ECE 220

Lecture x0009 - 02/13

Slides based on material originally by: Yuting Chen & Thomas Moon

Reminders

Reminders

- No lecture on Thursday

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook
  - https://hkn.illinois.edu/services

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook
  - https://hkn.illinois.edu/services
- Format

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook
  - https://hkn.illinois.edu/services
- Format
  - Four questions in total

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook
  - https://hkn.illinois.edu/services
- Format
  - Four questions in total
  - Two larger writing LC3

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook
  - https://hkn.illinois.edu/services
- Format
  - Four questions in total
  - Two larger writing LC3
  - One debugging LC3

Reminders

- No lecture on Thursday
- Midterm 1 at 1900 hrs CT on 02/15
- Check the course website for room assignments
- Material:
  - Lectures 1 - 6 + textbook
  - https://hkn.illinois.edu/services
- Format
  - Four questions in total
  - Two larger writing LC3
  - One debugging LC3
  - Conceptual questions (including C)

Recap

Recap

- Functions in C

Recap

- Functions in C
  - Similarity to subroutines

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack
- LC3 implementation

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack
- LC3 implementation
  - `R0 - R3` caller saved

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack
- LC3 implementation
  - `R0` – `R3` caller saved
  - `R4` - global variable

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack
- LC3 implementation
  - `R0 – R3` caller saved
  - `R4` - global variable
  - `R5` - stack frame

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack
- LC3 implementation
  - `R0 – R3` caller saved
  - `R4` - global variable
  - `R5` - stack frame
  - `R6` - top of stack

Recap

- Functions in C
  - Similarity to subroutines
  - Prototype vs. definition
  - Parameters & return types
  - Low-level implementation
  - Run time stack
- LC3 implementation
  - `R0` – `R3` caller saved
  - `R4` - global variable
  - `R5` - stack frame
  - `R6` - top of stack
  - `R7` - for `RET` instruction

Swap function

Swap function

```
void Swap(int first, int second);

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(valueA, valueB);
}

void Swap(int first, int second){
   int temp;
   temp = first;
   first = second;
   second = temp;
}
```

Swap function

```
void Swap(int first, int second);

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(valueA, valueB);
}

void Swap(int first, int second){
   int temp;
   temp = first;
   first = second;
   second = temp;
}
```

- Did this function from last time work?

Swap function

```c
void Swap(int first, int second);

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(valueA, valueB);
}

void Swap(int first, int second){
   int temp;
   temp = first;
   first = second;
   second = temp;
}
```

- Did this function from last time work?
- What needed to be changed for the `swap` function to work?

Swap function

```c
void Swap(int first, int second);

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(valueA, valueB);
}

void Swap(int first, int second){
   int temp;
   temp = first;
   first = second;
   second = temp;
}
```

- Did this function from last time work?
- What needed to be changed for the `swap` function to work?
- Somehow the `swap` function needs to know the *memory locations* of the variables that `main` needs swapped

Swap function

```c
void Swap(int first, int second);

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(valueA, valueB);
}

void Swap(int first, int second){
   int temp;
   temp = first;
   first = second;
   second = temp;
}
```

- Did this function from last time work?
- What needed to be changed for the `swap` function to work?
- Somehow the `swap` function needs to know the *memory locations* of the variables that `main` needs swapped
- Enter **pointers.**

Pointers in C

Pointers in C

- A **pointer** in C, is a variable whose value is the address of another variable, i.e., direct address of the memory location.

Pointers in C

- A **pointer** in C, is a variable whose value is the address of another variable, i.e., direct address of the memory location.

- Memory location utilized by a variable is obtained using the *address-of* operator `&`. For example:

Pointers in C

- A **pointer** in C, is a variable whose value is the address of another variable, i.e., direct address of the memory location.

- Memory location utilized by a variable is obtained using the *address-of* operator `&`. For example:

  `&val` → ***address operator***: returns address of variable `val`

Pointers in C

- A **pointer** in C, is a variable whose value is the address of another variable, i.e., direct address of the memory location.

- Memory location utilized by a variable is obtained using the *address-of* operator `&`. For example:

  `&val` → ***address operator***: returns address of variable `val`

- The declaration syntax for a pointer is:

Pointers in C

- A **pointer** in C, is a variable whose value is the address of another variable, i.e., direct address of the memory location.

- Memory location utilized by a variable is obtained using the *address-of* operator `&`. For example:

  `&val` → ***address operator***: returns address of variable `val`

- The declaration syntax for a pointer is:

```
type *ptr-name;
```

Pointers in C

Example declarations

```
int *ptr; // ptr is a pointer to an int

char *cptr; // cptr is a pointer to _____

double *dptr; // dptr is a pointer to _____
```

Simple pointers

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location
- Wild pointer: An *uninitialized* pointer, i.e. it points to some garbage memory address. Be careful with them.

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location
- Wild pointer: An *uninitialized* pointer, i.e. it points to some garbage memory address. Be careful with them.
- Null pointer: A standard pointer to the `NULL` value/location.

Needs. `#include <stddef.h>`

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location
- Wild pointer: An *uninitialized* pointer, i.e. it points to some garbage memory address. Be careful with them.
- Null pointer: A standard pointer to the `NULL` value/location.

  Needs. `#include <stddef.h>`

- Dangling pointer: A standard pointer which points to a memory location that was *deallocated* (more in later lectures).

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location
- Wild pointer: An *uninitialized* pointer, i.e. it points to some garbage memory address. Be careful with them.
- Null pointer: A standard pointer to the `NULL` value/location.

Needs. `#include <stddef.h>`

- Dangling pointer: A standard pointer which points to a memory location that was *deallocated* (more in later lectures).

  Be careful with them.

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location
- Wild pointer: An *uninitialized* pointer, i.e. it points to some garbage memory address. Be careful with them.
- Null pointer: A standard pointer to the `NULL` value/location.

Needs. `#include <stddef.h>`

- Dangling pointer: A standard pointer which points to a memory location that was *deallocated* (more in later lectures).

  Be careful with them.
- Void pointer: A pointer that has no data type associated with it. Why?

Simple pointers

- Standard pointer: A pointer with a well defined *C type* and referring to a well defined memory location
- Wild pointer: An *uninitialized* pointer, i.e. it points to some garbage memory address. Be careful with them.
- Null pointer: A standard pointer to the `NULL` value/location.

Needs. `#include <stddef.h>`

- Dangling pointer: A standard pointer which points to a memory location that was *deallocated* (more in later lectures).

Be careful with them.

- Void pointer: A pointer that has no data type associated with it. Why?

Usefulness well become clear in later lectures.

Asides: … wait so *complex* pointers exist?

Asides: … wait so *complex* pointers exist?

- Yes. Example:

Asides: … wait so *complex* pointers exist?

- Yes. Example:

```
char* (*(*foo[5])(char*))[]
```

Asides: … wait so *complex* pointers exist?

- Yes. Example:

```
char* (*(*foo[5])(char*))[]
```

- But we don’t know enough C for them to be useful to us …

Asides: … wait so *complex* pointers exist?

- Yes. Example:

```
char* (*(*foo[5])(char*))[]
```

- But we don’t know enough C for them to be useful to us …

- … yet. More importantly, ….

Asides: … wait so *complex* pointers exist?

- Yes. Example:

```
char* (*(*foo[5])(char*))[]
```

- But we don’t know enough C for them to be useful to us …
- … yet. More importantly, ….
- I have failed at my job if you start coding like this!

Asides: … wait so *complex* pointers exist?

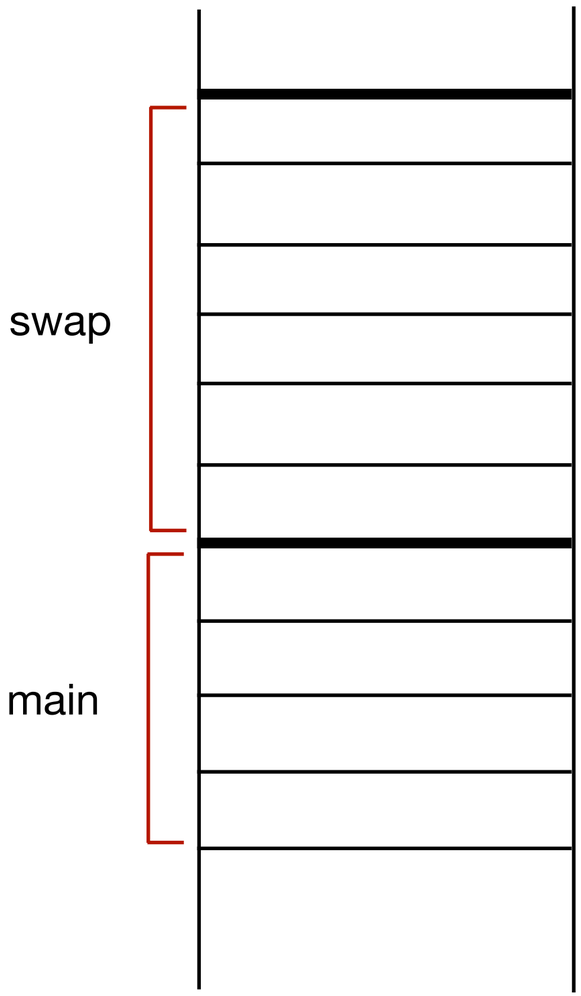
- Yes. Example:

```
char* (*(*foo[5])(char*))[]
```

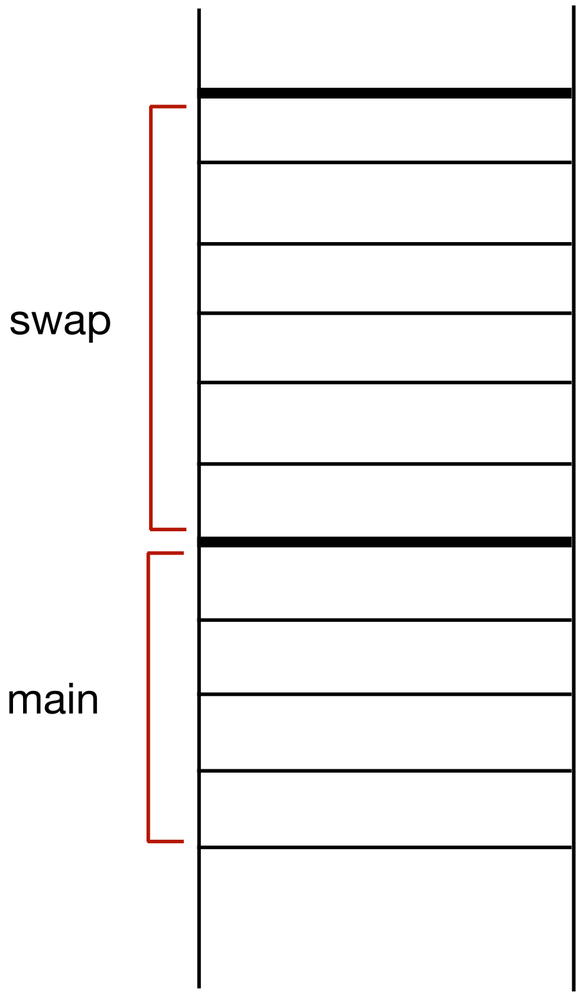
- But we don’t know enough C for them to be useful to us …
- … yet. More importantly, ….
- I have failed at my job if you start coding like this!

