Intermediate Programming

Day 10

Outline

- Exercise 9
- Pointers
- Review questions

Exercise 9

- Debug the program in `transpose.c`

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
void print( int table[][3] , int rows , int cols )
{
        for( int r=1 ; r<rows ; r++ )
        {
                for( int c=0 ; c<cols ; c++ )
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Debug the program in `transpose.c`

```
>> ./transpose
2nd table:
0 0 0
0 0 0
0 0 0
0 0 0
2nd after transpose:
2 7 12
3 8 13
32767 1256225752 0
0 0 0
>>
```

transpose.c

```c
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
void print( int table[][3] , int rows , int cols )
{
        for( int r=1 ; r<rows ; r++ )
        {
                for( int c=0 ; c<cols ; c++ )
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Run gdb

```
>> gdb ./transpose
...
(gdb)
```

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

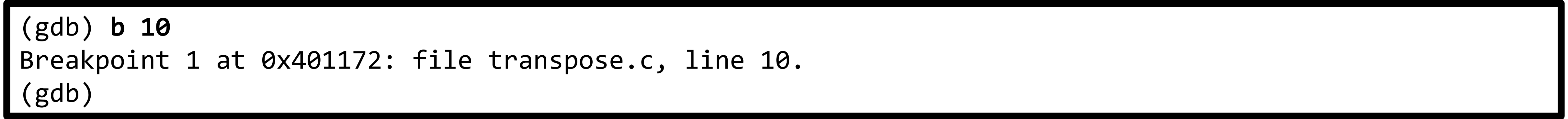
- Add a breakpoint at line 10

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers */
```



```
(gdb) b 10
Breakpoint 1 at 0x401172: file transpose.c, line 10.
(gdb)
```

```
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Run to the breakpoint

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
	int d1 = 3, d2 = 5;
	for( int r=0 ; r<d1 ; r++ )
		for( int c=0 ; c<d2 ; c++ )
			end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
	return 0;
}
```

```
(gdb) r
Starting program: /users/misha/transpose
...
2nd table:
0 0 0
0 0 0
0 0 0
0 0 0
2nd after transpose:

Breakpoint 1, transpose (start=0x7fffffffdcb0, end=0x7fffffffdc70) at transpose.c:10
10                              end[r][c] = start[c][r];
...
(gdb)
```

Exercise 9

- Display the variables **r** and **c**

```
(gdb) display {r,c}
1: {r,c} = {0, 0}
(gdb)
```

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Display the **start** array

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
(gdb) display {start[0],start[1],start[2]}
2: {start[0],start[1],start[2]} = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}}
(gdb)
```

```
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Display the **end** array

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
(gdb) display {end[0],end[1],end[2],end[3],end[4]}
3: {end[0],end[1],end[2],end[3],end[4]} = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
(gdb)
```

```
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Continue to the next breakpoint

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers */
```

```
(gdb) c
Continuing.

Breakpoint 1, transpose (start=0x7fffffffdcb0, end=0x7fffffffdc70) at transpose.c:10
10                              end[r][c] = start[c][r];
1: {r,c} = {0, 1}
2: {start[0],start[1],start[2]} = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}}
3: {end[0],end[1],end[2],end[3],end[4]} = {{1, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
(gdb)
```

```
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Continue to the next breakpoint

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
(gdb)
Continuing.

Breakpoint 1, transpose (start=0x7fffffffdcb0, end=0x7fffffffdc70) at transpose.c:10
10                              end[r][c] = start[c][r];
1: {r,c} = {0, 2}
2: {start[0],start[1],start[2]} = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}}
3: {end[0],end[1],end[2],end[3],end[4]} = {{1, 6, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
(gdb)
```

```
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Continue to the next breakpoint

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

```
(gdb)
Continuing.

Breakpoint 1, transpose (start=0x7fffffffdcb0, end=0x7fffffffdc70) at transpose.c:10
10                              end[r][c] = start[c][r];
1: {r,c} = {0, 3}
2: {start[0],start[1],start[2]} = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}}
3: {end[0],end[1],end[2],end[3],end[4]} = {{1, 6, 11}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0, 0}}
(gdb)
```

Exercise 9

- Continue to the next breakpoint

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[r][c] = start[c][r];
}

/* Print a 2D array of integers  */
```

```
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

```
(gdb)
Continuing.

Breakpoint 1, transpose (start=0x7fffffffdcb0, end=0x7fffffffdc70) at transpose.c:10
10                                end[r][c] = start[c][r];
1: {r,c} = {0, 4}
2: {start[0],start[1],start[2]} = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}}
3: {end[0],end[1],end[2],end[3],end[4]} = {{1, 6, 11}, {32767, 0, 0}, {0, 0, 0}, {0, 0, 0}, {0, 0,
0}}(gdb)
```

Exercise 9

- Fix the code and re-run

```
>> ./transpose
2nd table:
0 0 0
0 0 0
0 0 0
0 0 0
2nd after transpose:
2 7 12
3 8 13
4 9 14
5 10 15
>>
```

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[c][r] = start[r][c];
}

/* Print a 2D array of integers  */
void print( int table[][3] , int rows , int cols )
{
        for( int r=1 ; r<rows ; r++ )
        {
                for( int c=0 ; c<cols ; c++ )
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Re-run the debugger and identify that we only start printing from the second row.

transpose.c

```
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[c][r] = start[r][c];
}

/* Print a 2D array of integers  */
void print( int table[][3] , int rows , int cols )
{
        for( int r=1 ; r<rows ; r++ )
        {
                for( int c=0 ; c<cols ; c++ )
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Exercise 9

- Fix the code and re-run

