# CS 33

## Introduction to C
### Part 5

# Scope

```
int a;    // global variable

int main() {
   int a;    // local variable
   a = 0;
   proc();
   printf("a = %d\n", a); // what’s printed?
   return 0;
}

int proc() {
   a = 1;
   return a;
}
```

```
$ ./a.out
0
```

# Scope (continued)

```
int a;   // global variable

int main() {
   a = 2;
   proc(1);
   return 0;
}

int proc(int a) {
   printf("a = %d\n", a); // what’s printed?
   return a;
}
```

```
$ ./a.out
1
```

# Scope (still continued)

```
int a;    // global variable

int main() {
   a = 2;
   proc(1);
   return 0;
}

int proc(int a) {
   int a;
   printf("a = %d\n", a); // what’s printed?
   return a;
}
```

```
$ gcc prog.c
prog.c:12:8: error: redefinition of 'a'
   int a;
       ^
```

# Scope (more ...)

```
int a;   // global variable

int proc() {
   {
      // the brackets define a new scope
      int a;
      a = 6;
   }
   printf("a = %d\n", a); // what’s printed?
   return 0;
}
```

```
$ ./a.out
0
```

# Quiz 1

```c
int a;

int proc(int b) {
   {int b=6;}
   a = b;
   return a+2;
}

int main() {
   {int a = proc(4);}
   printf("a = %d\n", a);
   return 0;
}
```

- What’s printed?
  - a) 0
  - b) 4
  - c) 6
  - d) 8
  - e) nothing; there’s a syntax error

# Scope and For Loops (1)

```
int A[100];
for (int i=0; i<100; i++) {
  // i is defined in this scope
  A[i] = i;
}
```

# Scope and For Loops (2)

```
int A[100];
initializeA(A);
for (int i=0; i<100; i++) {
  // i is defined in this scope
  if (A[i] < 0)
    break;
}
if (i != 100)
  printf("A[%d] is negative\n", i);
```

**syntax error: reference to *i* is out of scope.**

# Lifetime

```
int count;

int main() {
    func();
    ...
    func(); // what's printed by func?
    return 0;
}

int func() {
    int a;
    if (count == 0) a = 1;
    count = count + 1;
    printf("%d\n", a);
    return 0;
}
```

```
% ./a.out
1
-38762173
```

# Lifetime (continued)

```
int main() {
   func(1); // what’s printed by func?
   return 0;
}
int a;
int func(int x) {
   if (x == 1) {
      a = 1;
      func(2);
      printf("%d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
2
```

# Lifetime (still continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}

int func(int x) {
   int a;
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
1
```

# Lifetime (more ...)

```
int main() {
    int *a;
    a = func();
    printf("%d\n", *a); // what’s printed?
    return 0;
}

int *func() {
    int x;
    x = 1;
    return &x;
}
```

```
% ./a.out
23095689
```

# Lifetime (and still more ...)

```
int main() {
    int *a;
    a = func(1);
    printf("%d\n", *a); // what’s printed?
    return 0;
}