Curious? You can play with: https://cdecl.org/

Using pointers in C

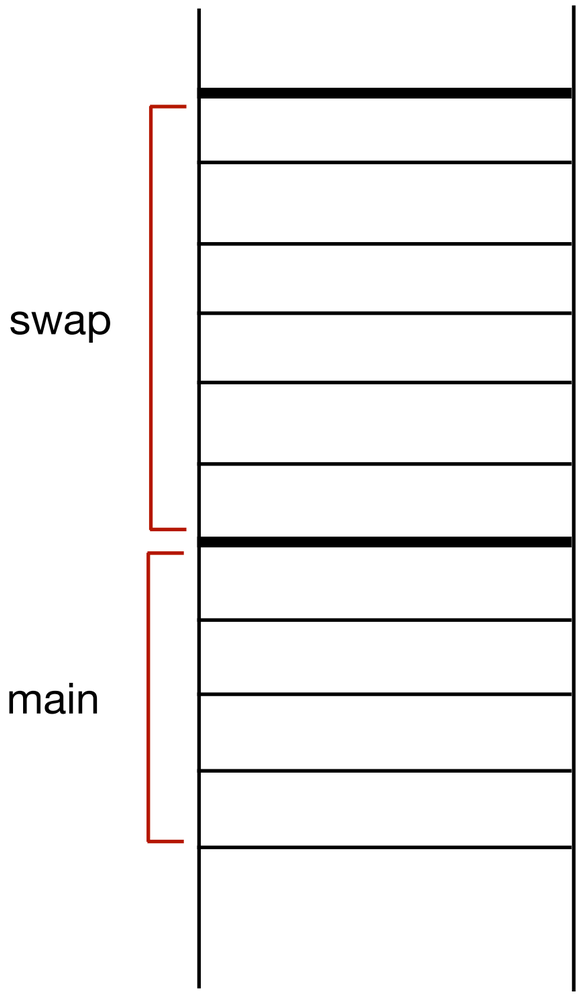


Using pointers in C



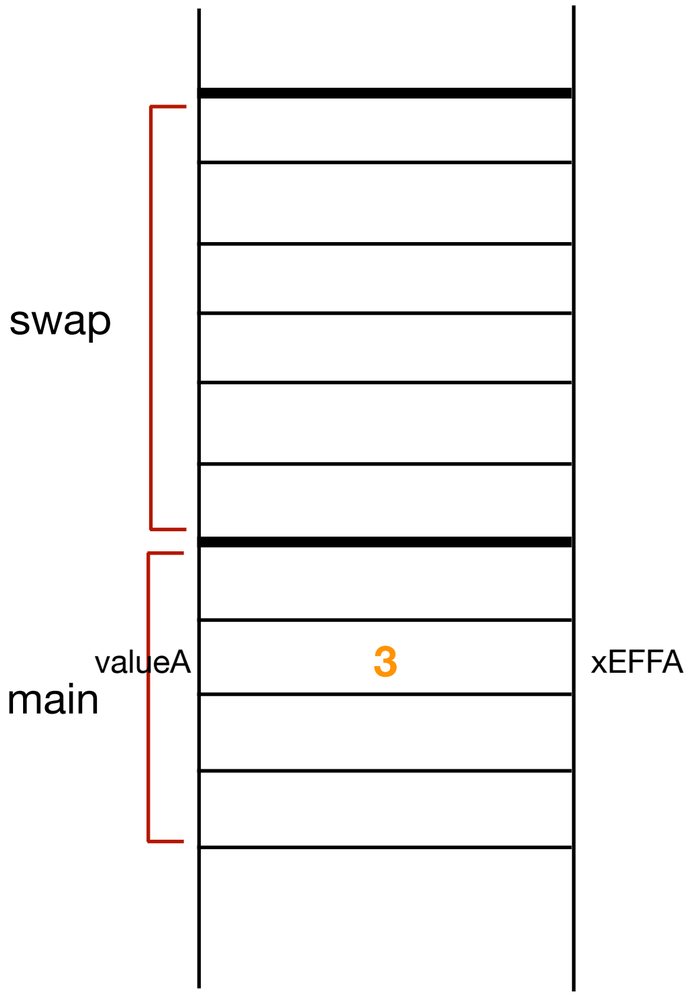

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



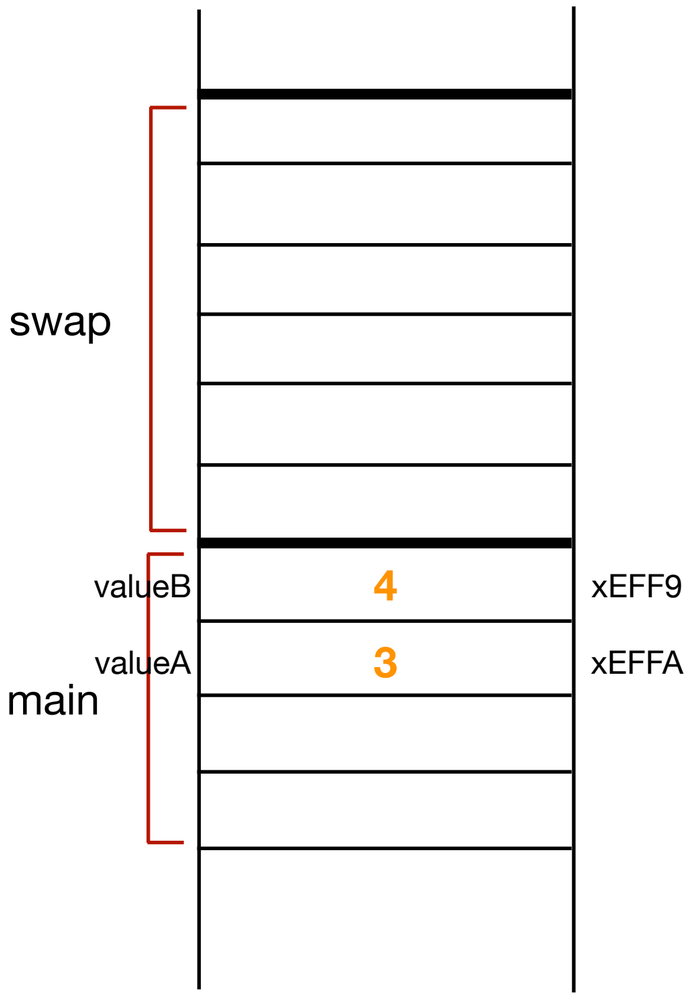

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



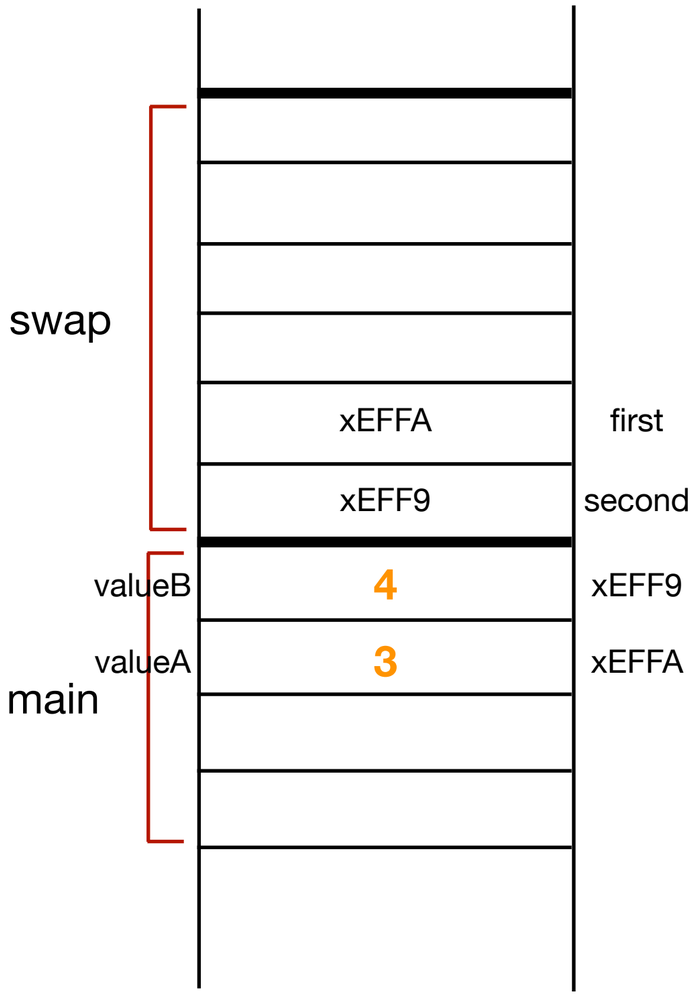

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



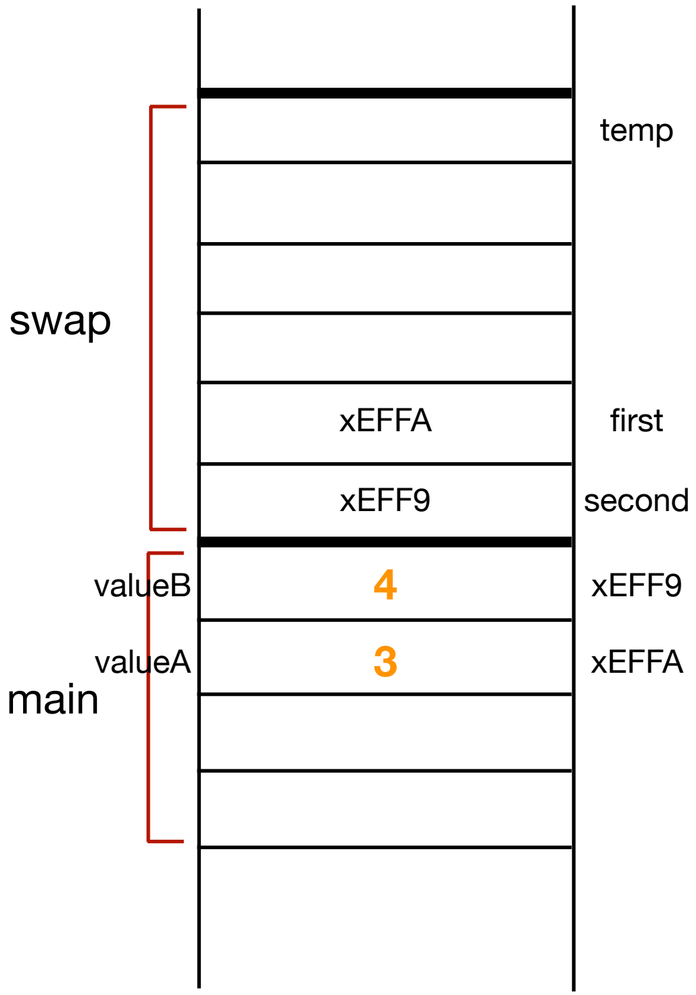

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



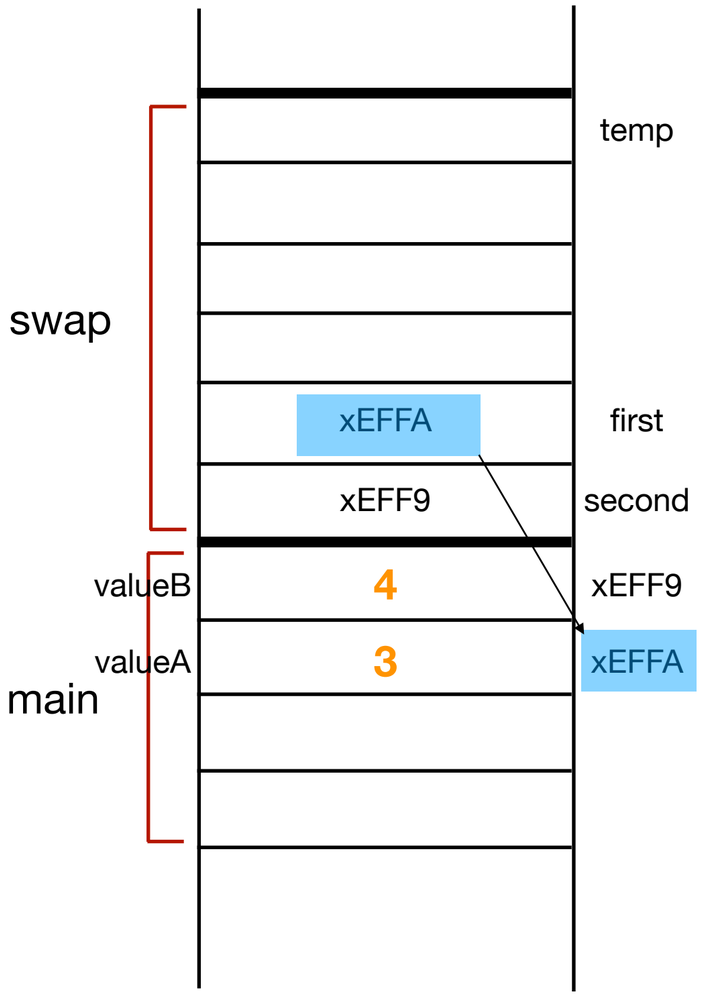

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(&valueA, &valueB);
}
```

Using pointers in C



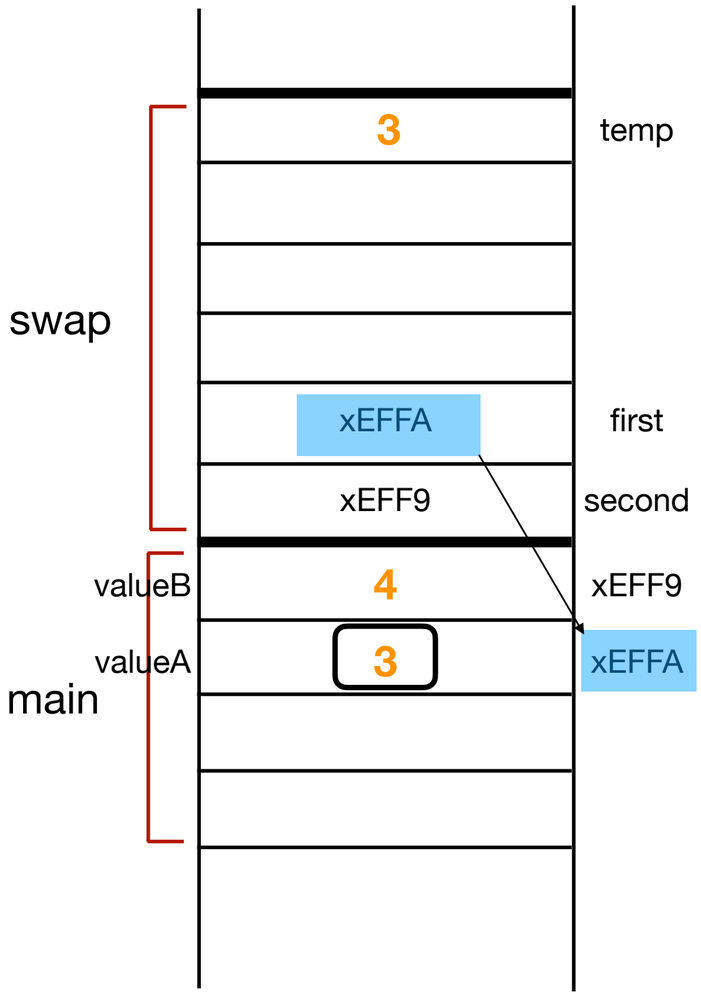

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



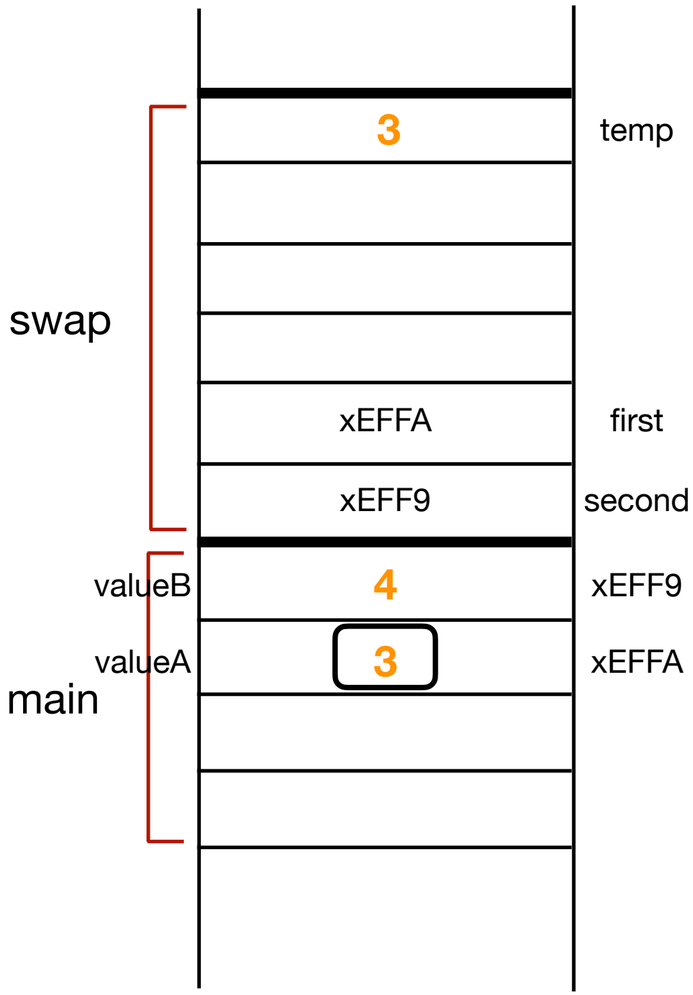

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



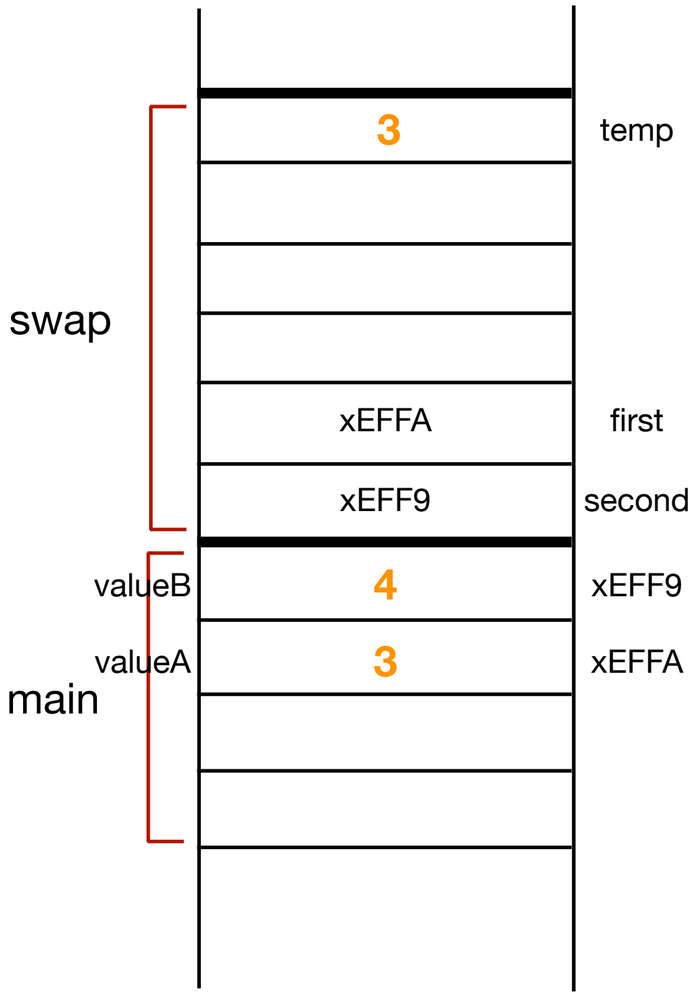

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



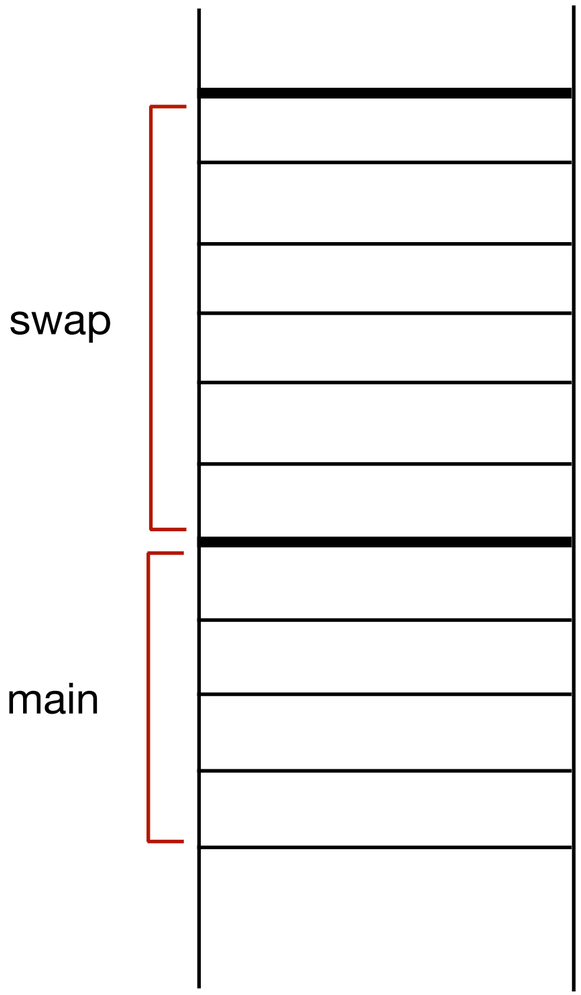

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



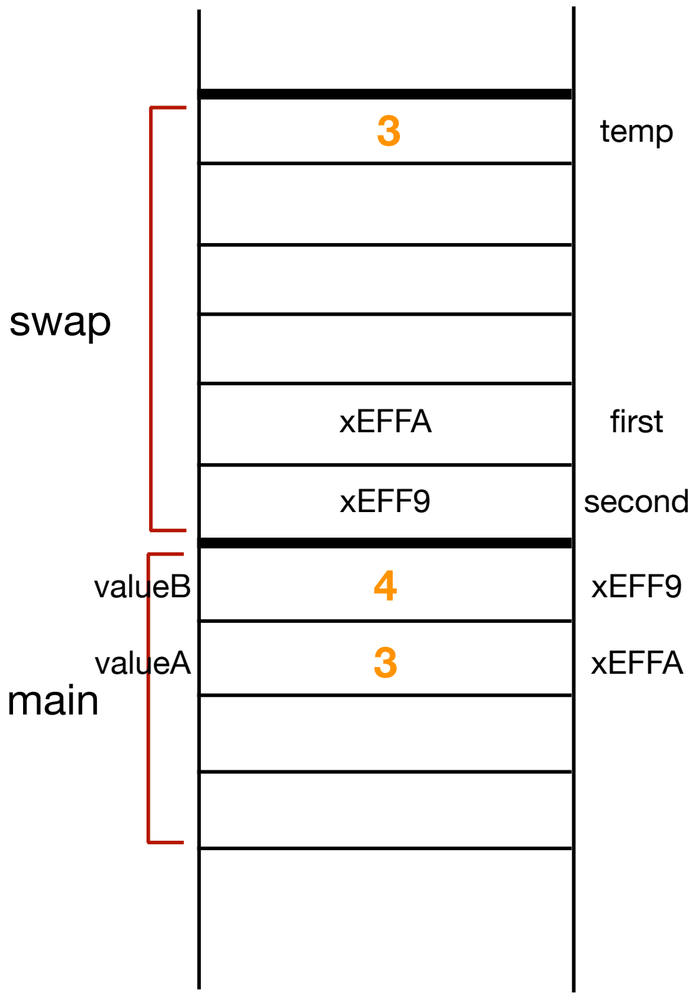

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



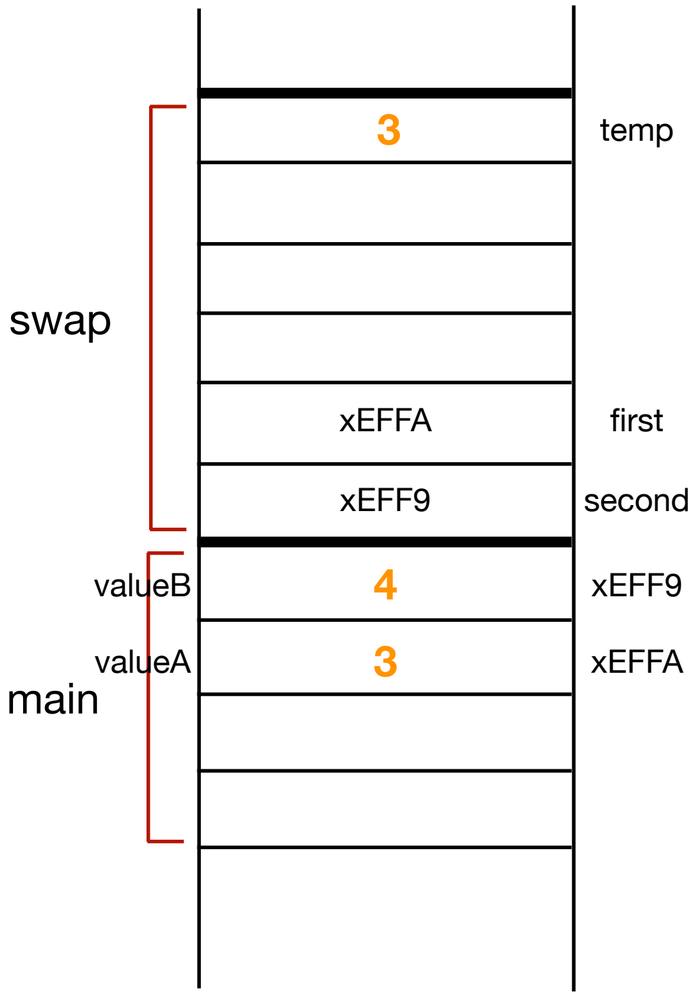