```
>> ./transpose
2nd table:
0 0 0
0 0 0
0 0 0
0 0 0
2nd after transpose:
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15
>>
```

transpose.c

```c
#include <stdio.h>
#include <string.h>

/* Transpose from a 3x5 array (start) into a 5x3 array (end) of integers. */
void transpose( int start[][5] , int end[][3] )
{
        int d1 = 3, d2 = 5;
        for( int r=0 ; r<d1 ; r++ )
                for( int c=0 ; c<d2 ; c++ )
                        end[c][r] = start[r][c];
}

/* Print a 2D array of integers  */
void print( int table[][3] , int rows , int cols )
{
        for( int r=0 ; r<rows ; r++ )
        {
                for( int c=0 ; c<cols ; c++ )
                        printf("%d ", table[r][c]);
                printf("\n");
        }
}

int main()
{
        int one[3][5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
        int two[5][3] = {0};

        printf( "2nd table:\n" );
        print( two , 5 , 3 );
        printf( "2nd after transpose:\n" );
        transpose( one , two );
        print( two , 5 , 3 );
        return 0;
}
```

Outline

- Exercise 9
- Pointers
- Review questions

Writing a **swap** function in C

Q: Why doesn’t this code work?

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

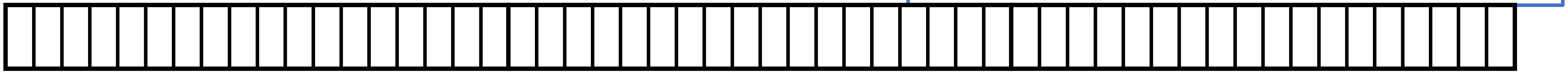
```
>> ./a.out
1 2
>>
```

Writing a **swap** function in C

Q: Why doesn’t this code work?

- Variables reside somewhere in memory.

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

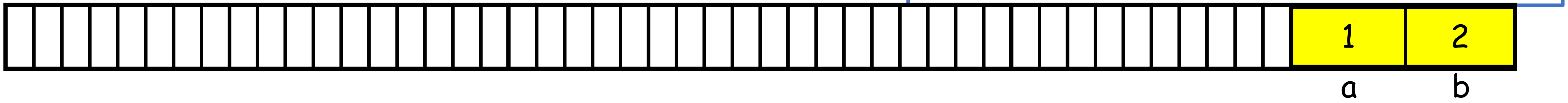


Writing a **swap** function in C

Q: Why doesn’t this code work?

- Variables reside somewhere in memory.
- When main is compiled, its variables are bound to a memory location.

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

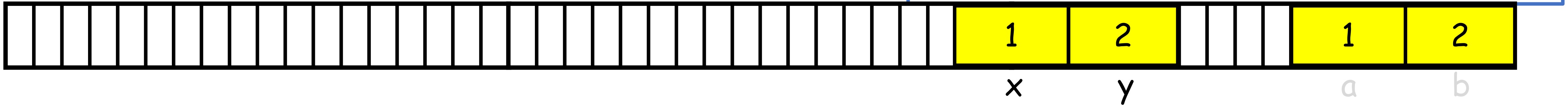


Writing a **swap** function in C

Q: Why doesn’t this code work?

- Variables reside somewhere in memory.
- When **main** is compiled, its variables are bound to a memory location.
- When we call **swap**, the arguments are duplicated (to a new memory location).

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

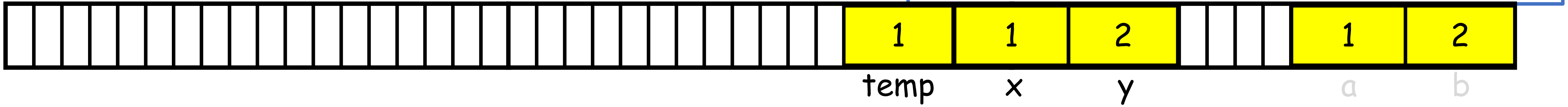


Writing a **swap** function in C

Q: Why doesn't this code work?

- Variables reside somewhere in memory.
- When **main** is compiled, its variables are bound to a memory location.
- When we call **swap**, the arguments are duplicated (to a new memory location).

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

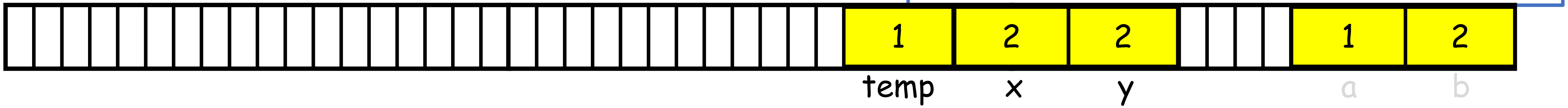


Writing a **swap** function in C

Q: Why doesn’t this code work?

- Variables reside somewhere in memory.
- When **main** is compiled, its variables are bound to a memory location.
- When we call **swap**, the arguments are duplicated (to a new memory location).

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

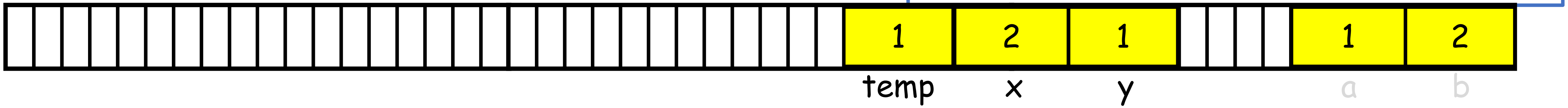


Writing a **swap** function in C

Q: Why doesn't this code work?

- Variables reside somewhere in memory.
- When **main** is compiled, its variables are bound to a memory location.
- When we call **swap**, the arguments are duplicated (to a new memory location).

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



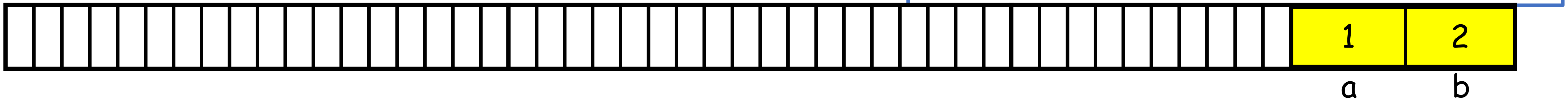
Writing a **swap** function in C

Q: Why doesn’t this code work?

- Variables reside somewhere in memory.
- When **main** is compiled, its variables are bound to a memory location.
- When we call **swap**, the arguments are duplicated (to a new memory location).

⇒ **swap** has a copy of the variables, so changes to the variables in **swap** are invisible to **main**.

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
        x = y;
        y = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( a , b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



Writing a **swap** function in C