int *func(int x) {
    return &x;
}
```

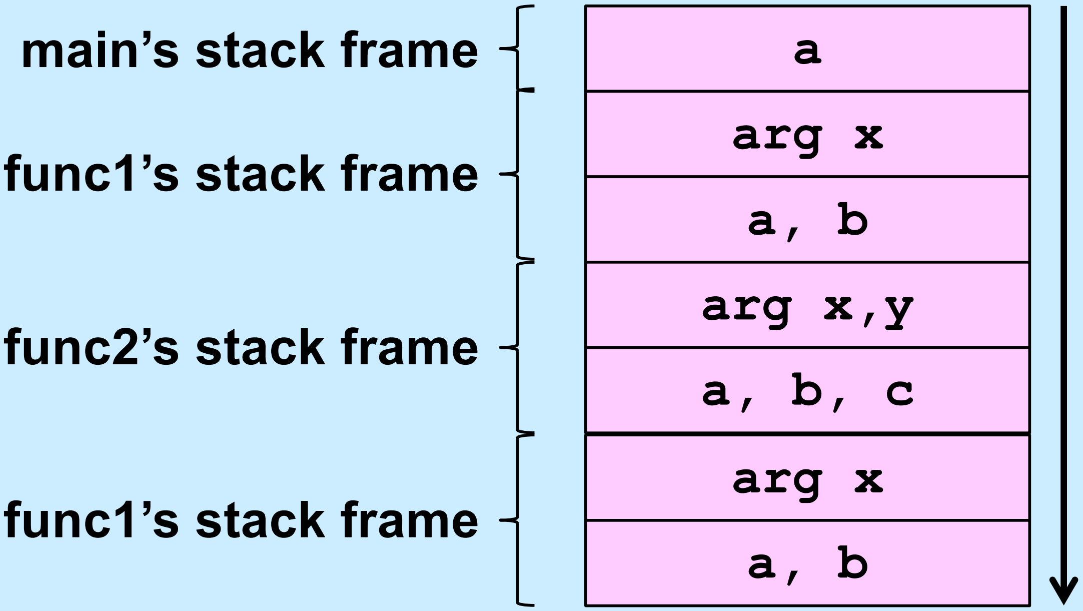
```
% ./a.out
98378932
```

# Rules

- Global variables exist for the duration of program's lifetime
- Local variables and arguments exist for the duration of the execution of the function
  - from call to return
  - each execution of a function results in a new instance of its arguments and local variables

# Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



main's stack frame

func1's stack frame

func2's stack frame

func1's stack frame

a

arg x

a, b

arg x,y

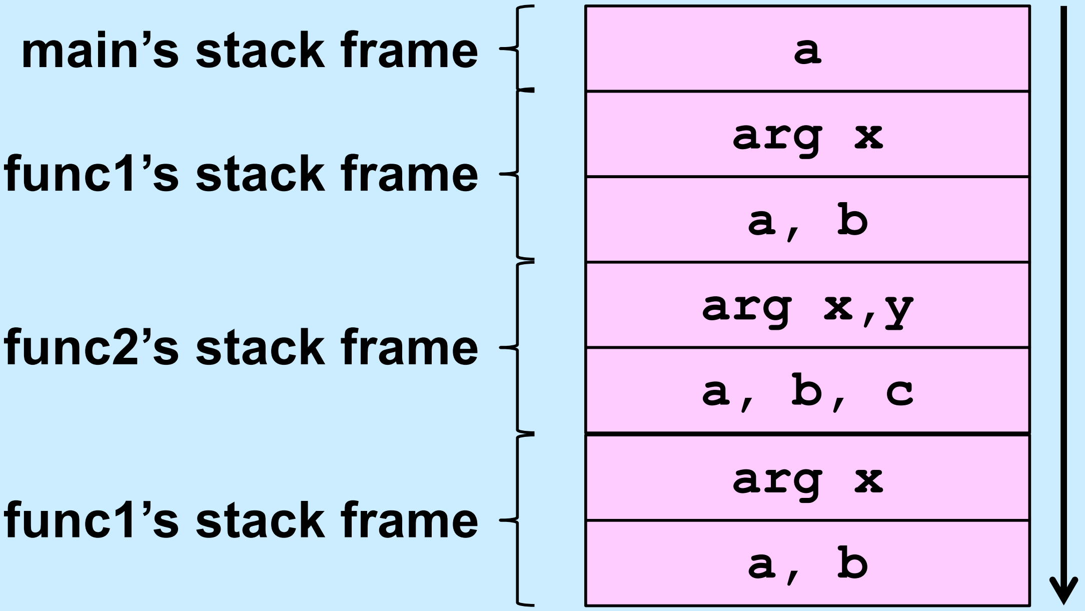
a, b, c

arg x

a, b

# Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



main's stack frame

a

func1's stack frame

arg x

a, b

func2's stack frame

arg x,y

a, b, c

func1's stack frame

arg x

a, b

# Quiz 2

