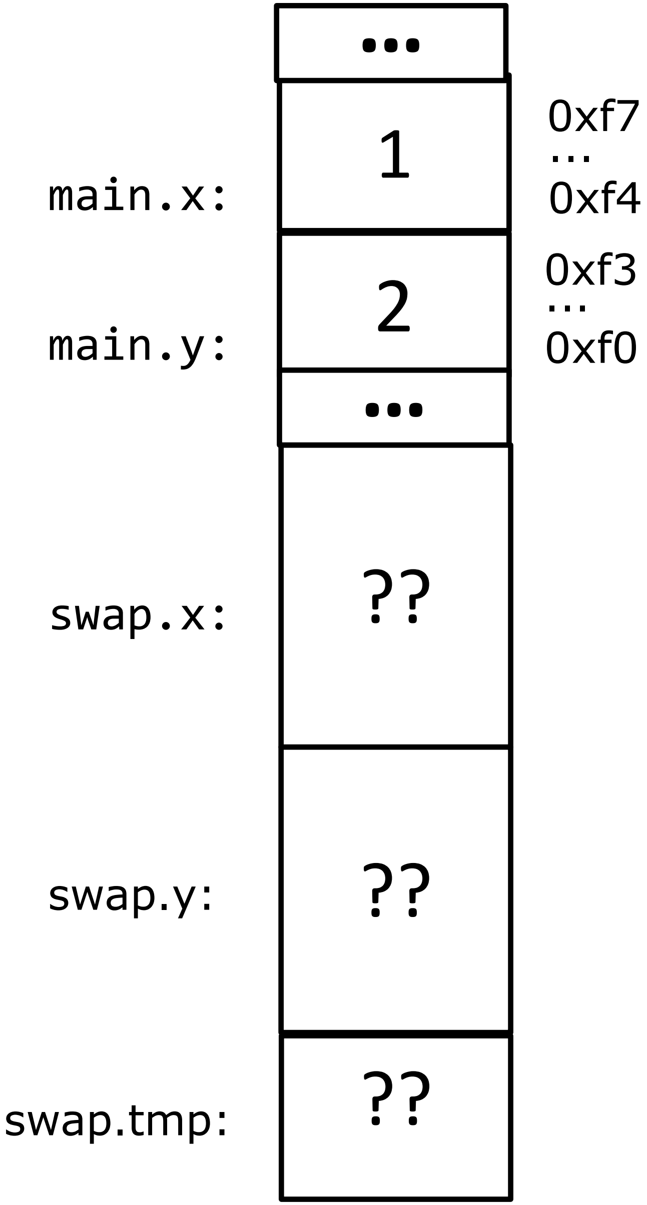
C’s syntax for declaring multiple pointer variables on one line

char*   b, c; does not work

C’s syntax for declaring and initializing multiple pointer variables on one line

Pass pointers to function

```
void swap(int* x, int* y)
{
      int tmp = *x;
      *x = *y;
      *y = tmp;
}
int main()
{
     int x = 1;
     int y = 2;
     swap(&x, &y);

     printf(“x:%d, y:%d”,x,y);
}
```



Size and value of x, y, tmp
in swap upon function entrance?

Summary

- Control Flow
  - goto
- Local vs. global variable
  - Local variables allocated/deallocated upon function entrance/return
  - Global variable: always there
- Pointers
  - Pointer values are memory addresses
  - p =&x; (makes p points to variable x)
  - *p … (refers to the variable pointed to by p)