```
#include <stdio.h>
void swap( int x , int y )
{
        int temp = x;
```

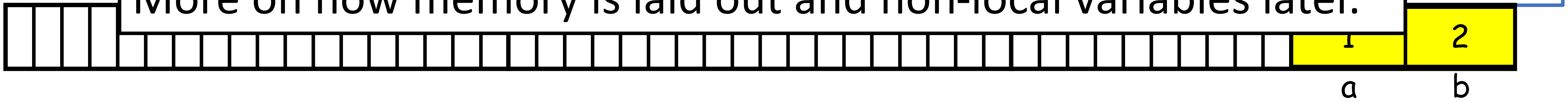
Q: Why doesn’t this code work?

- Variables reside somewhere in memory.

Warning:

- This representation of memory is a little simplistic.
- Recall that functions are associated with stack frames on the call stack.
- In addition to storing who call the function, a stack frame also stores the (local) variables used by the function.
- This is why the variables **x**, **y**, and **temp** “disappear” after we return from the **swap** function.

More on how memory is laid out and non-local variables later.



Pointers

- A *pointer* is a variable that stores a memory address/location
  - Every pointer points to a specific data type
    (except a pointer to void, more on that later)
    - Describes "what kind of variable resides at this memory address/location"
  - Declare a pointer using type of variable it will point to, and a "*":
    - "int *iP" is a pointer to an int
    - "double* dP" is a pointer to a double
    - "char * cP" is a pointer to a char

    (Note that spaces are not important)
- Operations related to pointers
  - variable to pointer: operator "&" returns address of whatever follows it
  - pointer to variable: operator "*" returns value being pointed to (dereferencing)

Pointers

- A *pointer* is a variable that stores a memory address/location
  - Every pointer points to a specific data type
    (except a pointer to void, more on that later)
    - Describes "what kind of variable resides at this memory address/location"
  - Declare a pointer using type of variable it will point to, and a "*****":
    - "int *iP" is a pointer to an int
    - "double* dP" is a pointer to a double
    - "char * cP" is a pointer to a char

    (Note that spaces are not important)

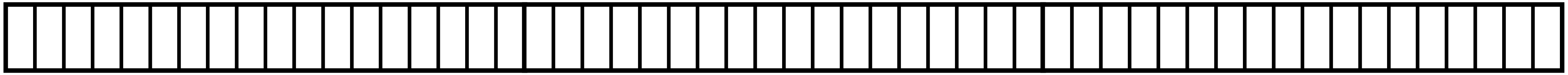
<u>Note</u>:

When declaring a pointer, the "*****" needs to be associated with the variable name, not the type

- int * a , b; ⇔ declares a pointer to an int called a and an int called b
- int * a , * b; ⇔ declares a pointer to an int called a and a pointer to an int called b

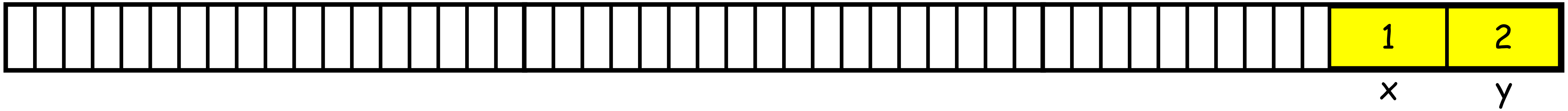
Pointers

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;            // a pointer to an int
        iP = &x;            // iP points to x
        y = *iP;            // y has the value of what iP points to (x)
        *iP = 0;            // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```



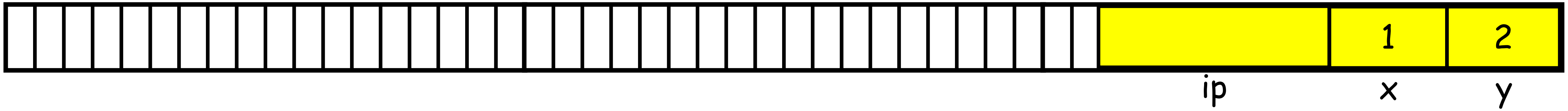
Pointers

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;          // a pointer to an int
        iP = &x;          // iP points to x
        y = *iP;          // y has the value of what iP points to (x)
        *iP = 0;          // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```



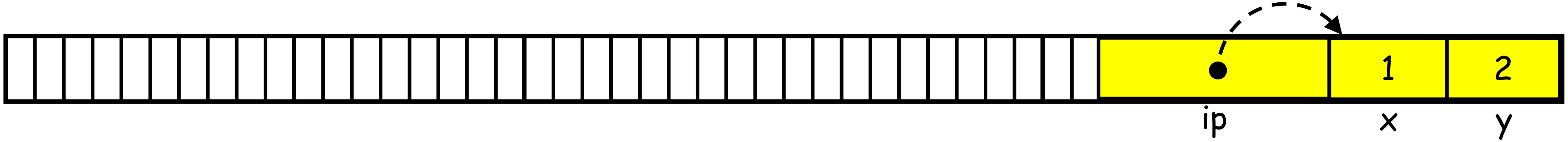
Pointers

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;            // a pointer to an int
        iP = &x;            // iP points to x
        y = *iP;            // y has the value of what iP points to (x)
        *iP = 0;            // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```



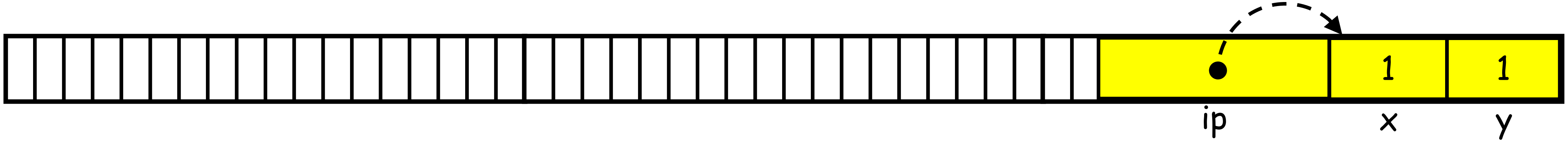
Pointers

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;          // a pointer to an int
        iP = &x;          // iP points to x
        y = *iP;          // y has the value of what iP points to (x)
        *iP = 0;          // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```



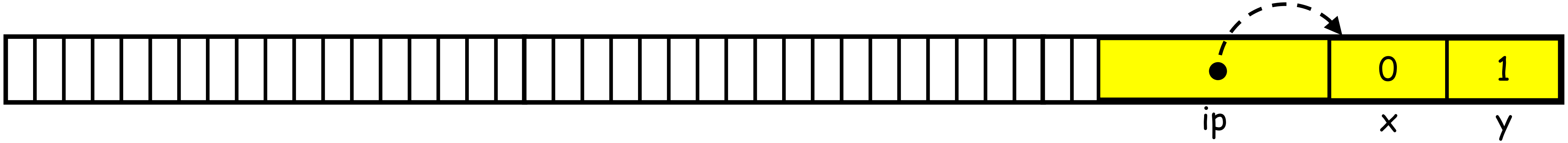
Pointers

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;           // a pointer to an int
        iP = &x;           // iP points to x
        y = *iP;           // y has the value of what iP points to (x)
        *iP = 0;           // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```



Pointers

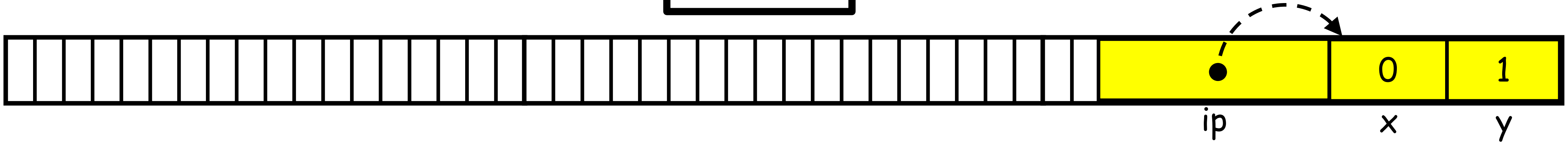
```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;             // a pointer to an int
        iP = &x;             // iP points to x
        y = *iP;             // y has the value of what iP points to (x)
        *iP = 0;             // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```