```
void func(int a) {
   int b=2;
   if (a == 1) {
      func(2);
      printf("%d\n", b);
   } else {
      b = a*(b++)*b;
   }
}
int main() {
   func(1);
   return 0;
}
```

- **What's printed?**
  a) 0
  b) 1
  c) 2
  d) 4

# Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```

- **Scope**
  - **like local variables**
- **Lifetime**
  - **like global variables**
- **Initialized just once**
  - **when program begins**
  - **implicit initialization to 0**

# Quiz 3

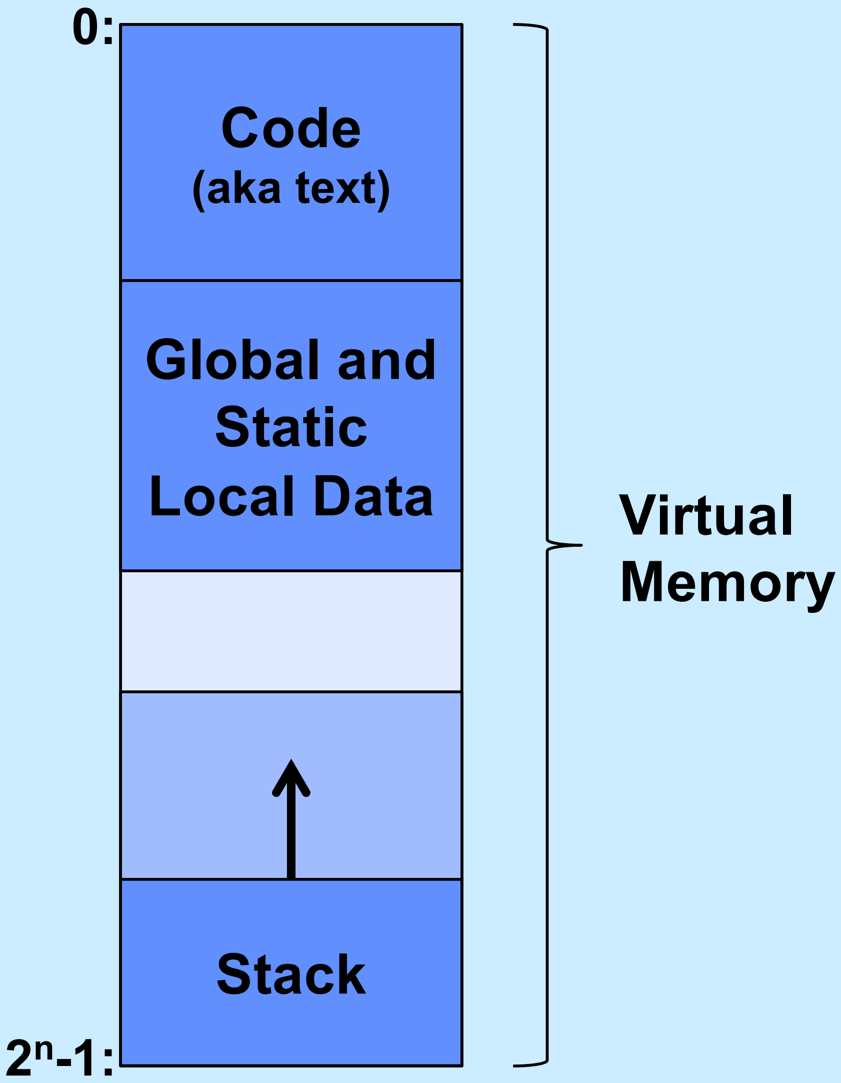
```
int sub() {
  static int svar = 2;
  int lvar = 1;
  svar += lvar;
  lvar++;
  return svar;
}

int main() {
  sub();
  printf("%d\n", sub());
  return 0;
}
```

**What is printed?**

a) 2
b) 3
c) 4
d) 5

# Digression: Where Stuff Is

## (Roughly)



# scanf: Reading Data

```
int main() {
   int i, j;
   scanf("%d %d", &i, &j);
   printf("%d, %d", i, j);
}
```

```
$ ./a.out
   3        12
3, 12
```

Two parts

- formatting instructions
  - whitespace in format string matches any amount of white space in input
    - » whitespace is space, tab, newline (‘\n’)
- arguments: must be addresses
  - why?

# #define (again)

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

**Simple textual substitution:**