Question: What are the missing items on the run time stack?

Using pointers in C



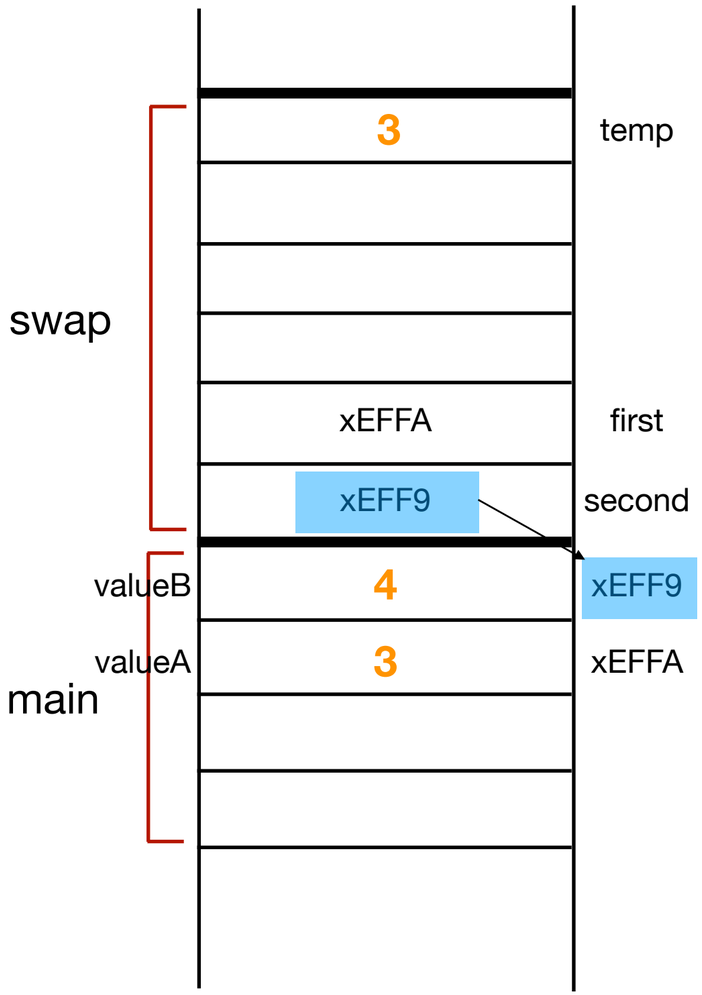
Question: What are the missing items on the run time stack?

Using pointers in C



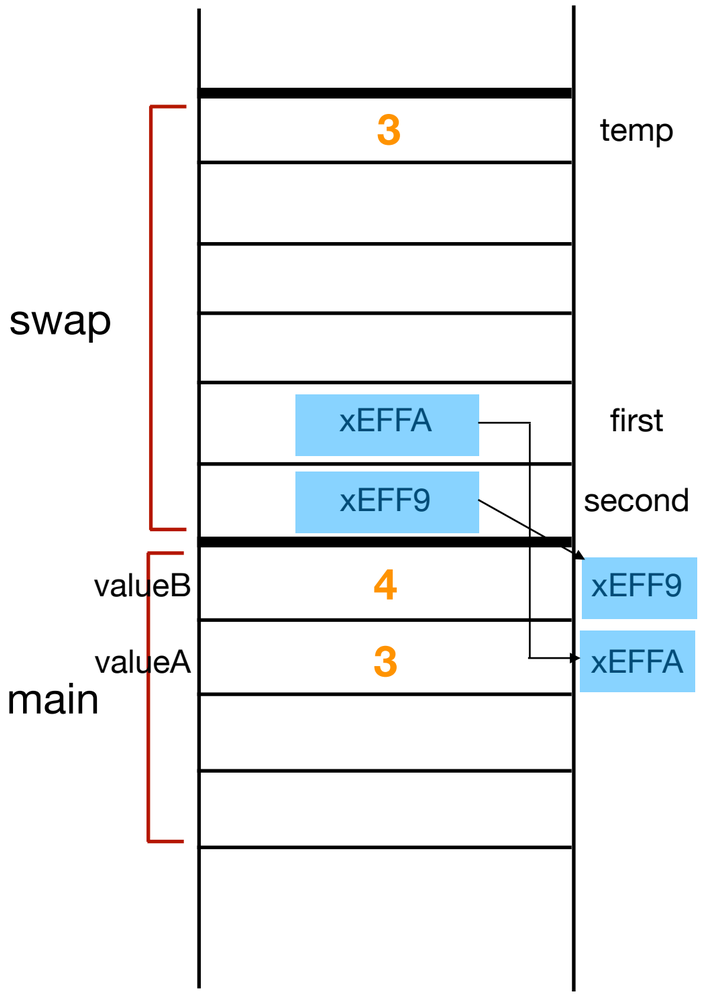

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



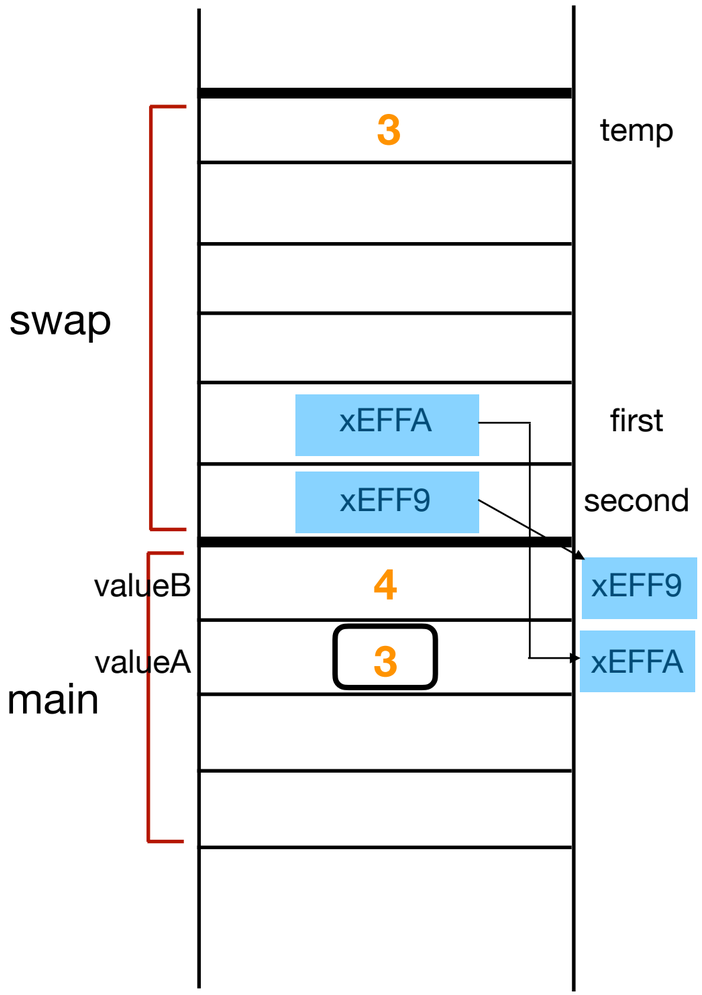

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



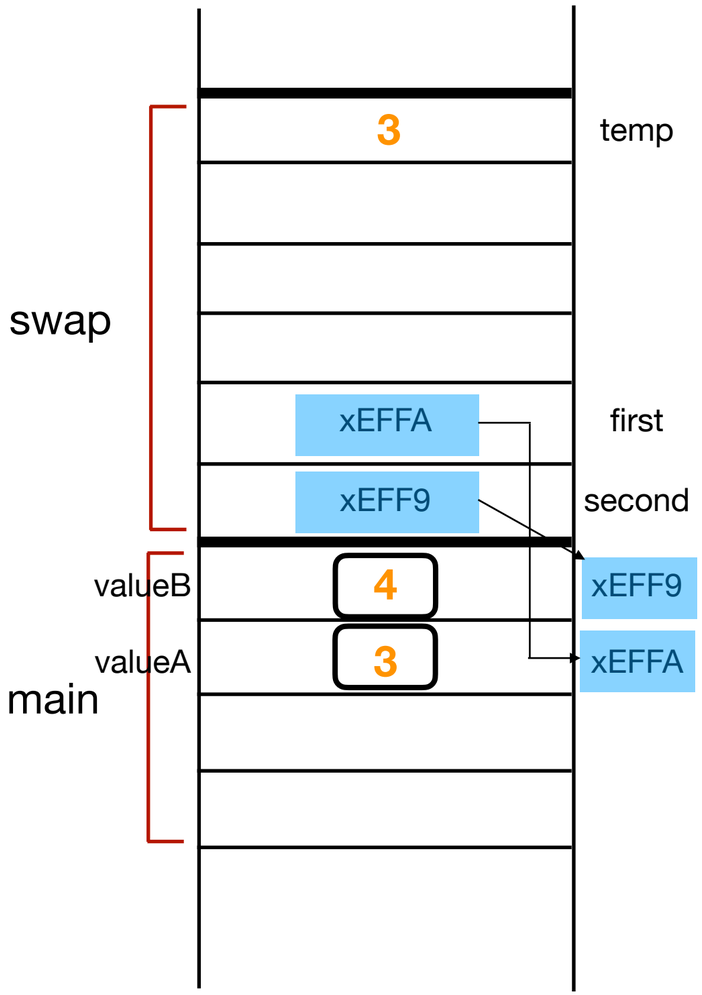

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



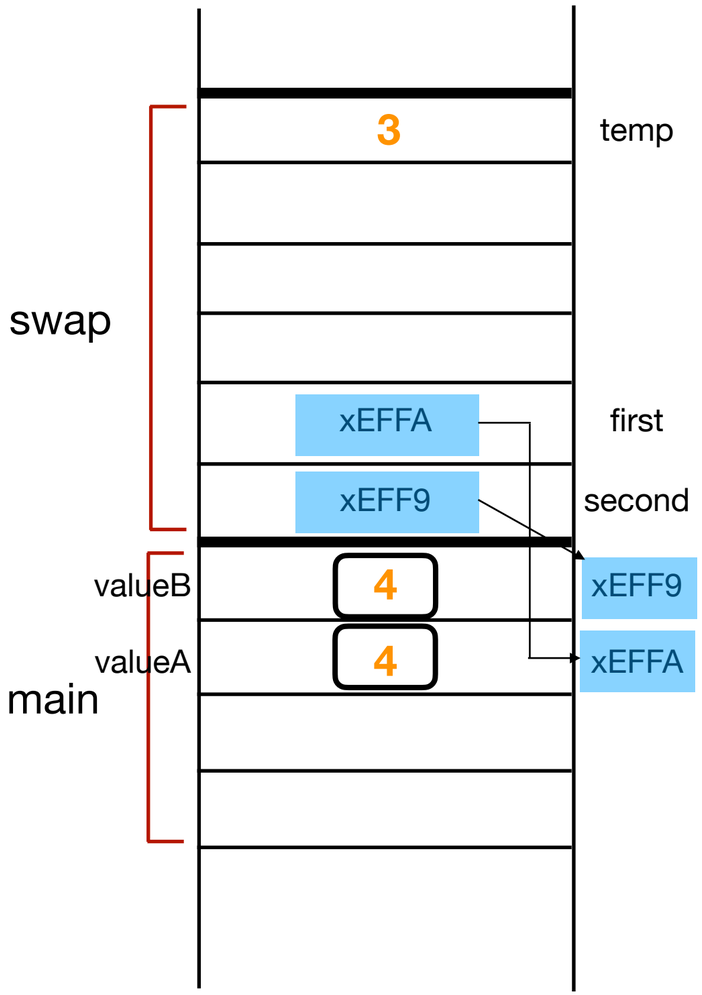

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



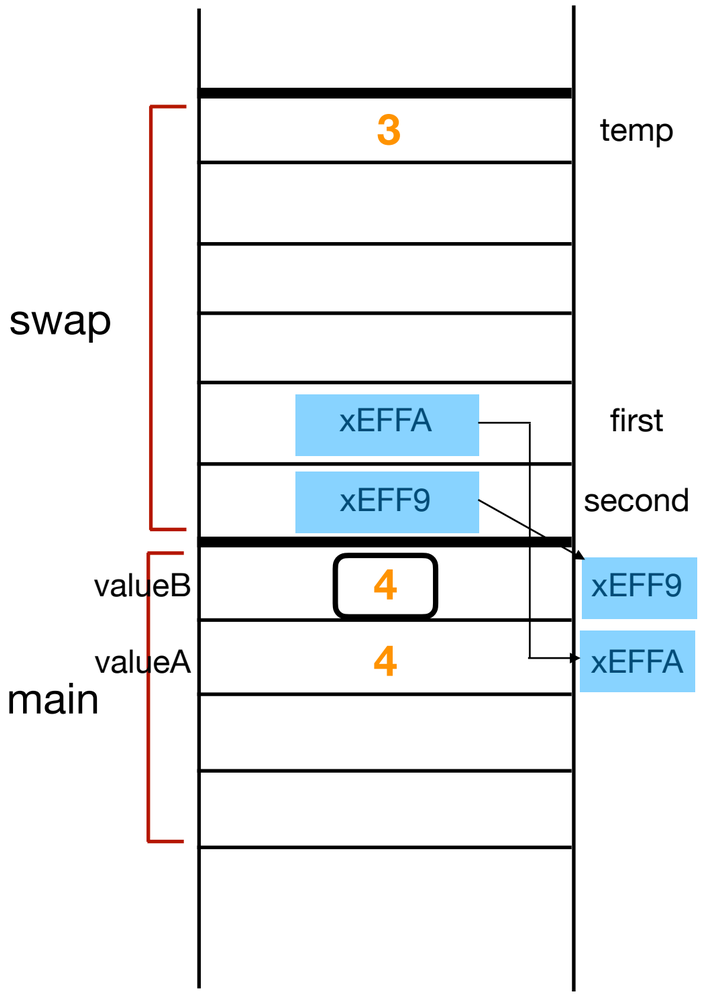

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



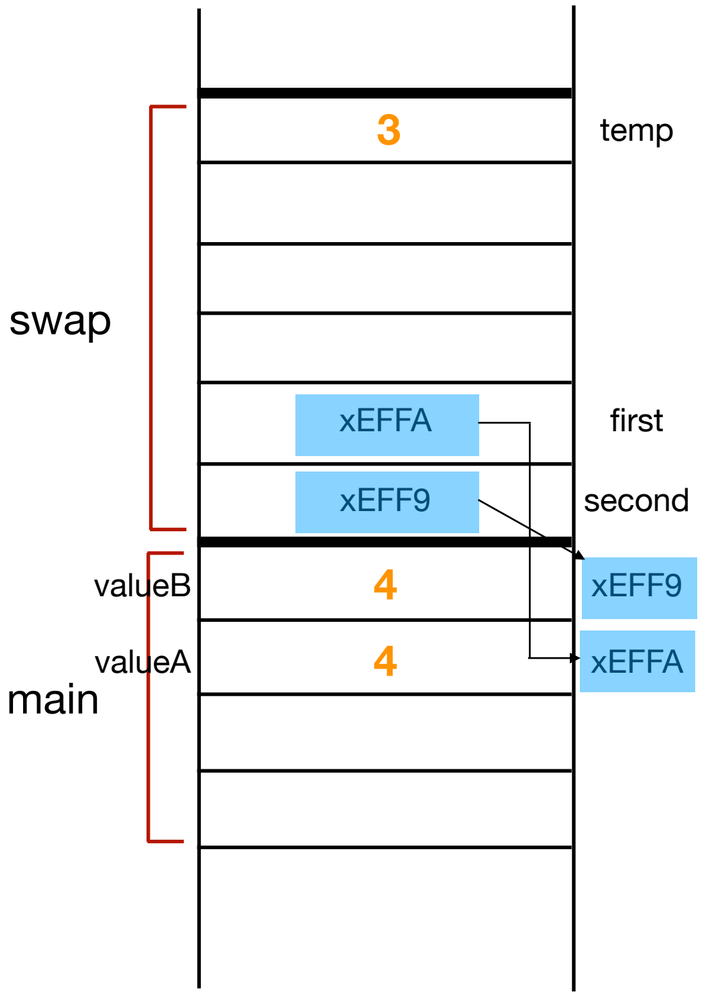

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



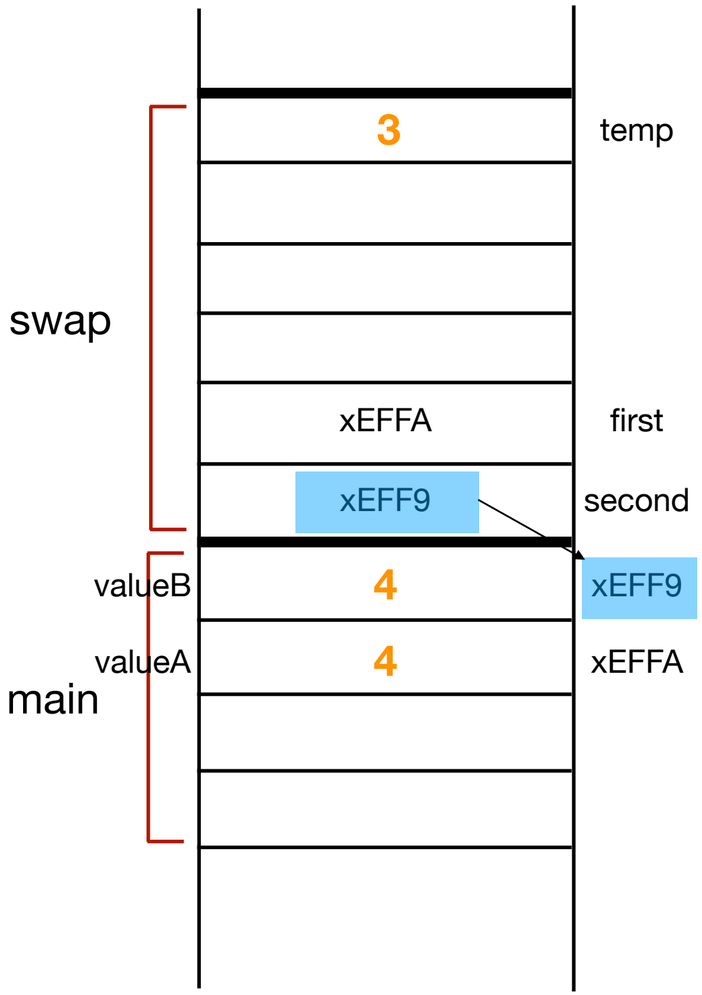

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



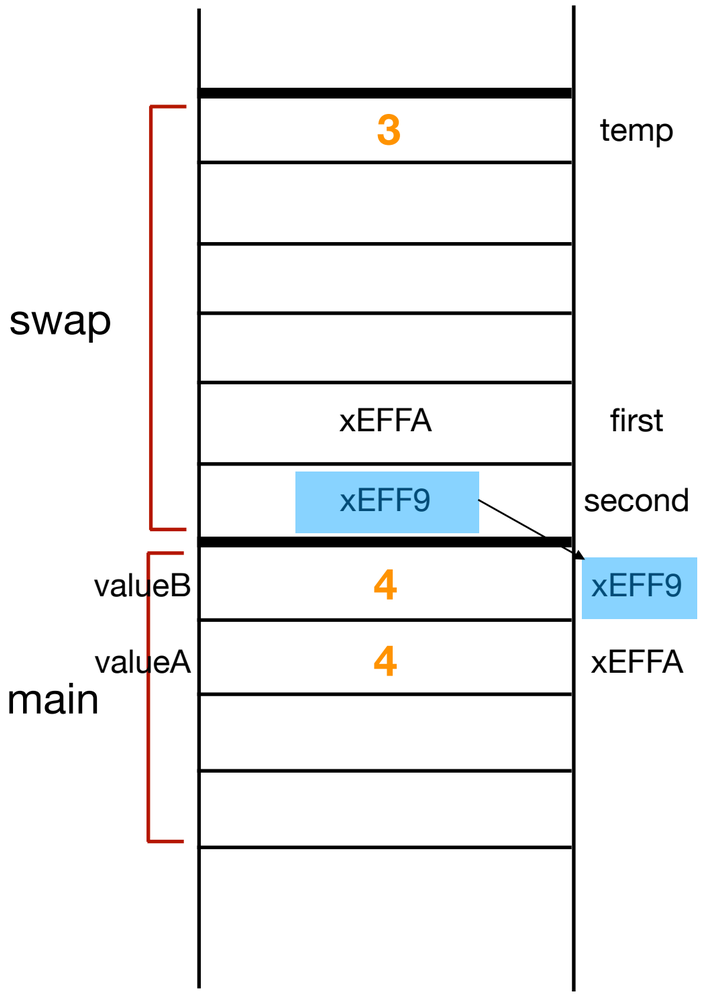

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



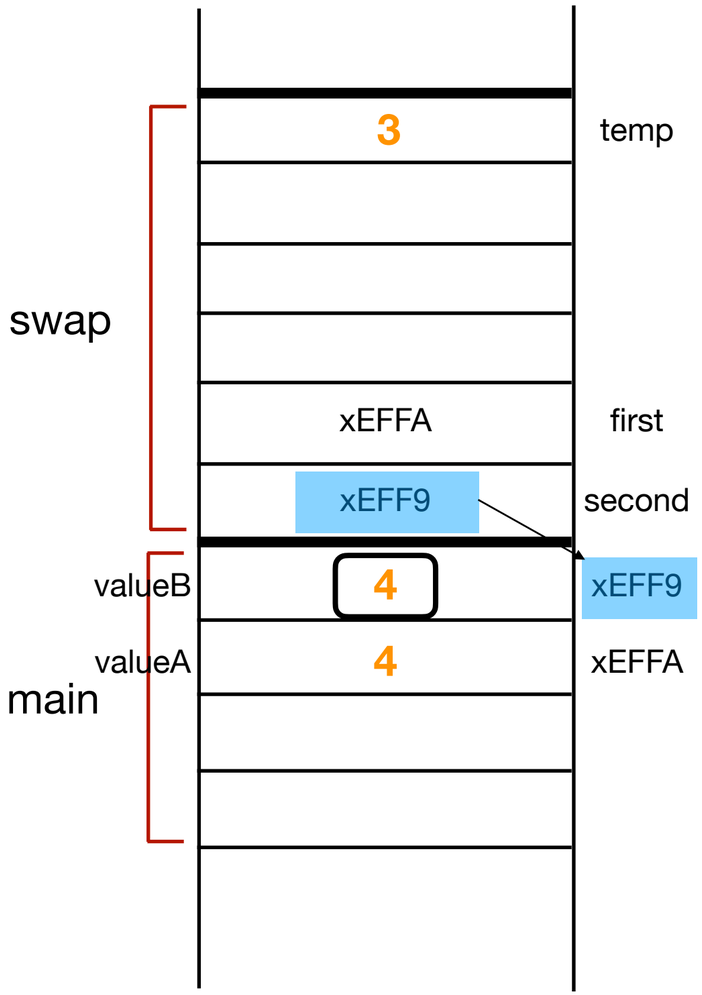

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



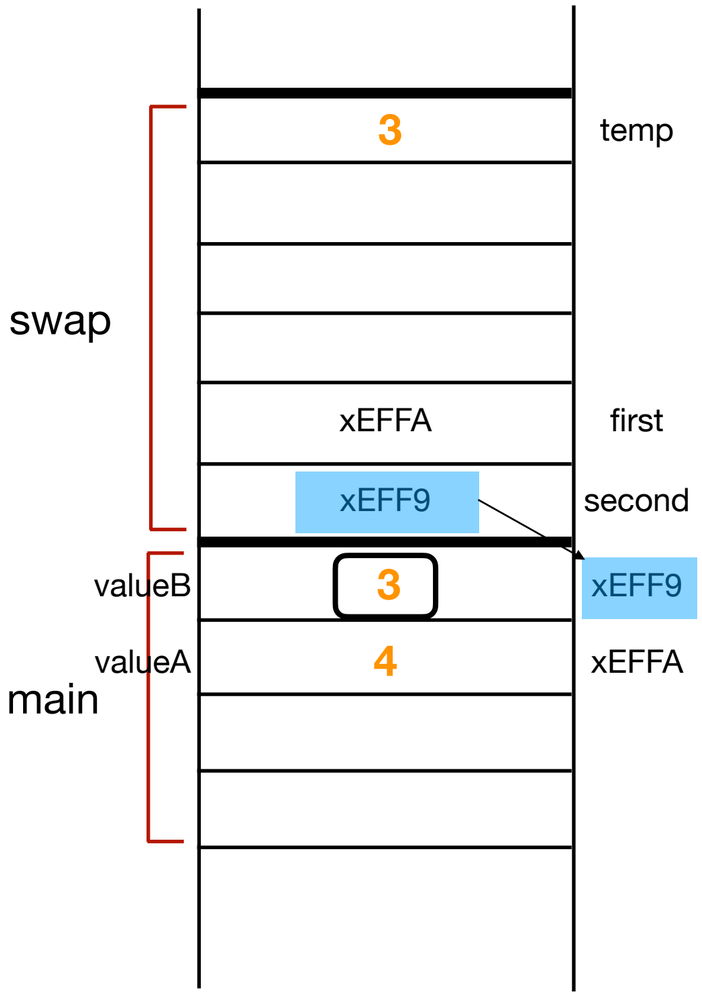

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(&valueA, &valueB);
}
```