Pointers

```
#include <stdio.h>
int main( void )
{
        int x = 1 , y = 2;// ints
        int *iP;              // a pointer to an int
        iP = &x;              // iP points to x
        y = *iP;              // y has the value of what iP points to (x)
        *iP = 0;              // what iP points to (x) has value 0
        printf( "%d %d\n" , x , y );
        return 0;
}
```

```
>> ./a.out
0 1
>>
```



A working **swap** function

- The call in main is now swap(&a , &b) since we pass the addresses of a and b
- Pointer arguments allow swap to access and modify values in main

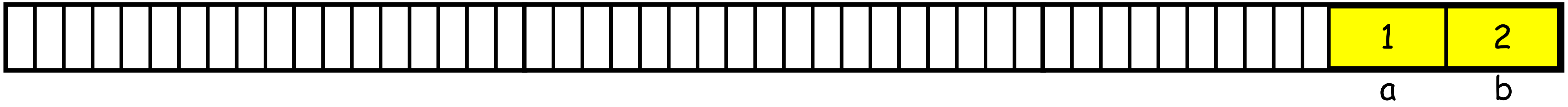
```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

```
>> ./a.out
2 1
>>
```

A working **swap** function

- The call in main is now swap(&a , &b) since we pass the addresses of a and b
- Pointer arguments allow swap to access and modify values in main

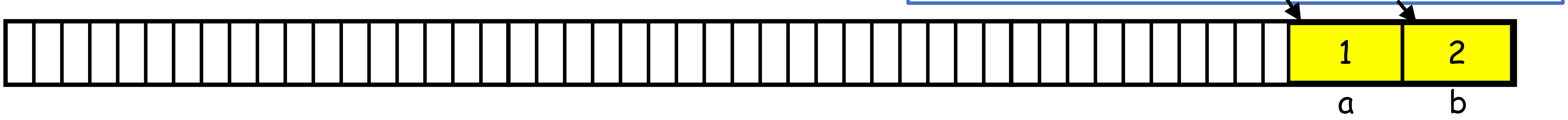
```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