```
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```

# Careful ...

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

```
float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*tempc+10)/5.0 + 32.0;
```

```
#define CtoF(cel) (9.0*(cel))/5.0 + 32.0
```

```
float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*(tempc+10))/5.0 + 32.0;
```

# Conditional Compilation

```
#ifdef DEBUG
  #define DEBUG_PRINT(a1, a2) printf(a1,a2)
#else
  #define DEBUG_PRINT(a1, a2)
#endif
```

```
int buggy_func(int x) {
   DEBUG_PRINT("x = %d\n", x);
     // printed only if DEBUG is defined
   ...
}
```

# Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x;
x.real = 1.4;
x.imag = 3.65e-10;
```

# Pointers to Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x, *y;
x.real = 1.4;
x.imag = 3.65e-10;
y = &x;
y->real = 2.6523;
y->imag = 1.428e20;
```

# *struct*s and Functions

```
struct ComplexNumber ComplexAdd(
            struct ComplexNumber a1,
            struct ComplexNumber a2) {
      struct ComplexNumber result;
      result.real = a1.real + a2.real;
      result.imag = a1.imag + a2.imag;
      return result;
}
```

# Would This Work?

```
struct ComplexNumber *ComplexAdd(
        struct ComplexNumber *a1,
        struct ComplexNumber *a2) {
    struct ComplexNumber result;
    result.real = a1->real + a2->real;
    result.imag = a1->imag + a2->imag;
    return &result;
}
```

# How About This?

```
void ComplexAdd(
     struct ComplexNumber *a1,
     struct ComplexNumber *a2,
     struct ComplexNumber *result) {
  result->real = a1->real + a2->real;
  result->imag = a1->imag + a2->imag;
  return;
}
```

# Using It ...

```
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber sum;

ComplexAdd(&j1, &j2, &sum);
```

# Arrays of *structs*

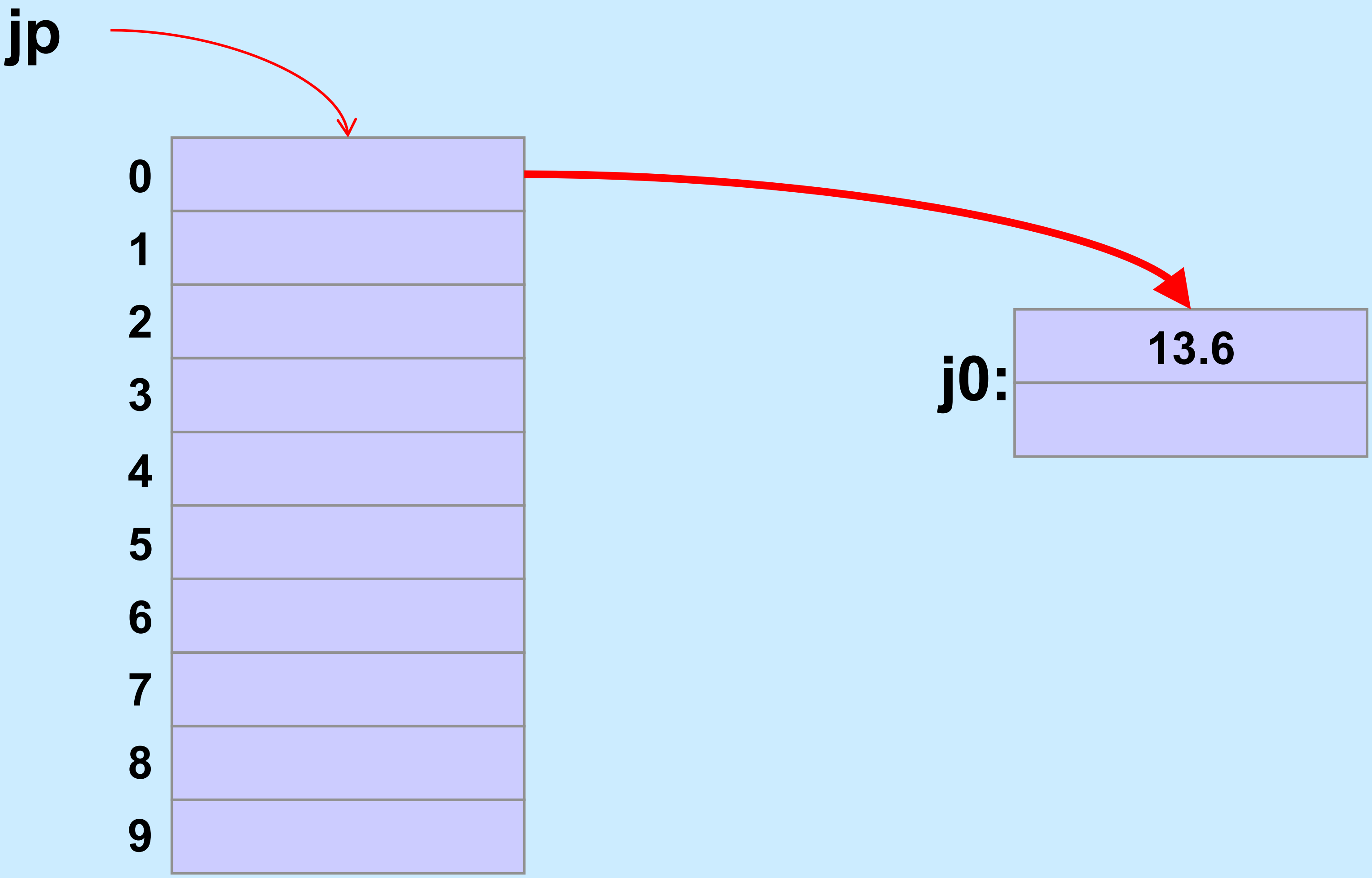
```
struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
```