Using pointers in C



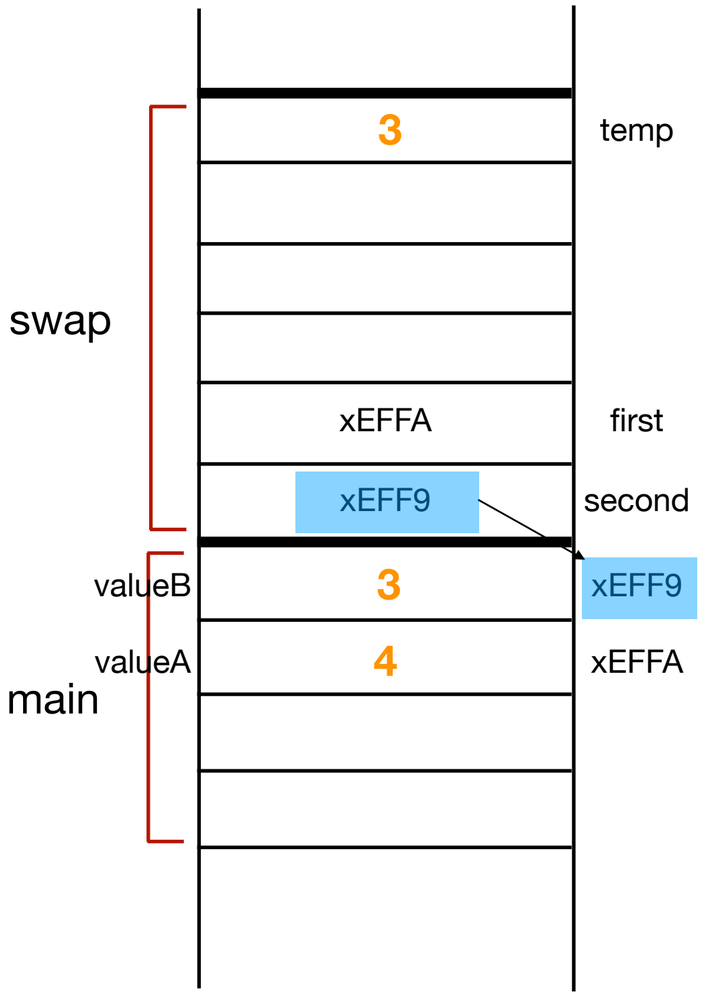

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(&valueA, &valueB);
}
```

Using pointers in C



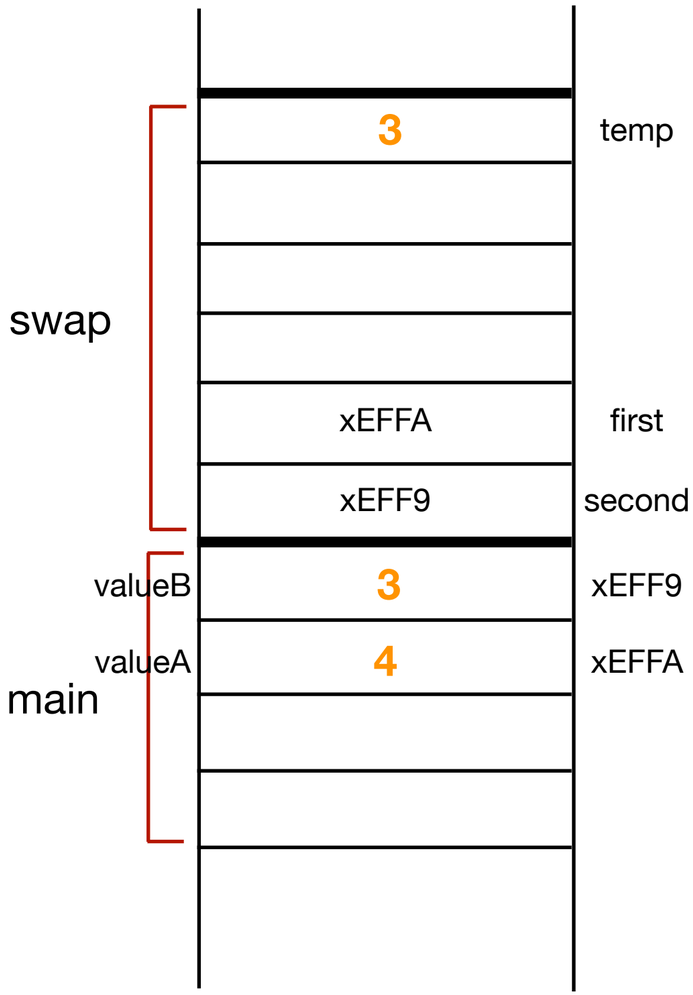

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(&valueA, &valueB);
}
```

Using pointers in C



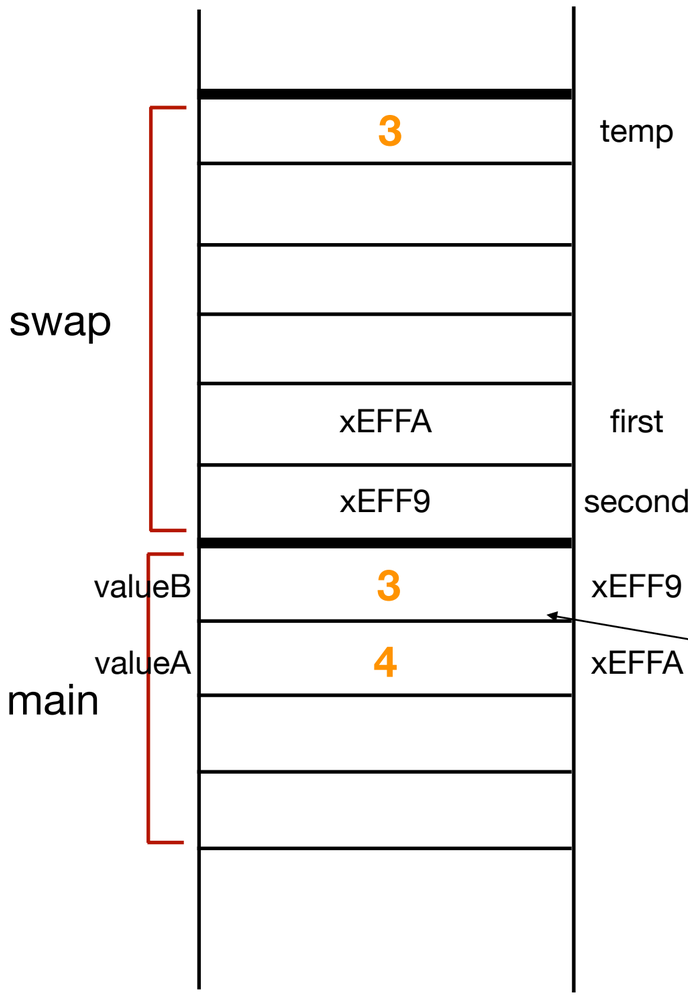

```
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(&valueA, &valueB);
}
```

Using pointers in C




```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C



Compare

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
   int valueA = 3;
   int valueB = 4;
   Swap(&valueA, &valueB);
}
```

Using pointers in C

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C

- Pointers *need* to be indicated when making parameter declarations.

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C

- Pointers *need* to be indicated when making parameter declarations.

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C

- Pointers *need* to be indicated when making parameter declarations.
- How did we use the value at memory location which pointer is pointing to?

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C

- Pointers *need* to be indicated when making parameter declarations.
- How did we use the value at memory location which pointer is pointing to?

`*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```c
#include <stdio.h>

void Swap(int *first, int *second){
int temp;
temp = *first;
*first = *second;
*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Using pointers in C

- Which uses of * are *dereferencing* (not declarations) ?

```
#include <stdio.h>

                    1              2
void Swap(int *first, int *second){
int temp;
temp = *first;3
4*first = *second;5
6*second = temp;
}

int main(){
  int valueA = 3;
  int valueB = 4;
  Swap(&valueA, &valueB);
}
```

Asides: … pointers *only* point to variables?

Asides: … pointers *only* point to variables?

- No.

Asides: … pointers *only* point to variables?

- No.
- They can point to functions,
  structs, *other pointers*, etc.

Asides: … pointers *only* point to variables?

- No.
- They can point to functions, structs, *other pointers*, etc.
- Example on left shows a pointer to a function.

Asides: … pointers *only* point to variables?

- No.
- They can point to functions, structs, *other pointers*, etc.
- Example on left shows a pointer to a function.

```c
#include <stdio.h>

void fun(int a){
    printf("Value of a is %d\n", a);
}

int main(void){
    void (*fun_ptr)(int) = &fun;
    (*fun_ptr)(10);

    return 0;
}
```

Asides: … pointers *only* point to variables?