A working **swap** function

- The call in main is now swap(&a , &b) since we pass the addresses of a and b
- Pointer arguments allow swap to access and modify values in main

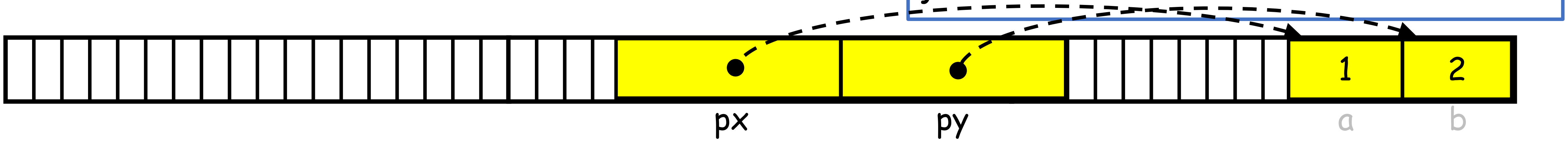
```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



A working **swap** function

- The call in main is now swap(&a , &b) since we pass the addresses of a and b
- Pointer arguments allow swap to access and modify values in main

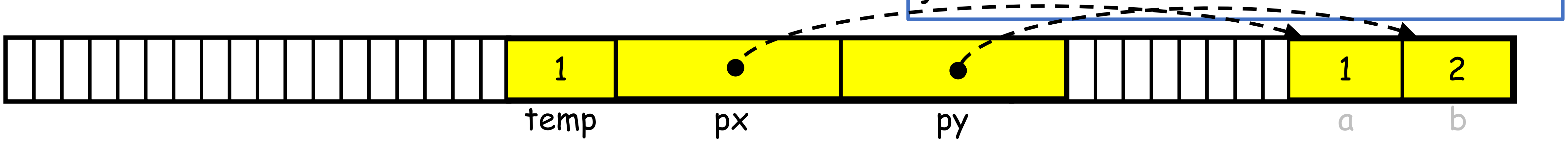
```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



A working **swap** function

- The call in **main** is now **swap(&a , &b)** since we pass the addresses of **a** and **b**
- Pointer arguments allow **swap** to access and modify values in **main**

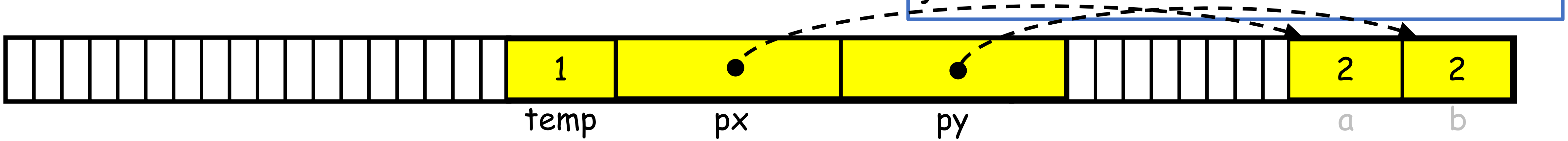
```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



A working **swap** function

- The call in main is now swap(&a , &b) since we pass the addresses of a and b
- Pointer arguments allow swap to access and modify values in main

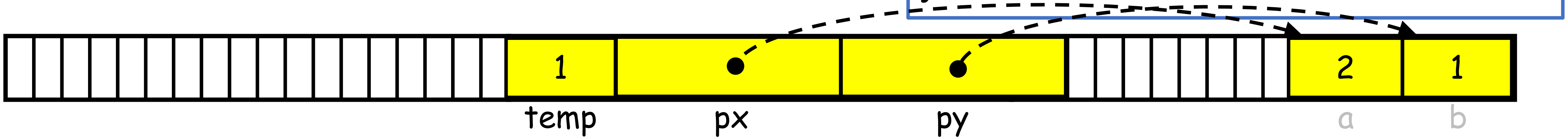
```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```



A working **swap** function

- The call in **main** is now **swap(&a , &b)** since we pass the addresses of **a** and **b**
- Pointer arguments allow **swap** to access and modify values in **main**

```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return 0;
}
```

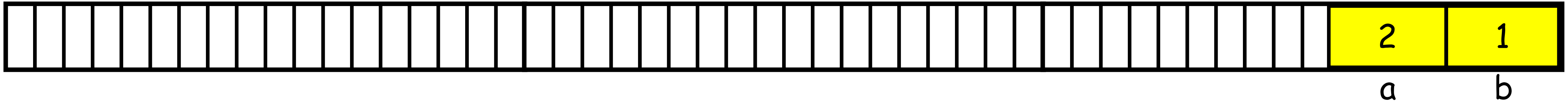


A working **swap** function

- The call in main is now swap(&a , &b) since we pass the addresses of a and b
- Pointer arguments allow swap to access and modify values in main

```
#include <stdio.h>
void swap( int *px , int *py )
{
        int temp = *px;
        *px = *py;
        *py = temp;
}
int main( void )
{
        int a = 1 , b = 2;
        swap( &a , &b );
        printf( "%d %d\n" , a , b );
        return
}
```