# Arrays, Pointers, and *structs*

```
  /* What's this? */
  struct ComplexNumber *jp[10];
```

```
struct ComplexNumber j0;
jp[0] = &j0;
jp[0]->real = 13.6;
```

# Memory View



# Quiz 4

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a->val = 1;
   a->next = &b;
   b->val = 2;
   printf("%d\n", a->next->val);
   return 0;
}
```

- **What happens?**
  a) **prints something and terminates**
  b) **seg fault**
  c) **syntax error**

# Quiz 5

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   a.next = &b;
   b.val = 2;
   printf("%d\n", a.next.val);
   return 0;
}
```

- **What happens?**
  - **a) prints something and terminates**
  - **b) seg fault**
  - **c) syntax error**

# Quiz 6

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   b.val = 2;
   printf("%d\n", a.next->val);
   return 0;
}
```

- **What happens?**
  a) **prints something and terminates**
  b) **seg fault**
  c) **syntax error**

# Quiz 7

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   a.next = &b;
   b.val = 2;
   printf("%d\n", a.next->val);
   return 0;
}
```

- **What happens?**
  a) **prints something and terminates**
  b) **seg fault**
  c) **syntax error**

# Structures vs. Objects

- Are structs objects?

NO!

(What’s an object?)

# Structures Containing Arrays

```
struct Array {
   int A[6];
} S1, S2;

int A1[6], A2[6];

A1 = A2;
   // not legal: array variables refer to the
   // addresses of the first elements

S1 = S2;
   // legal: structure variables refer to contents
   // of the entire structure
```

# A Bit More Syntax …

- **Constants**

```
const double pi =
  3.14159265358979323 8;
```

```
area = pi*r*r;     /* legal */
pi = 3.0;          /* illegal */
```

# More Syntax …

```
const int six = 6;
int nonconstant;
const int *ptr_to_constant;
int *const constant_ptr = &nonconstant;
const int *const constant_ptr_to_constant = &six;

ptr_to_constant = &six;
   // ok
*ptr_to_constant = 7;
   // not ok
*constant_ptr = 7;
   // ok
constant_ptr = &six;
   // not ok
```

# And Still More ...

- **Array initialization**

```
int FirstSixPrimes[6] = {2, 3, 5, 7, 11, 13};
int SomeMorePrimes[] = {17, 19, 23, 29};
int MoreWithRoomForGrowth[10] = {31, 37};
int MagicSquare[][] = {{2, 7, 6},
                       {9, 5, 1},
                       {4, 3, 8}};
```