- No.
- They can point to functions, structs, *other pointers*, etc.
- Example on left shows a pointer to a function.
- We will learn about them on a **need-to-know** basis.

```c
#include <stdio.h>

void fun(int a){
    printf("Value of a is %d\n", a);
}

int main(void){
    void (*fun_ptr)(int) = &fun;
    (*fun_ptr)(10);

    return 0;
}
```

Using pointers in C

Using pointers in C

Usage summary:

Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
```

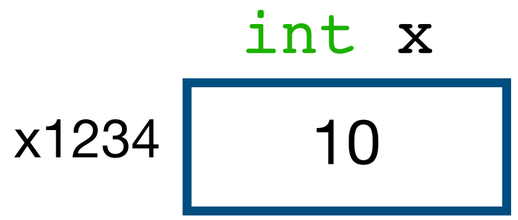
Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
```



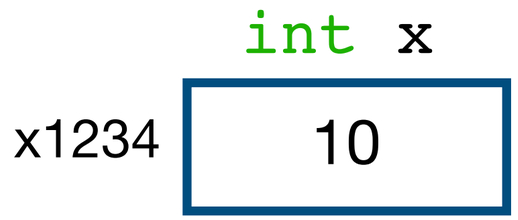
Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
int *p;
```



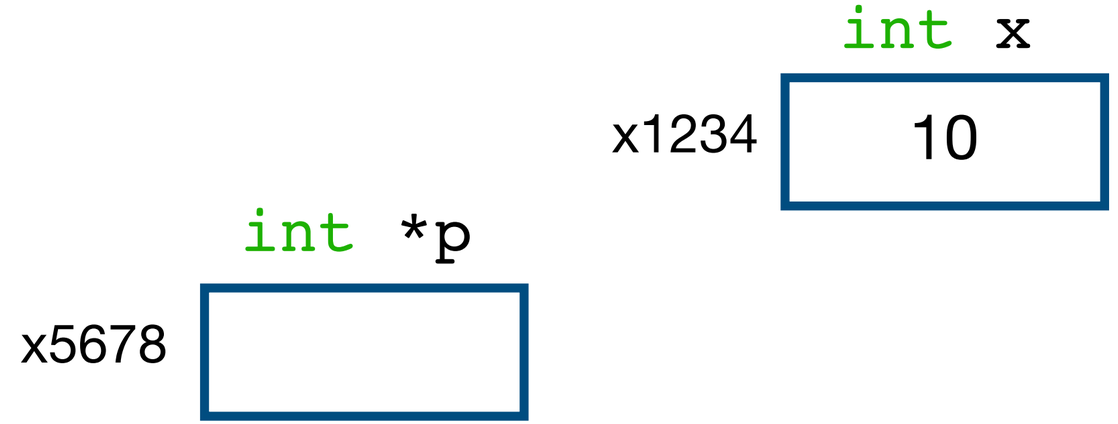
Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
int *p;
```



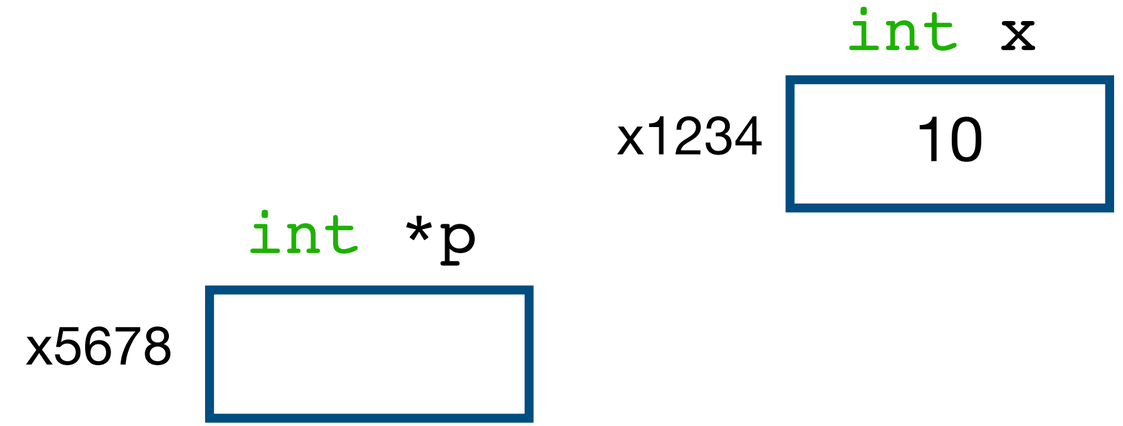
Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
int *p;
p = &x;
```



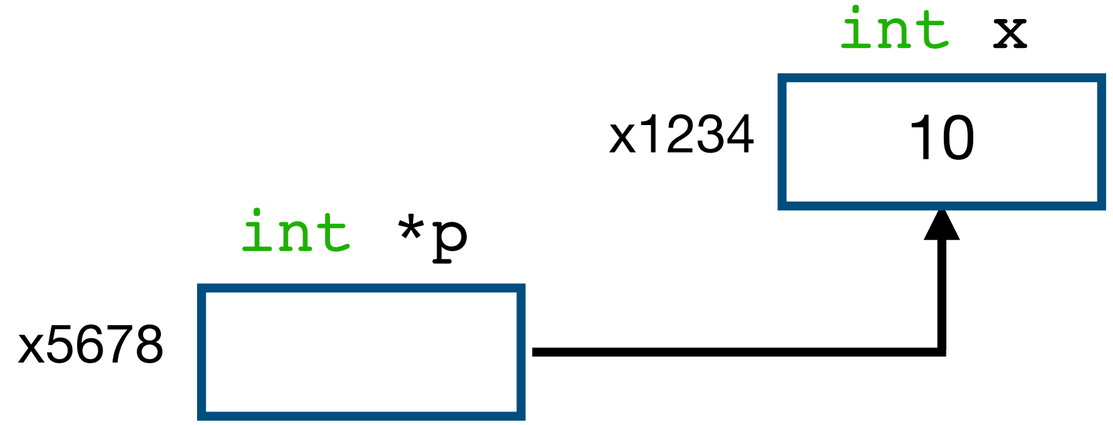
Using pointers in C

Usage summary:

`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
int *p;
p = &x;
```



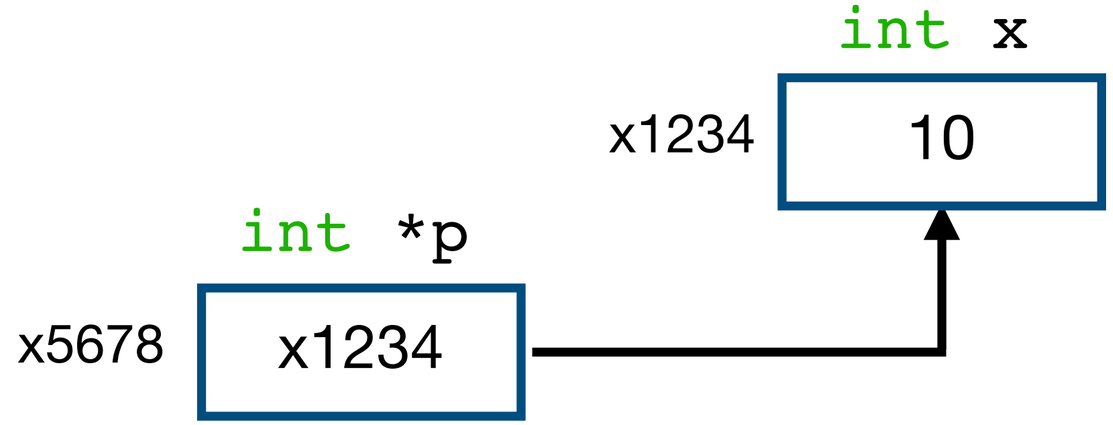
Using pointers in C

Usage summary:

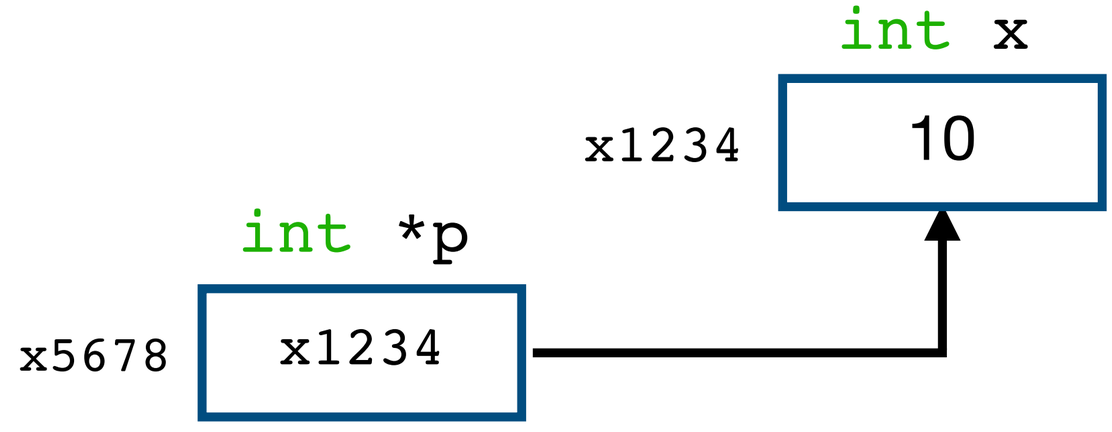
`&` → `&val` → ***address operator***: returns address of variable `val`

`*` → `*ptr` → ***dereference operator***: returns the value pointed to by `ptr`

```
int x = 10;
int *p;
p = &x;
```

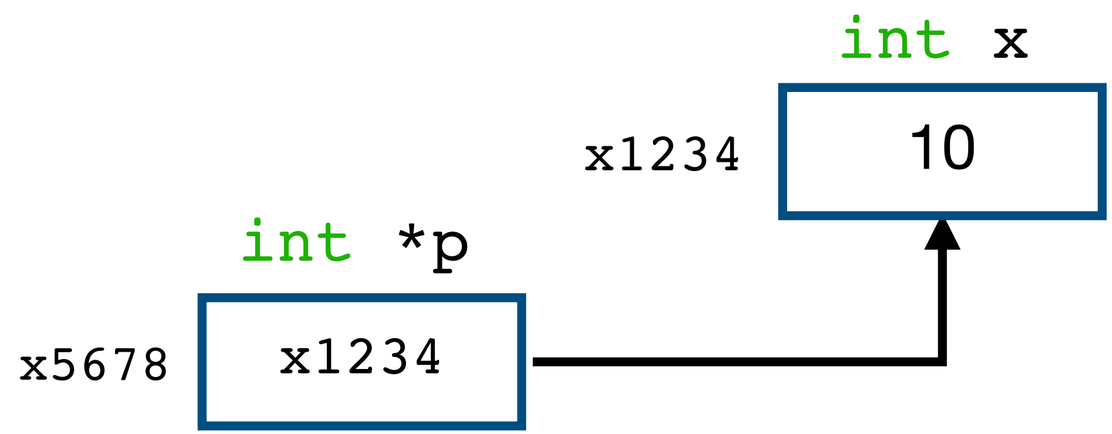


More pointers in C



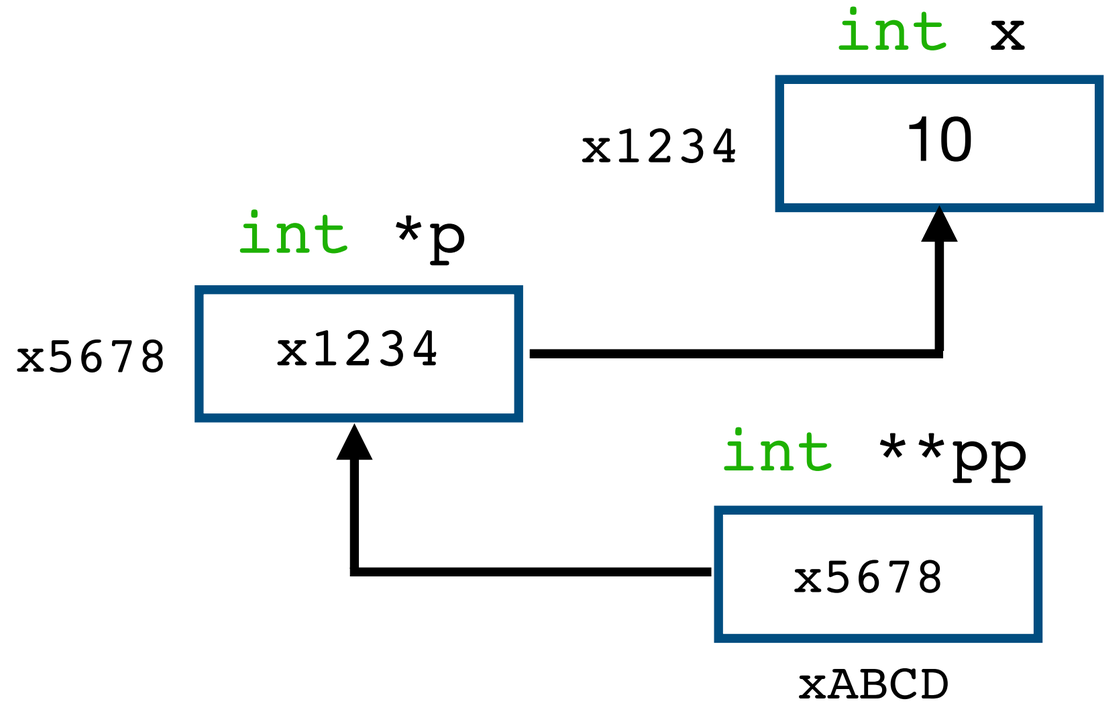
More pointers in C

```
int x = 10;
int *p = &x;

/* Guess the outputs 1*/
```




```
printf("x%X\n", &x);
printf("x%X\n", p);
printf("x%X\n", &p);
printf("%d\n", *p);

*p = *p + 10;

printf("%d\n", *p);
printf("%d\n", x);
```

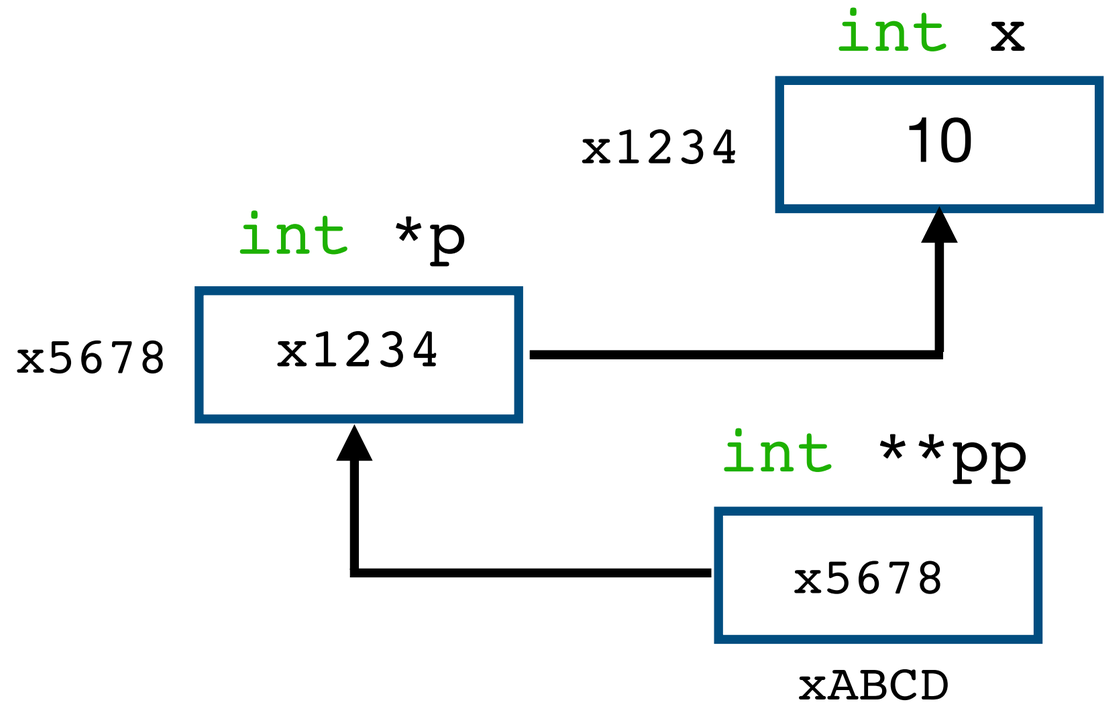
Even more pointers in C



Even more pointers in C

```
int x = 10;
int *p = &x;
int **pp = &p;
```

```
/* Guess the outputs 2 */

printf("x%X\n", &pp);
printf("x%X\n", pp);
printf("x%X\n", *pp);
printf("%d\n", **pp);
```