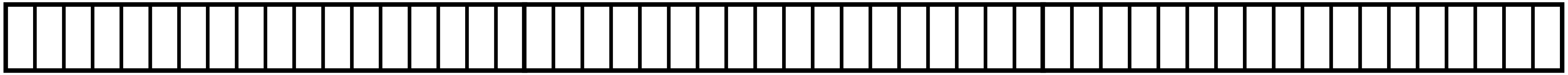
```
>> ./a.out
2 1
>>
```



Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

```
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 };
    int *v2 = v1;
    v1[0] = 1;
    v2[1] = 2;
    printf( "%d %d %d\n" , *v1 , v1[1] , v1[2] );
    printf( "%d %d %d\n" , *v2 , v2[1] , v2[2] );
    return 0;
}
```

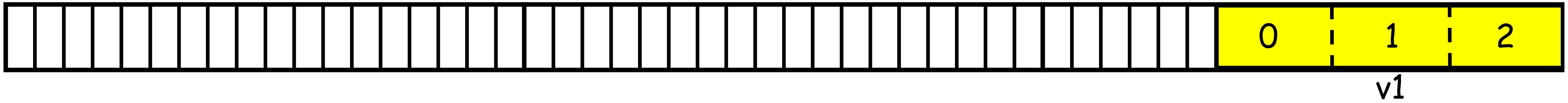


memory

Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

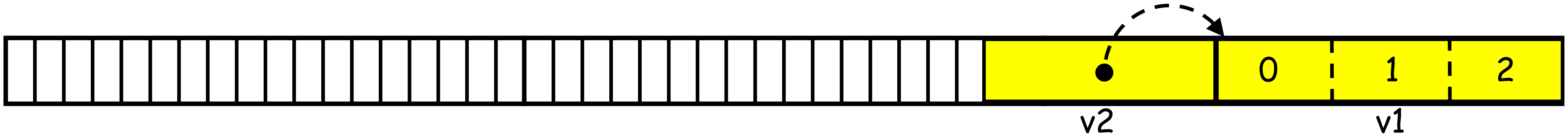
```
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 };
    int *v2 = v1;
    v1[0] = 1;
    v2[1] = 2;
    printf( "%d %d %d\n" , *v1 , v1[1] , v1[2] );
    printf( "%d %d %d\n" , *v2 , v2[1] , v2[2] );
    return 0;
}
```



Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

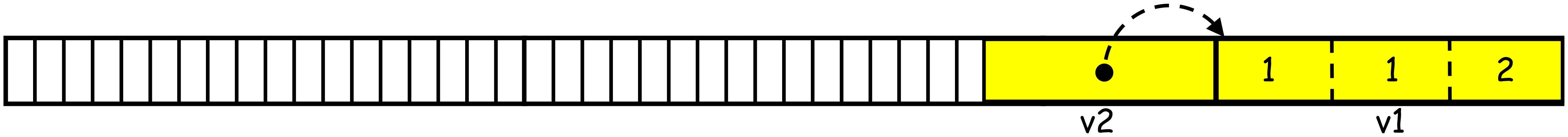
```
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 };
    int *v2 = v1;
    v1[0] = 1;
    v2[1] = 2;
    printf( "%d %d %d\n" , *v1 , v1[1] , v1[2] );
    printf( "%d %d %d\n" , *v2 , v2[1] , v2[2] );
    return 0;
}
```



Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

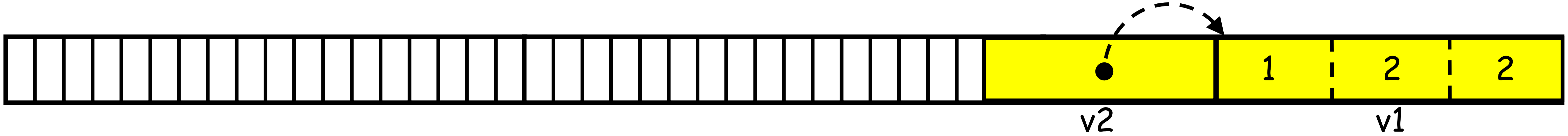
```
#include <stdio.h>
int main( void )
{
     int v1[] = { 0 , 1 , 2 };
     int *v2 = v1;
     v1[0] = 1;
     v2[1] = 2;
     printf( "%d %d %d\n" , *v1 , v1[1] , v1[2] );
     printf( "%d %d %d\n" , *v2 , v2[1] , v2[2] );
     return 0;
}
```



Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

```
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 };
    int *v2 = v1;
    v1[0] = 1;
    v2[1] = 2;
    printf( "%d %d %d\n" , *v1 , v1[1] , v1[2] );
    printf( "%d %d %d\n" , *v2 , v2[1] , v2[2] );
    return 0;
}
```

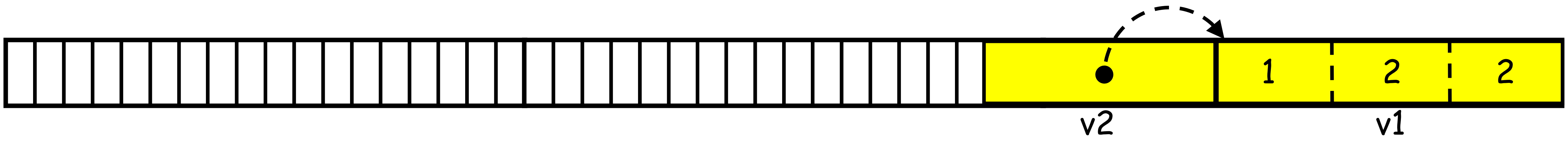


Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

```
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 };
    int *v2 = v1;
    v1[0] = 1;
    v2[1] = 2;
    printf( "%d %d %d\n" , *v1 , v1[1] , v1[2] );
    printf( "%d %d %d\n" , *v2 , v2[1] , v2[2] );
    return 0;
}
```

```
>> ./a.out
1 2 2
1 2 2
>>
```



Pointer access

- In C, nothing can reside at memory address 0.