# Characters

- **ASCII**
  - American Standard Code for Information Interchange
  - works for:
    - » English
    - » Swahili
    - » not much else
  - doesn't work for:
    - » French
    - » Spanish
    - » German
    - » Korean
    - » Arabic
    - » Sanskrit
    - » Chinese
    - » pretty much everything else

# Characters

- Unicode
  - support for the rest of world
  - defines a number of encodings
  - most common is UTF-8
    - variable-length characters
    - ASCII is a subset and represented in one byte
    - larger character sets require an additional one to three bytes
  - not covered in CS 33

# ASCII Character Set

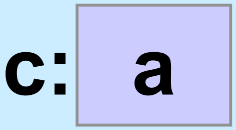
```
    00  10  20  30  40  50  60  70  80  90  100  110  120
 --------------------------------------------------------
0:  \0  \n          (   2   <   F   P   Z   d    n    x
1:      \v          )   3   =   G   Q   [   e    o    y
2:      \f      sp  *   4   >   H   R   \   f    p    z
3:      \r      !   +   5   ?   I   S   ]   g    q    {
4:              "   ,   6   @   J   T   ^   h    r    |
5:              #   -   7   A   K   U   _   i    s    }
6:              $   .   8   B   L   V   `   j    t    ~
7:  \a          %   /   9   C   M   W   a   k    u   DEL
8:  \b          &   0   :   D   N   X   b   l    v
9:  \t          '   1   ;   E   O   Y   c   m    w
```

# *chars* as Integers

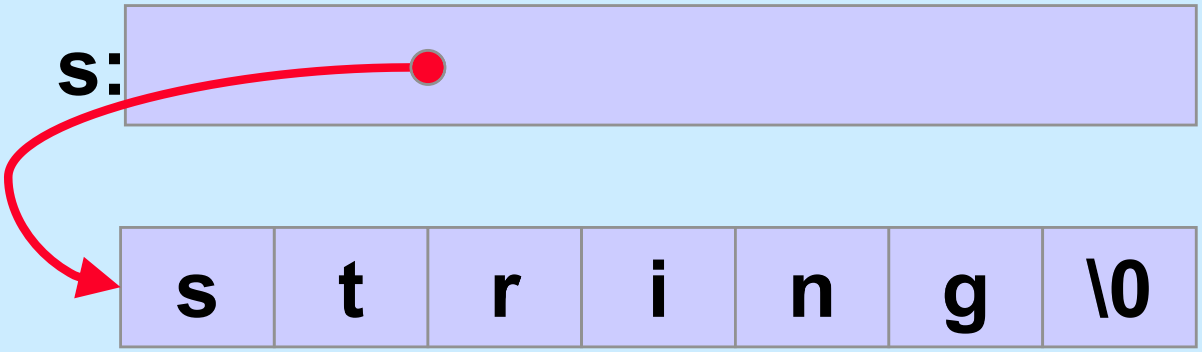
```
char tolower(char c) {
  if (c >= 'A' && c <= 'Z')
      return c + 'a' - 'A';
  else
      return c;
}
```