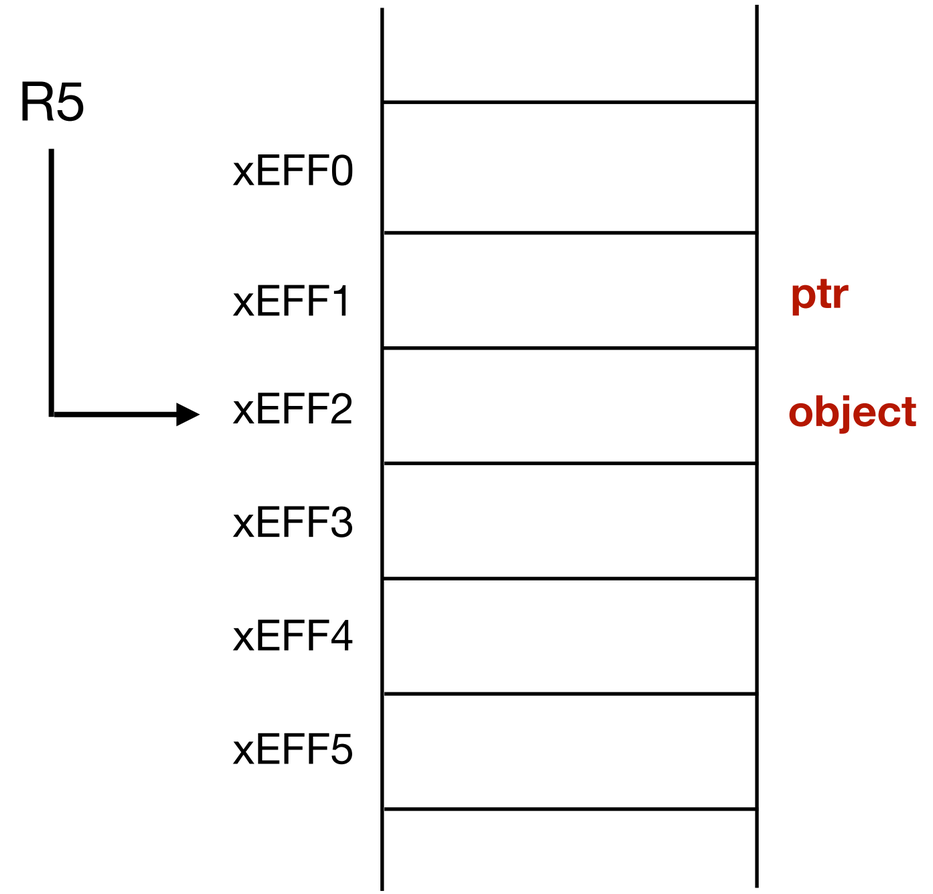
Pointers in LC-3

Pointers in LC-3

```
int object;
int *ptr;
```

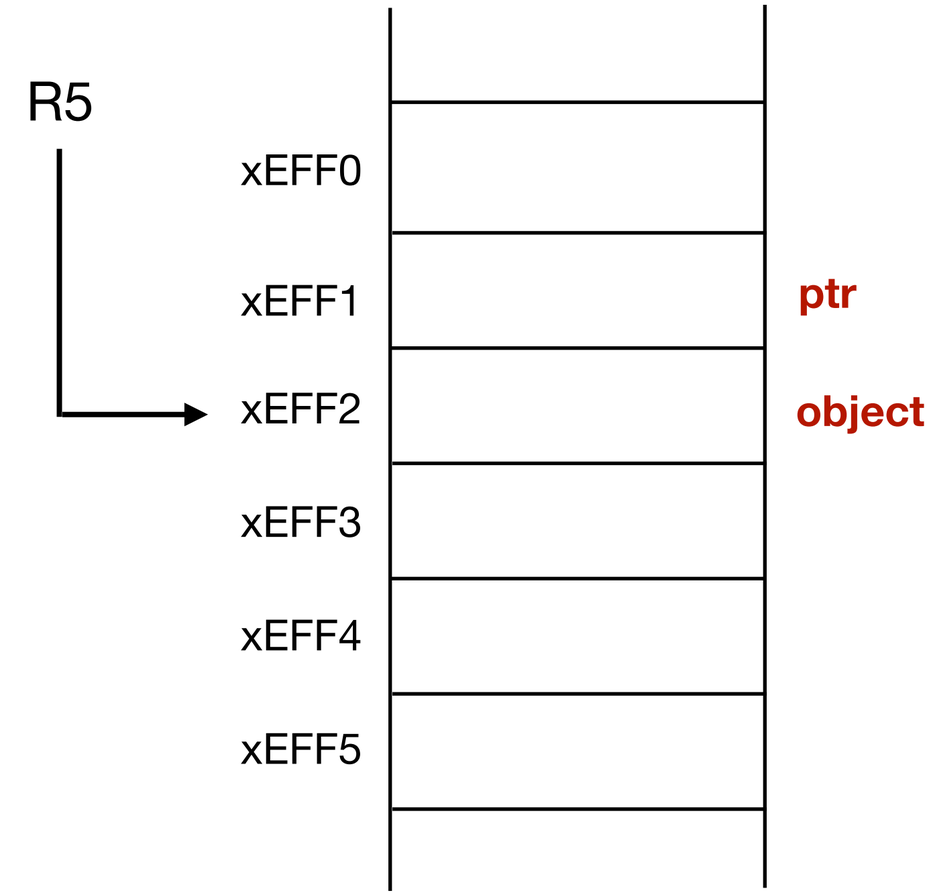
Pointers in LC-3

```
int object;
int *ptr;
```



Pointers in LC-3

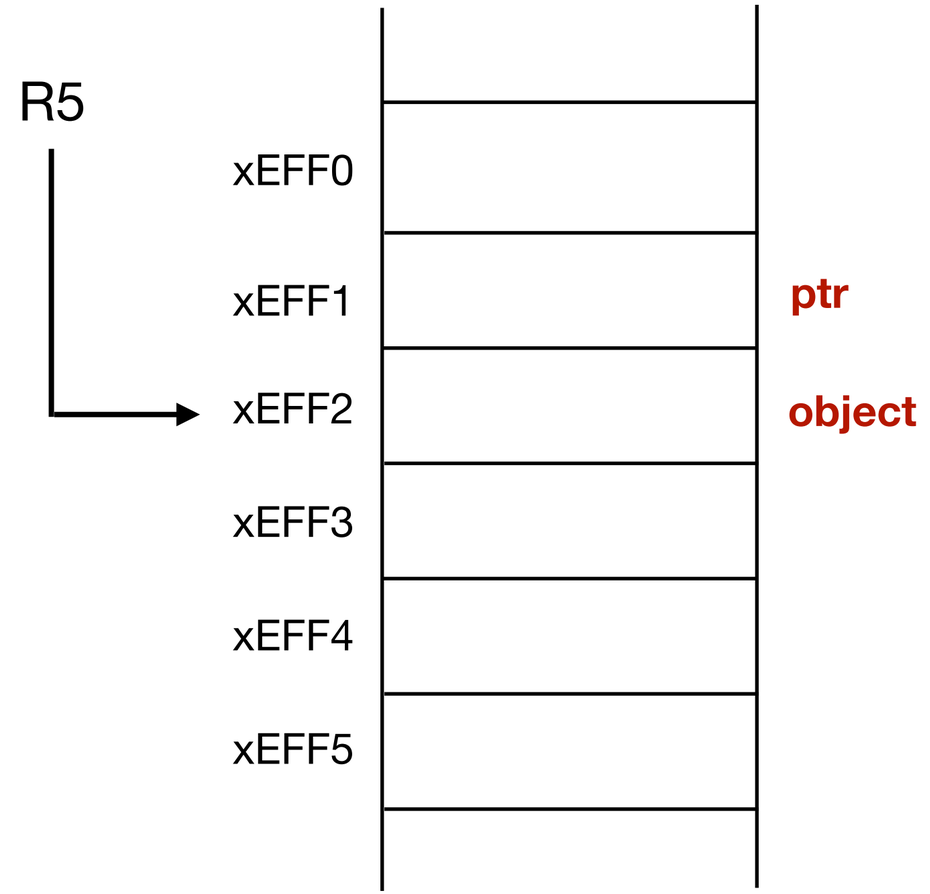
```
int object;
int *ptr;

object = 4;
ptr = &object;
```



Pointers in LC-3

```
int object;
int *ptr;

object = 4;
ptr = &object;
```

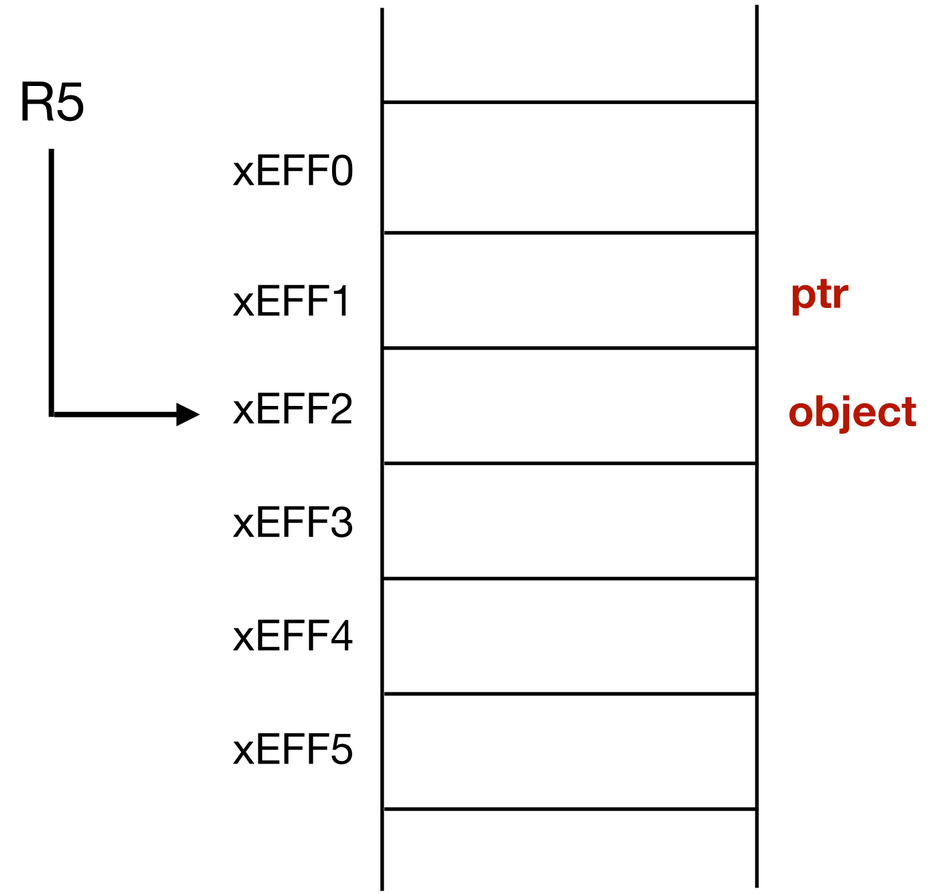
```
AND R0, R0, #0    ; Clear R0
```



Pointers in LC-3

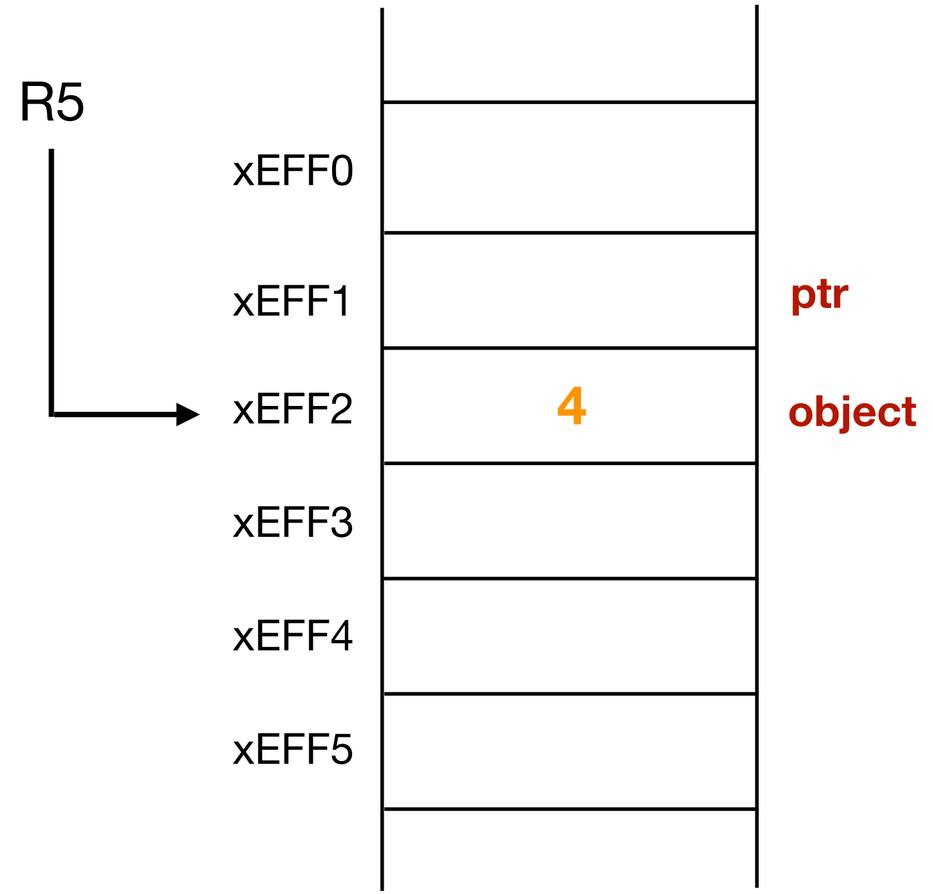
```
int object;
int *ptr;

object = 4;
ptr = &object;
```

```
AND R0, R0, #0    ; Clear R0
ADD R0, R0, #4    ; R0 = 4
```



Pointers in LC-3

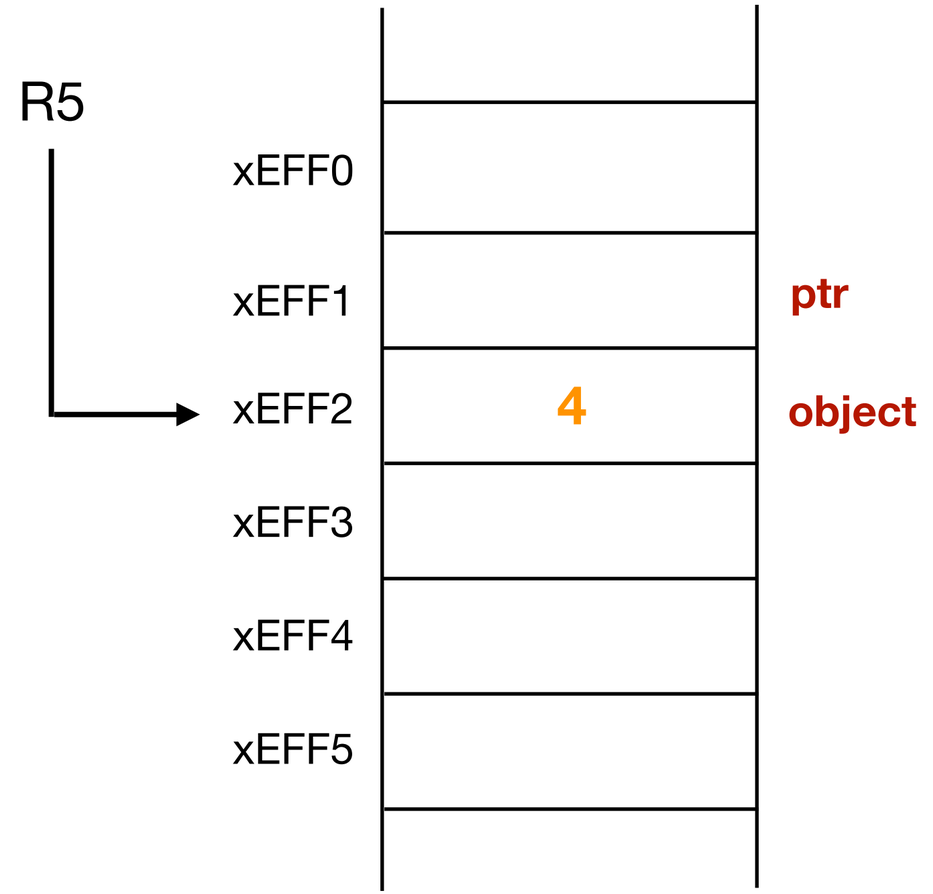
```
int object;
int *ptr;

object = 4;
ptr = &object;
```




```
AND R0, R0, #0    ; Clear R0
ADD R0, R0, #4    ; R0 = 4
STR R0, R5, #0    ; object = 4
```

Pointers in LC-3

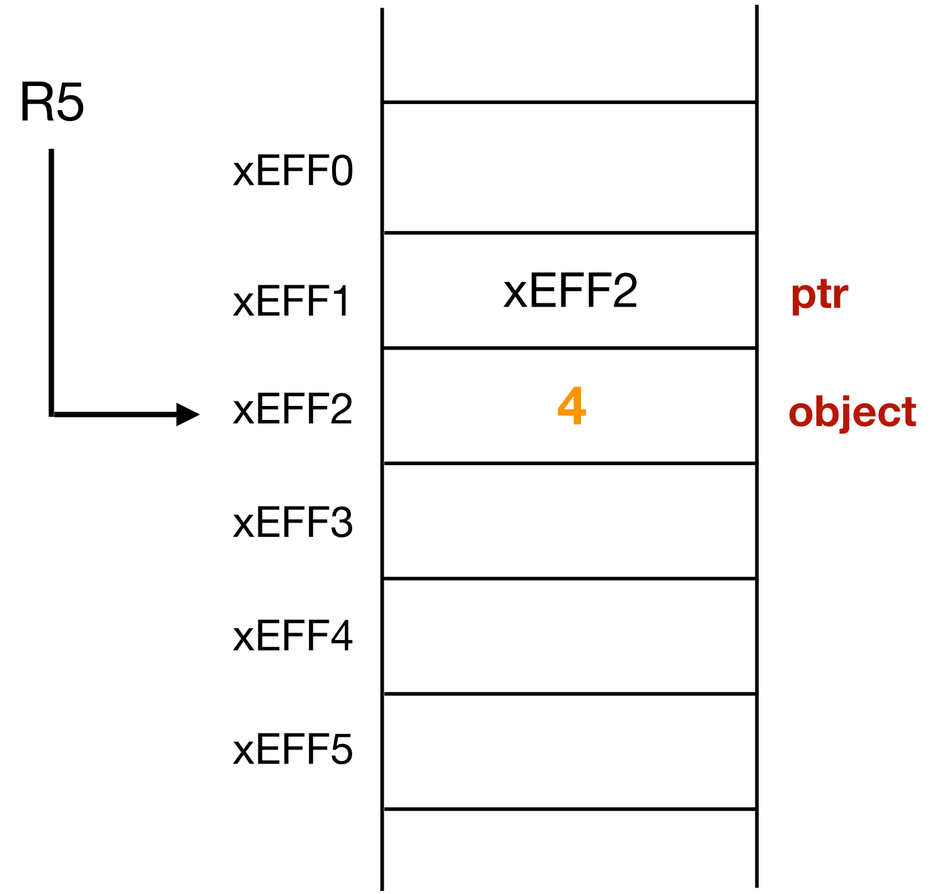
```
int object;
int *ptr;

object = 4;
ptr = &object;
```




```
AND R0, R0, #0    ; Clear R0
ADD R0, R0, #4    ; R0 = 4
STR R0, R5, #0    ; object = 4

ADD R0, R5, #0    ; Generate memory address of object
```