⇒ The null pointer is a special pointer defined to point to address 0.

  - The variable NULL is defined to be a pointer to address 0.
  - This is often returned when a function that is meant to return a pointer fails.

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    FILE *fp = fopen( "misha.txt" , "r" );
    if( fp==NULL )
    {
        fprintf( stderr , "[ERROR] …" );
        return 1;
    }
    return 0;
}
```

Pointer access

- In C, nothing can reside at memory address 0.

⇒ The null pointer is a special pointer defined to point to address 0.

  - The variable NULL is defined to be a pointer to address 0.
  - This is often returned when a function that is meant to return a pointer fails.

```
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
    FILE *fp = fopen( "misha.txt" , "r" );
    if( !fp )
    {
        fprintf( stderr , "[ERROR] …" );
        return 1;
    }
    return 0;
}
```

Since NULL is the same as zero,
we can just check if fp is zero.

Pointer access

- In C, nothing can reside at memory address 0.

⇒ The null pointer is a special pointer defined to point to address 0.

  - The variable NULL is defined to be a pointer to address 0.
  - This is often returned when a function that is meant to return a pointer fails.
  - Trying to access an entry at the zero address will cause bad behavior so make sure to check that a pointer is valid before trying to use it.

```
#include <stdio.h>
int main( void )
{
    int *arr = NULL;
    printf( "Value = %d\n" , arr[0] );
    return 0;
}
```

```
>> ./a.out
Segmentation fault (core dumped)
>>
```

Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.

```c
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 };
    int *v2 = v1;
    printf( "%d\n" , (int)(v2-v1) );
    printf( "%p %p\n" , (void*)v1 , (void*)v2 );
    return 0;
}
```

```
>> ./a.out
0
0x7fff6783e980 0x7fff6783e980
>>
```

The “0x” prefix indicates that the number is represented in hexadecimal notation (base 16).*

*More on this later

Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.
  - The big difference is how `sizeof` behaves within the body where the (static) array is defined.
    - The array has `sizeof` 16 bytes since it consists of four 4-byte integers
    - The pointer has `sizeof` 8 since memory addresses are 8 bytes long on 64-bit architectures.

```c
#include <stdio.h>
int main( void )
{
    int v1[] = { 0 , 1 , 2 , 3 };
    int *v2 = v1;
    printf( "%d %d\n" ,
        (int)sizeof( v1 ) ,
        (int)sizeof( v2 ) );
    return 0;
}
```

```
>> ./a.out
16 8
>>
```

Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.
  - The big difference is how `sizeof` behaves within the body where the array is defined.
  - If you pass the array to a function it gets “downgraded” to a pointer.

```
#include <stdio.h>
void print_size( const int *a )
{
    printf( "%d\n" , (int)sizeof( a ) );
}
int main( void )
{
    int v1[] = { 0 , 1 , 2 , 3 };
    int *v2 = v1;
    print_size( v1 );
    print_size( v2 );
    return 0;
}
```

```
>> ./a.out
8
8
>>
```

Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.
  - The big difference is how sizeof behaves within the body where the array is defined.
  - If you pass the array to a function it gets “downgraded” to a pointer.

```
#include <stdio.h>
void print_size( const int a[] )
{
    printf( "%d\n" , (int)sizeof( a ) );
}
int main( void )
{
    int v1[] = { 0 , 1 , 2 , 3 };
    int *v2 = v1;
    print_size( v1 );
    print_size( v2 );
    return 0;
}
```

```
>> ./a.out
8
8
>>
```

Pointers vs. arrays

- For the most part, pointers and arrays are the same thing.
  - The big difference is how `sizeof` behaves within the body where the array is defined.
  - If you pass the array to a function it gets “downgraded” to a pointer.

```c
#include <stdio.h>
void print_size( const int a[4] )
{
    printf( "%d\n" , (int)sizeof( a ) );
}
int main( void )
{
    int v1[] = { 0 , 1 , 2 , 3 };
    int *v2 = v1;
    print_size( v1 );
    print_size( v2 );
    return 0;
}
```

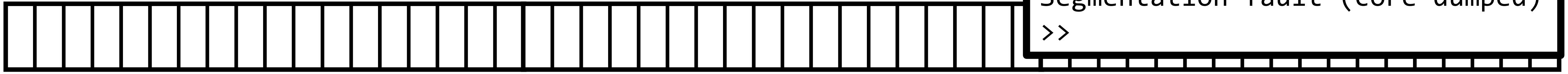
```
>> ./a.out
8
8
>>
```

Returning an array in C

Q: Why doesn’t this code work?

```
#include <stdio.h>
int * getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

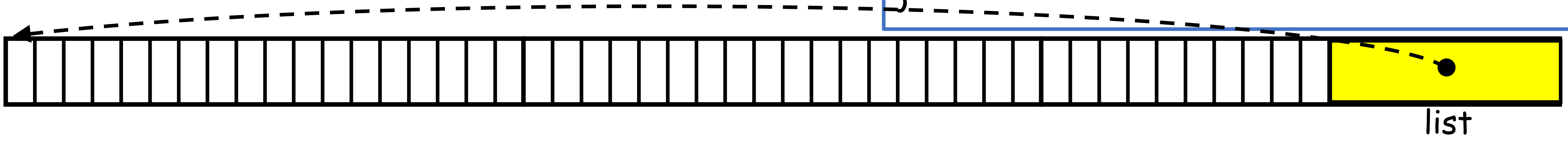
```
>> ./a.out
Segmentation fault (core dumped)
>>
```



Returning an array in C

Q: Why doesn’t this code work?