# Character Strings

```
char c = 'a';
```



```
char *s = "string";
```

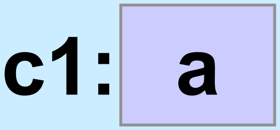


# Is there any difference between *c1* and *c2* in the following?

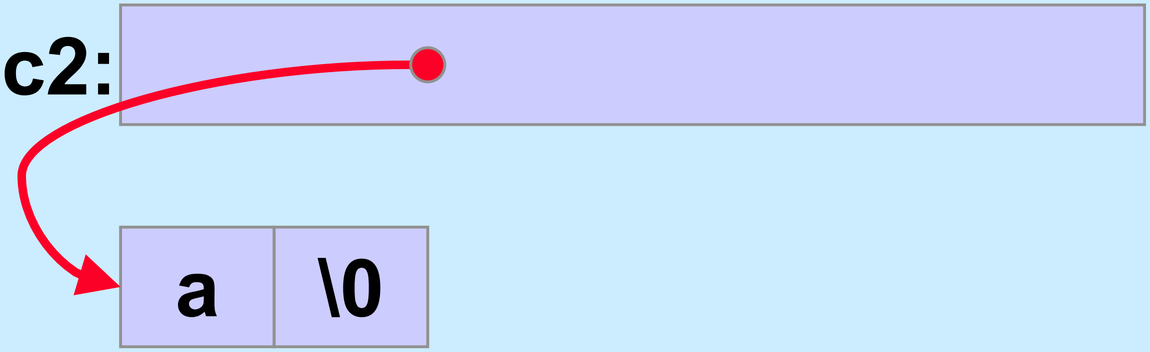
```
char c1 = 'a';
char *c2 = "a";
```

# Yes!!

```
char c1 = 'a';
```

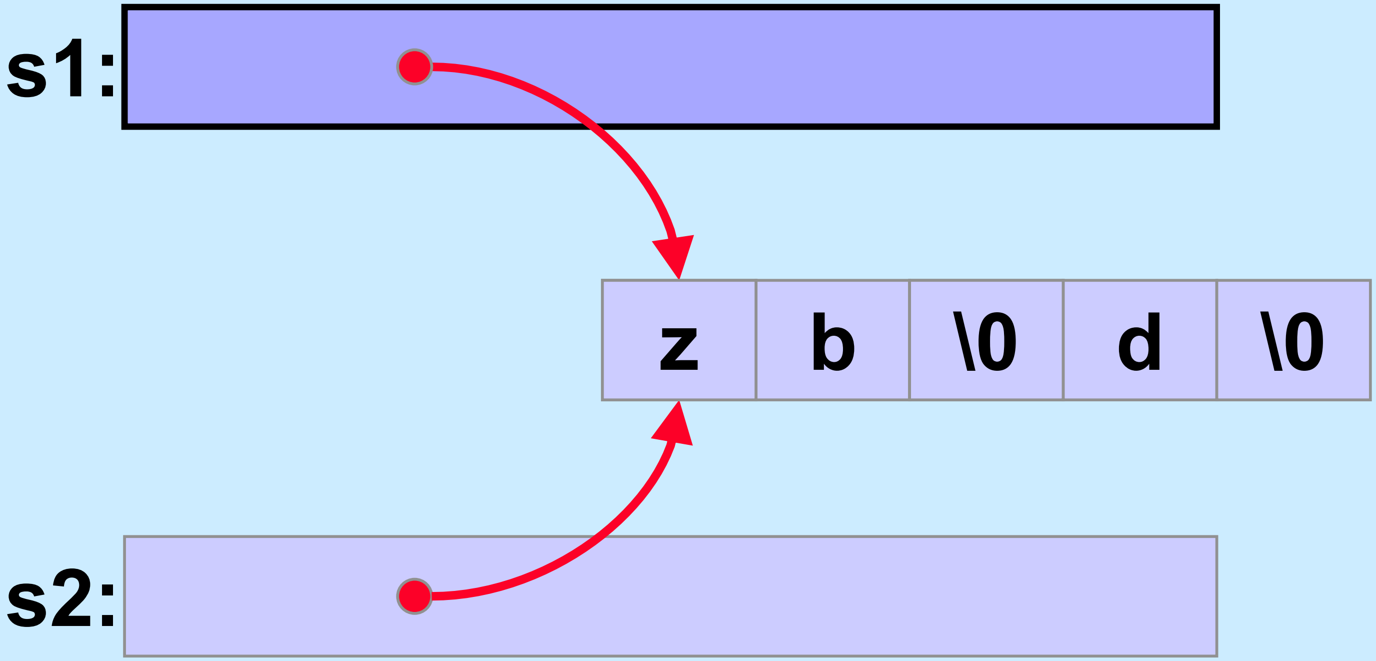


```
char *c2 = "a";
```



## What do *s1* and *s2* refer to after the following is executed?

```
char s1[] = "abcd";
char *s2 = s1;
s1[0] = 'z';
s2[2] = '\0';
```


s1:
z
b
\0
d
\0
s2:

# Weird ...

Suppose we did it this way:

```
char *s1 = "abcd";
char *s2 = s1;
s1[0] = 'z';
s1[2] = '\0';
```

```
% gcc -o char char.c
% ./char
Segmentation fault
```