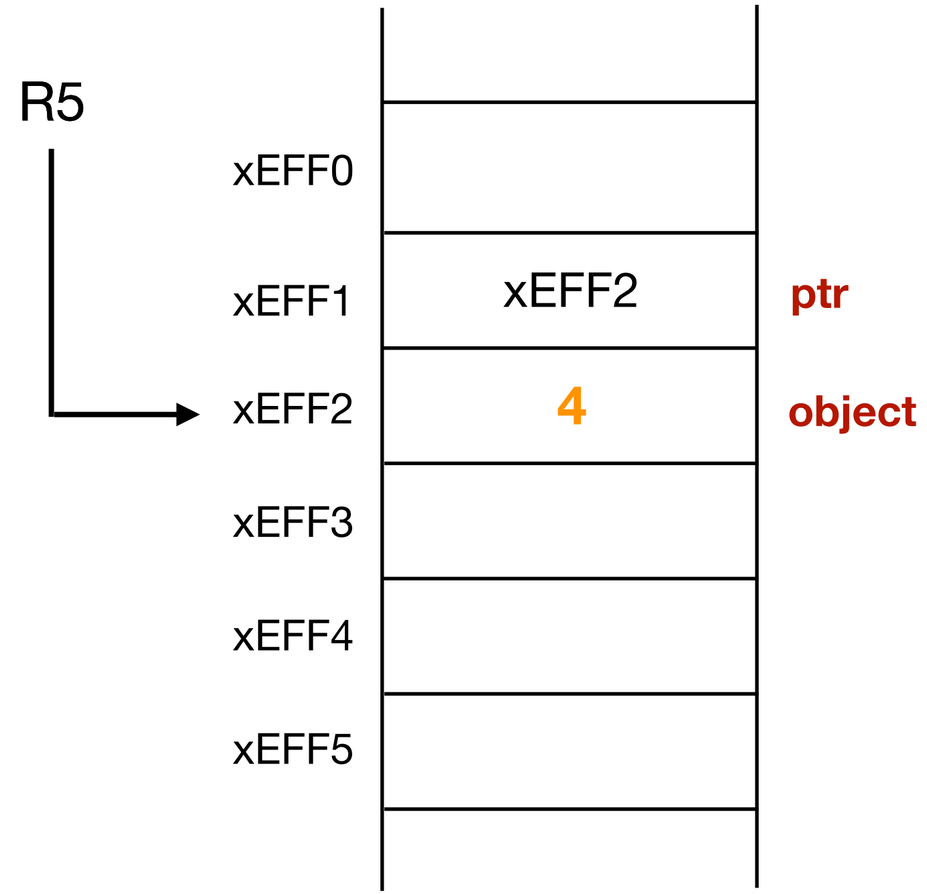
Pointers in LC-3

```
int object;
int *ptr;

object = 4;
ptr = &object;
```



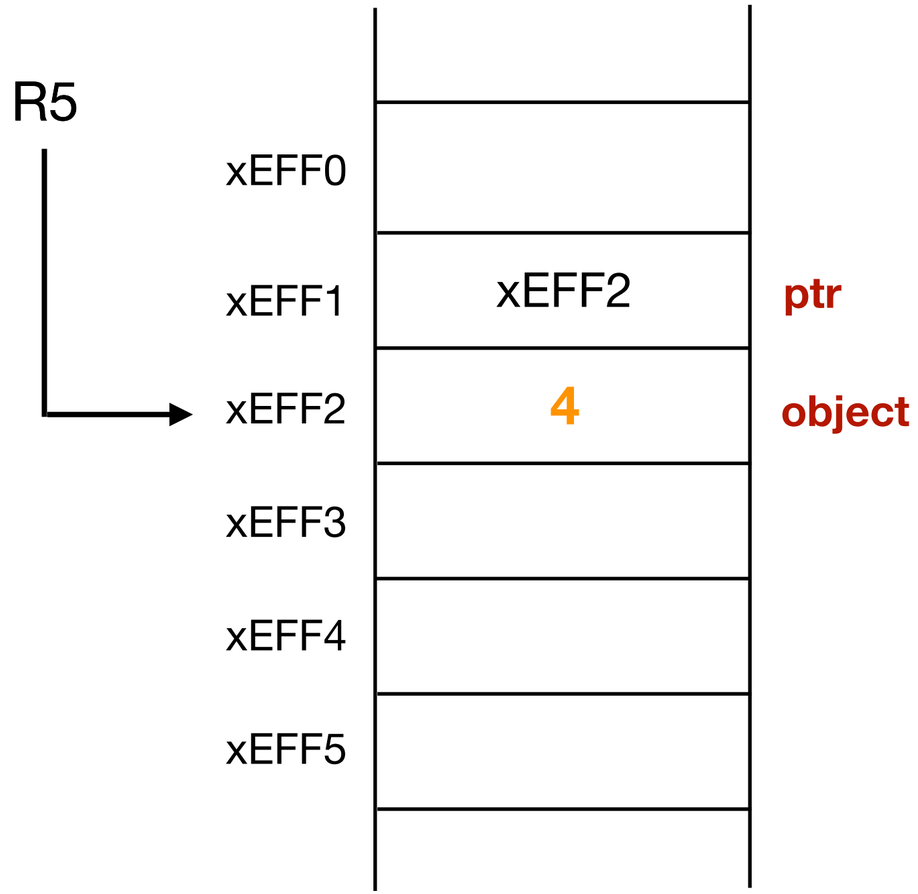

```
AND R0, R0, #0    ; Clear R0
ADD R0, R0, #4    ; R0 = 4
STR R0, R5, #0    ; object = 4

ADD R0, R5, #0    ; Generate memory address of object
STR R0, R5, #-1   ; ptr = &object
```

Pointers in LC-3



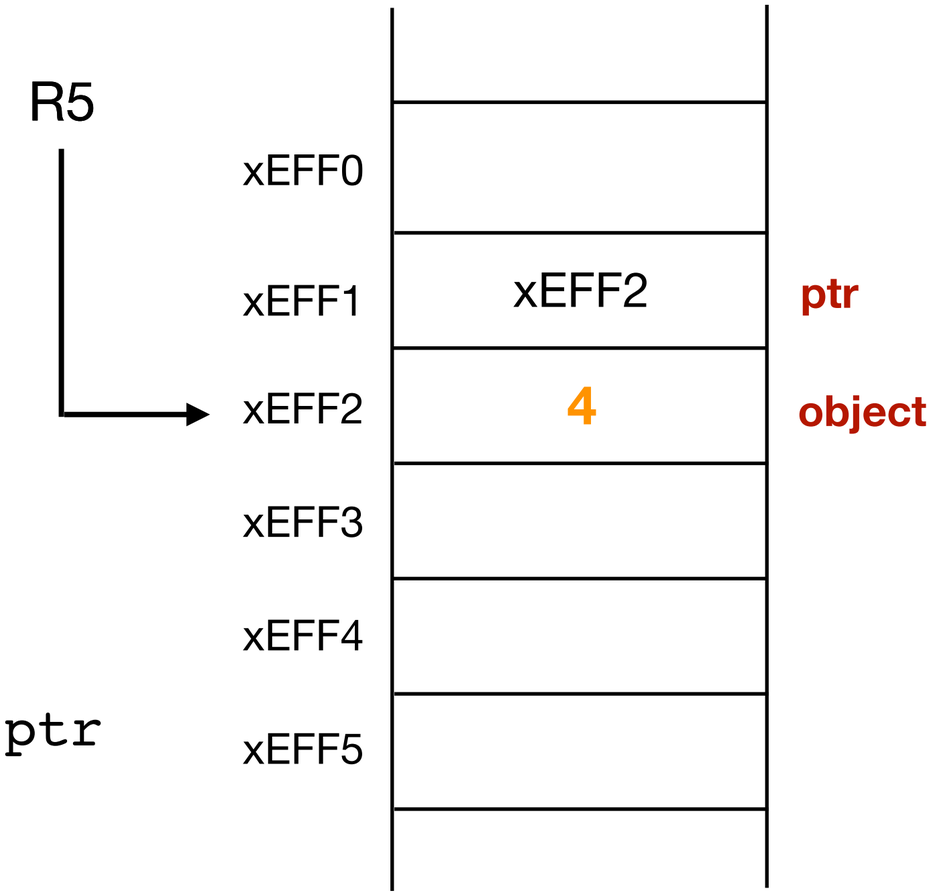
Pointers in LC-3

```
*ptr = *ptr + 1;
```



Pointers in LC-3

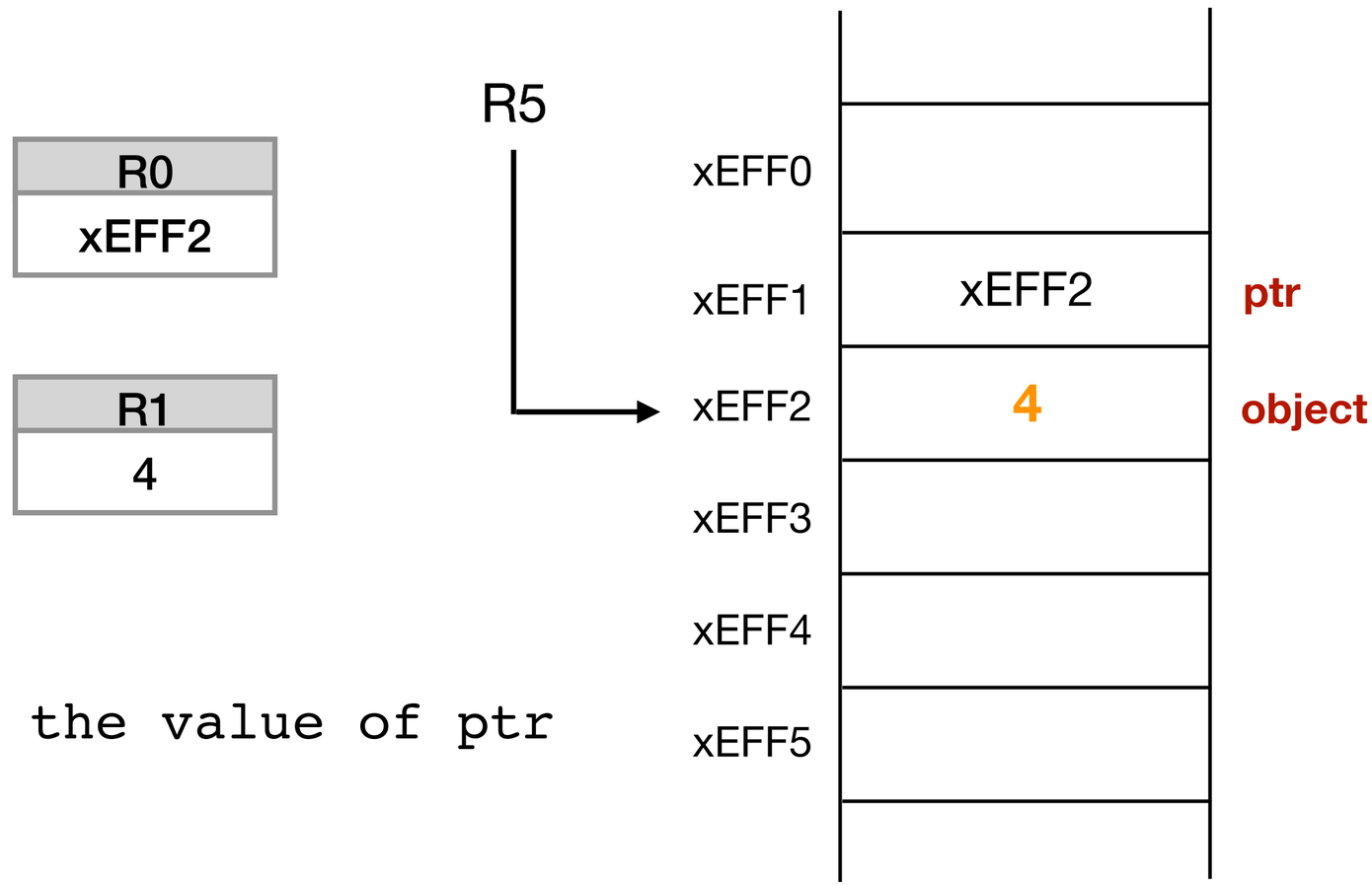
```
*ptr = *ptr + 1;
```




```
LDR R0, R5, #-1  ; R0 contains the value of ptr
```

Pointers in LC-3

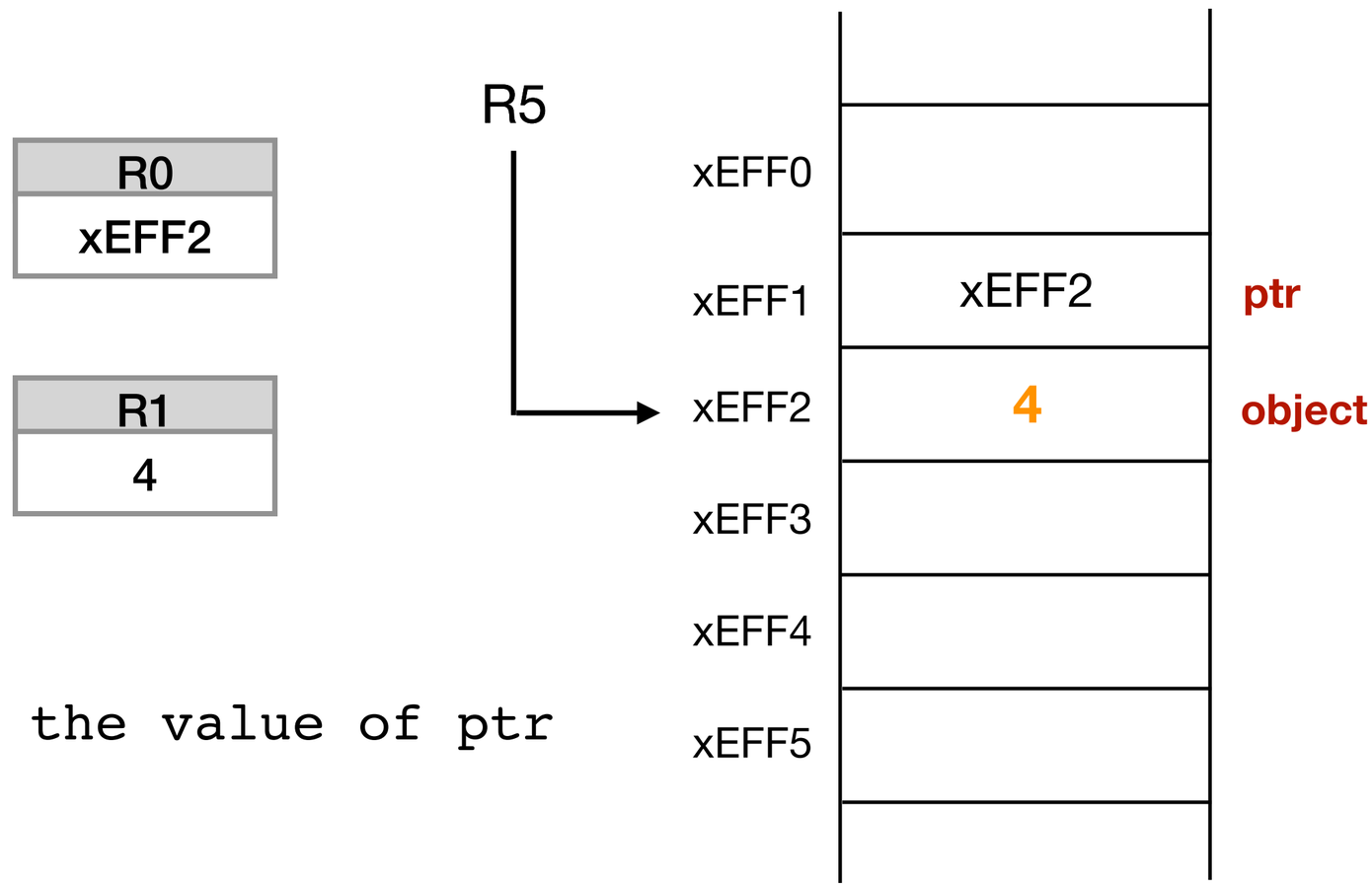
```
*ptr = *ptr + 1;
```




```
LDR R0, R5, #-1   ; R0 contains the value of ptr
LDR R1, R0, #0    ; R1 = *ptr
```

Pointers in LC-3

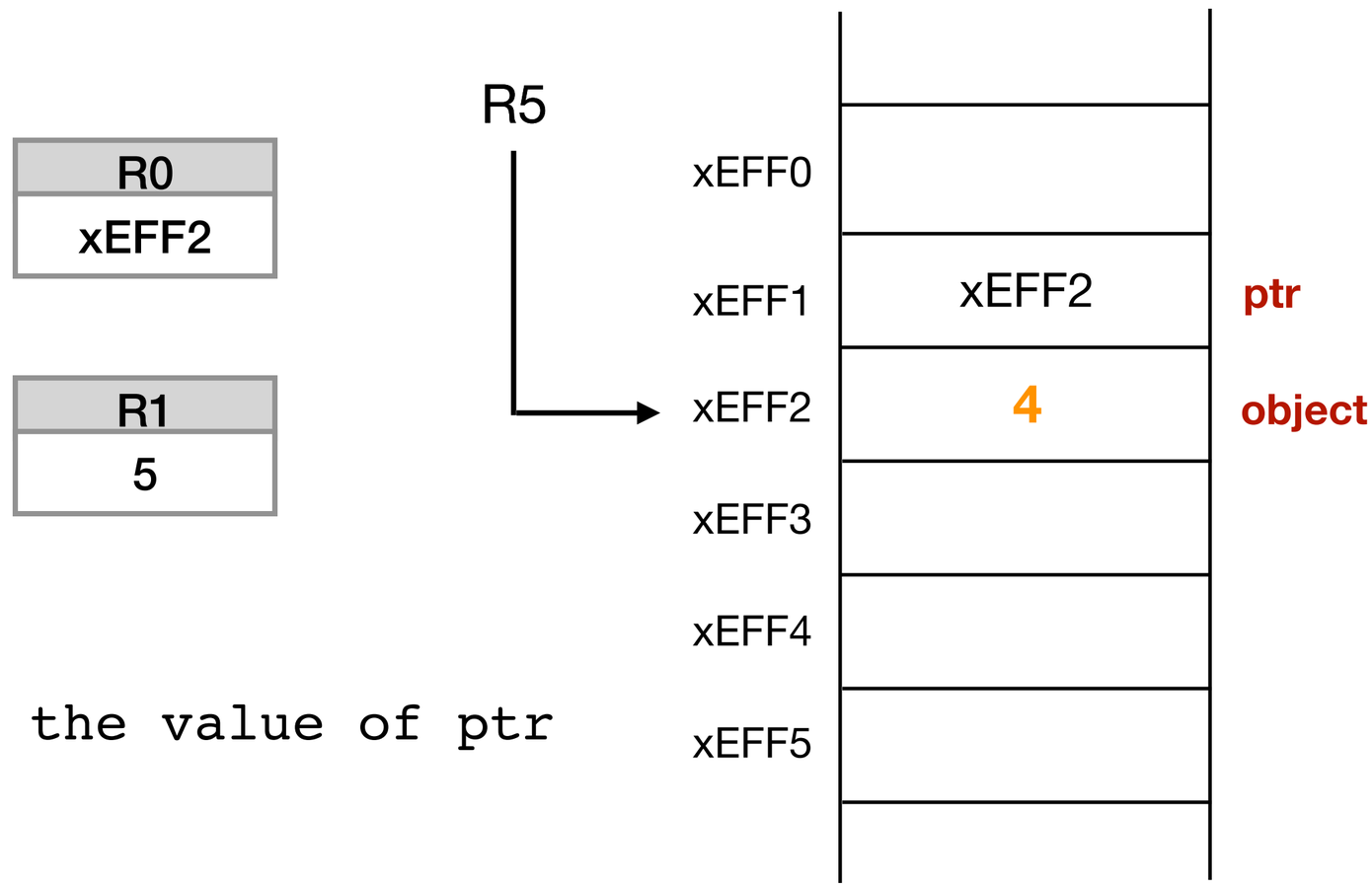
```
*ptr = *ptr + 1;
```




```
LDR R0, R5, #-1   ; R0 contains the value of ptr
LDR R1, R0, #0    ; R1 = *ptr


ADD R1, R1, #1    ; *ptr + 1
```