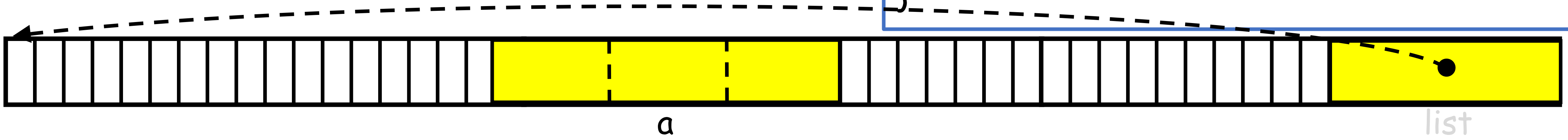
```
#include <stdio.h>
int * getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```



Returning an array in C

Q: Why doesn’t this code work?

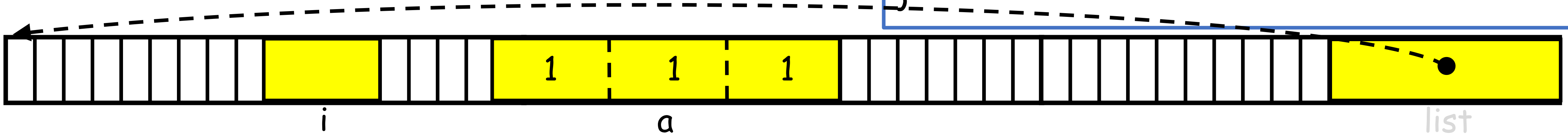
```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```



Returning an array in C

Q: Why doesn’t this code work?

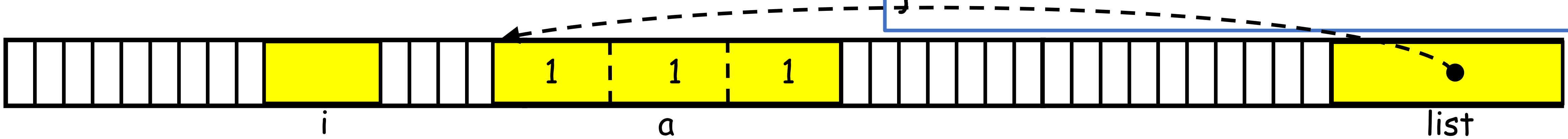
```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```



Returning an array in C

Q: Why doesn't this code work?

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```

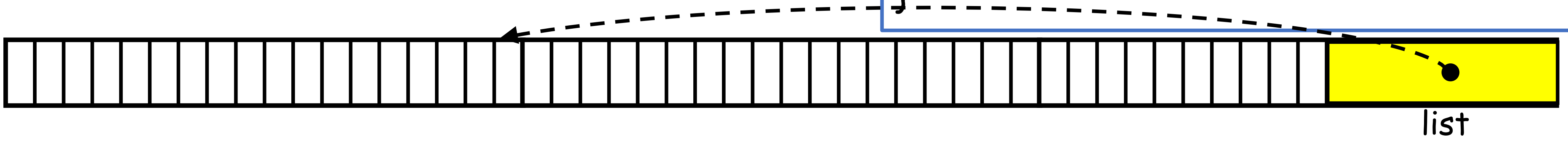


Returning an array in C

Q: Why doesn’t this code work?

A: Recall that a lives on the stack frame of getArray3.
When list is assigned the address a, that stack frame no longer exists on the call stack, so the address is no longer valid.

```
#include <stdio.h>
int *getArray3( void )
{
        int a[3];
        for( int i=0 ; i<3 ; i++ ) a[i] = 1;
        return a;
}
int main( void )
{
        int *list = NULL;
        list = getArray3();
        for( int i=0 ; i<3 ; i++ )
                printf( "%d " , list[i] );
        printf( "\n" );
        return 0;
}
```



Outline

- Exercise 9
- Pointers
- Review questions

Review questions

1. What is a pointer?

A pointer is a type describing a location in memory (as well as the type being stored there)

Review questions

2. If a is an int variable and p is a variable whose type is *pointer-to-int*, how do you make p point to a?

```
p = &a;
```

Review questions

3. If p is a *pointer-to-int* variable that points to an int variable a, how can you access the value of a or assign a value to a without directly referring to a? Show examples of printing the value of a and modifying the value of a, but without directly referring to a.

```
*p = 5;
```

Review questions

4. When calling scanf, why do you need to put a & symbol in front of a variable in which you want scanf to store an input value?

We pass the address of the variable we want scanf to set so that it can make changes to the variable (not its copy)

Review questions

5. Trace the program below and determine what the output will be.

```
int func( float ra[] , float x , float *y )
{
        ra[0] += 10;
        x *= 20;
        *y += 30;
        return 40;
}
int main( void )
{
        float a = 1;
        float b = 2;
        float c[] = { 3 , 4 , 5 , 6 };
        float d;
        d = func( c , a , &b );
        printf( "%f, %f, %f, %d\n“ , a , b , c[0] , d );
}
```

```
>> ./a.out
1.000000, 32.000000, 13.000000, -2126392028
>>
```

Exercise 4-1

- Website -> Course Materials -> Ex4-1