Pointers in LC-3

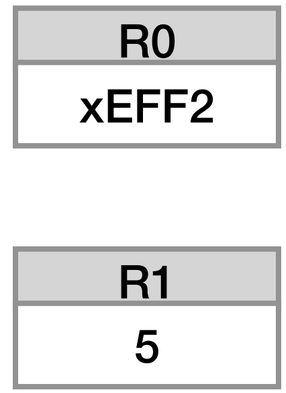
```
*ptr = *ptr + 1;
```

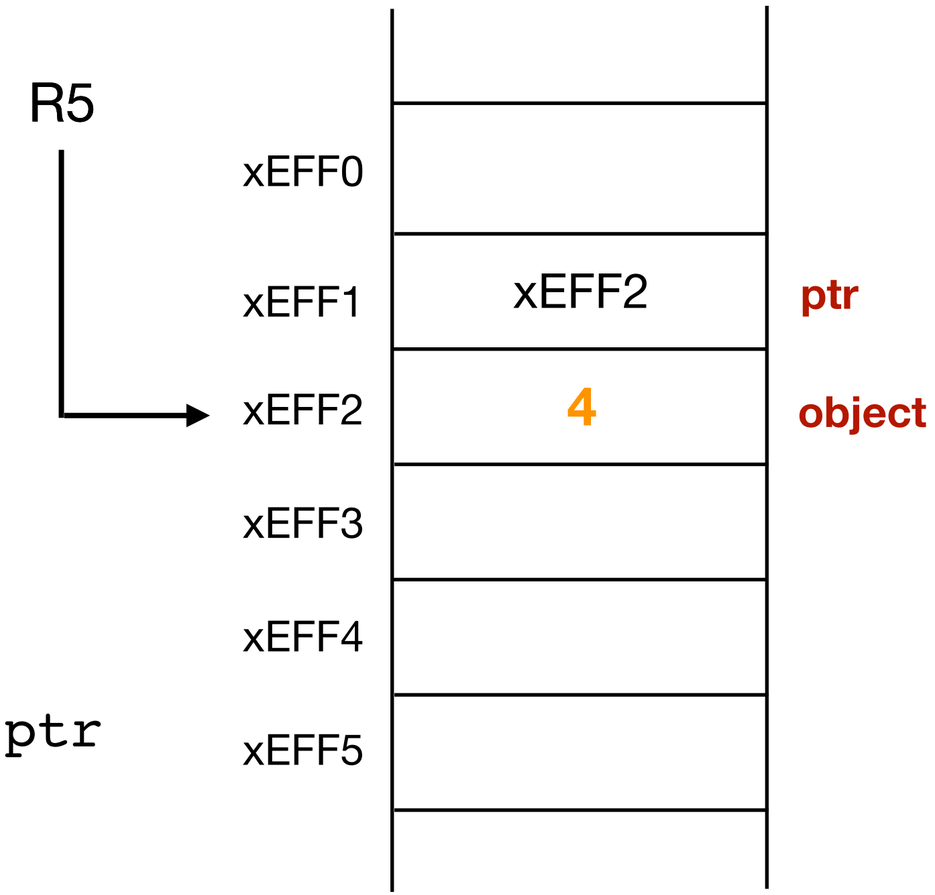



```
LDR R0, R5, #-1   ; R0 contains the value of ptr
LDR R1, R0, #0    ; R1 = *ptr


ADD R1, R1, #1    ; *ptr + 1
```

Pointers in LC-3

```
*ptr = *ptr + 1;
```

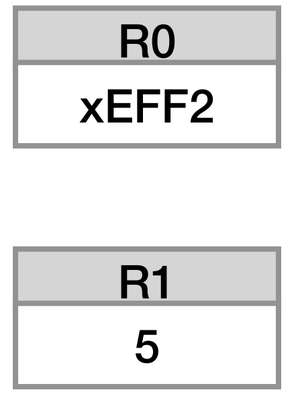


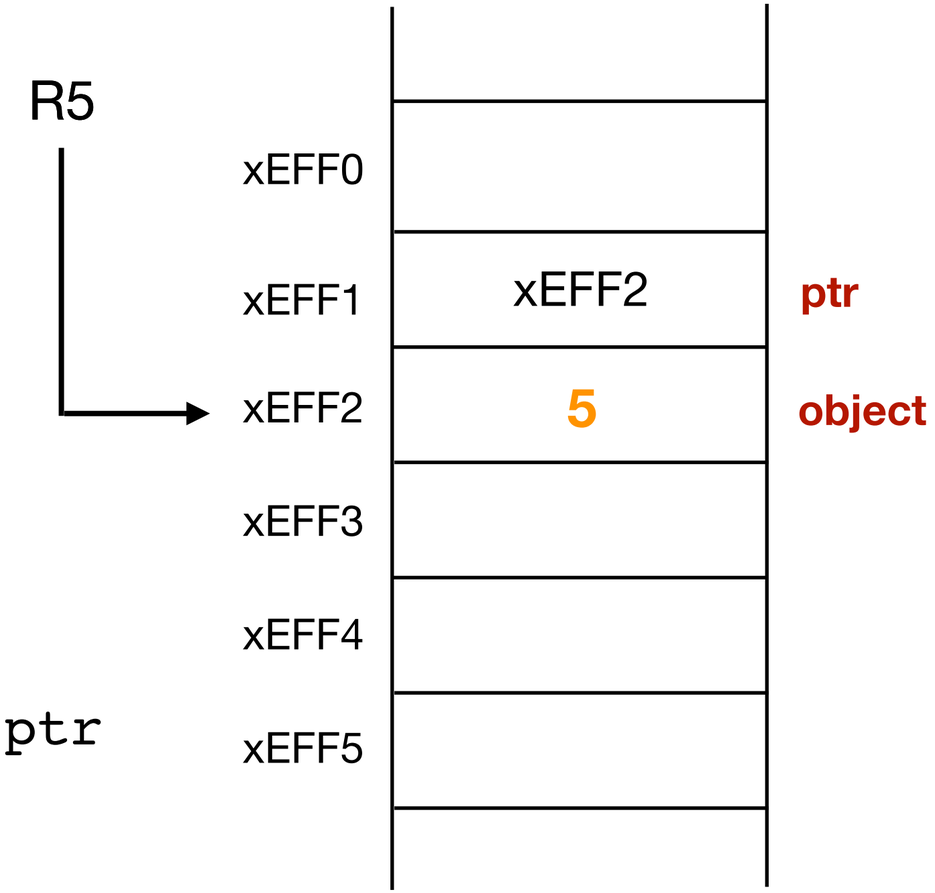



```
LDR R0, R5, #-1   ; R0 contains the value of ptr
LDR R1, R0, #0    ; R1 = *ptr


ADD R1, R1, #1    ; *ptr + 1
STR R1, R0, #0    ; *ptr = *ptr +1
```

Pointers in LC-3

```
*ptr = *ptr + 1;
```

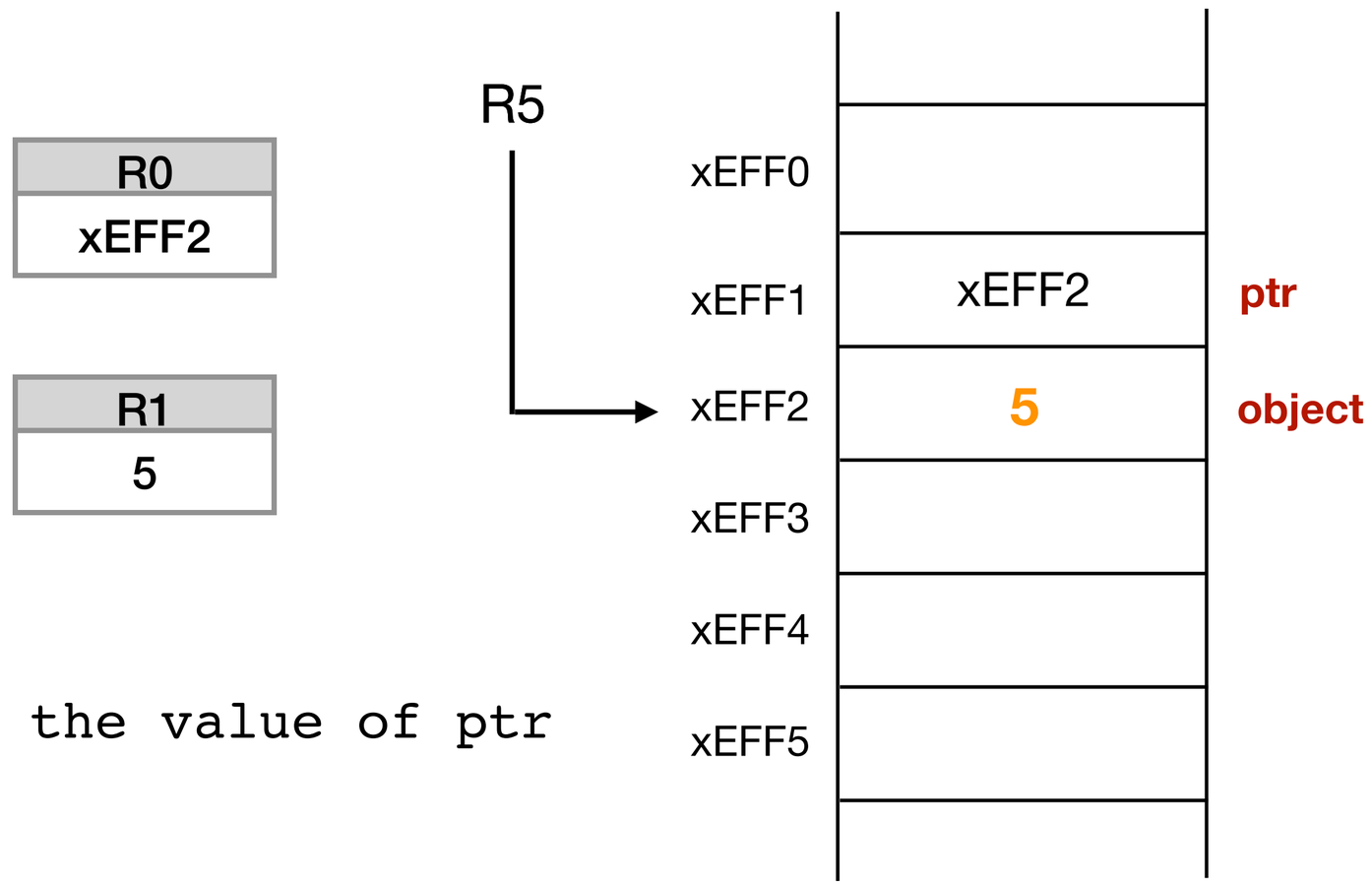





```
LDR R0, R5, #-1   ; R0 contains the value of ptr
LDR R1, R0, #0    ; R1 = *ptr


ADD R1, R1, #1    ; *ptr + 1
STR R1, R0, #0    ; *ptr = *ptr +1
```

Pointers in LC-3

```
*ptr = *ptr + 1;
```




```
LDR R0, R5, #-1   ; R0 contains the value of ptr
LDR R1, R0, #0    ; R1 = *ptr

ADD R1, R1, #1    ; *ptr + 1
STR R1, R0, #0    ; *ptr = *ptr +1
```

Why not?

```
STR R1, R5, #0
```

Arrays

Arrays

- A list of values arranged sequentially in memory

Arrays

- A list of values arranged sequentially in memory
- *Example*: a list of telephone numbers

Arrays

- A list of values arranged sequentially in memory
- *Example*: a list of telephone numbers
- Declaration syntax:

Arrays

- A list of values arranged sequentially in memory
- *Example*: a list of telephone numbers
- Declaration syntax:

```
type arrayName[arraySize];
```

Arrays

- A list of values arranged sequentially in memory
- *Example*: a list of telephone numbers
- Declaration syntax:

```
type arrayName[arraySize];
```

- `arraySize` has to be positive, nonzero and integer values

Arrays

- A list of values arranged sequentially in memory
- *Example*: a list of telephone numbers
- Declaration syntax:

```
type arrayName[arraySize];
```

- `arraySize` has to be positive, nonzero and integer values
- `type` is any valid C type

Arrays

Arrays

- Initializing arrays

Arrays

- Initializing arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Arrays

- Initializing arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

Arrays

- Initializing arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

balance[4] = 50.0;
```

Arrays

- Initializing arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

balance[4] = 50.0;
```

- Accessing elements?

Arrays

- Initializing arrays

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

balance[4] = 50.0;
```

- Accessing elements?
  - Expression **`a[4]`** refers to the 5th element of the array **`a`** (index starts from 0)

Arrays

- How do we calculate the length of an array?

Arrays

- How do we calculate the length of an array? `sizeof` function

Arrays

- How do we calculate the length of an array? `sizeof` function

```c
#include <stdio.h>

int main() {
  //simple array
  int arr[] = {19, 25, 8,  22, 17, 7,  84, 9,  19, 25, 10, 3,  1,
               7,  84, 9,  19, 25, 10, 3,  1,  8,  22, 17, 19, 25,
               10, 3,  1,  8,  22, 17, 7,  84, 9,  33, 1,  8,  22,
               17, 7,  84, 9,  19, 25, 10, 22, 17, 7,  84, 9,  19,
               25, 10, 3,  1,  8,  84, 9,  11, 23, 45, 5,  3};

  // using sizeof() operator to get length of array
  int len = sizeof(arr) / sizeof(arr[0]);

  printf("The length of int array is : %d ", len);
}
```

Arrays

- How do we calculate the length of an array? `sizeof` function

```c
#include <stdio.h>

int main() {
  //simple array
  int arr[] = {19, 25, 8,  22, 17, 7,  84, 9,  19, 25, 10, 3,  1,
               7,  84, 9,  19, 25, 10, 3,  1,  8,  22, 17, 19, 25,
               10, 3,  1,  8,  22, 17, 7,  84, 9,  33, 1,  8,  22,
               17, 7,  84, 9,  19, 25, 10, 22, 17, 7,  84, 9,  19,
               25, 10, 3,  1,  8,  84, 9,  11, 23, 45, 5,  3};

  // using sizeof() operator to get length of array
  int len = sizeof(arr) / sizeof(arr[0]);

  printf("The length of int array is : %d ", len);
}
```

Gives memory occupied by all of `arr`

Arrays

- How do we calculate the length of an array? `sizeof` function

```c
#include <stdio.h>

int main() {
  //simple array
  int arr[] = {19, 25, 8,  22, 17, 7,  84, 9,  19, 25, 10, 3,  1,
               7,  84, 9,  19, 25, 10, 3,  1,  8,  22, 17, 19, 25,
               10, 3,  1,  8,  22, 17, 7,  84, 9,  33, 1,  8,  22,
               17, 7,  84, 9,  19, 25, 10, 22, 17, 7,  84, 9,  19,
               25, 10, 3,  1,  8,  84, 9,  11, 23, 45, 5,  3};

  // using sizeof() operator to get length of array
  int len = sizeof(arr) / sizeof(arr[0]);

  printf("The length of int array is : %d ", len);
}
```

Gives memory occupied by all of `arr`

Gives memory occupied by `arr[0]`

Exercise

Using loops, write a C program that prompts the user for *five* integers one by one and stores them into an array `arr`. Then print out the five integers in a single line but in reverse order.

Exercise

Add a function `int my_first_sum` to the previous program which will take the list of five numbers and return their sum. Use this function to display the sum to the console instead of the numbers in reverse order.

Passing arrays

Passing arrays

- How did we let the compiler know `my_first_sum` takes an array of integers?

Passing arrays

- How did we let the compiler know `my_first_sum` takes an array of integers?

```
int my_first_sum(int array[]){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

Passing arrays

- How did we let the compiler know `my_first_sum` takes an array of integers?

```
int my_first_sum(int array[]){
   int i, sum=0;
   for (i=0; i<5; i++)
      sum = sum + array[i];
   return sum;
}
```

Passing arrays

- How did we let the compiler know `my_first_sum` takes an array of integers?

```
int my_first_sum(int array[]){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

- How did we pass the parameter `arr` to the function `my_first_sum` ?

Passing arrays

- How did we let the compiler know `my_first_sum` takes an array of integers?

- How did we pass the parameter `arr` to the function `my_first_sum` ?

```
int my_first_sum(int array[]){  printf("Sum is: %d \n", my_sum(arr));
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

Passing arrays

- How did we let the compiler know `my_first_sum` takes an array of integers?
- How did we pass the parameter `arr` to the function `my_first_sum` ?

```
int my_first_sum(int array[]){  printf("Sum is: %d \n", my_sum(arr));
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

Fact: The name of the array is *pointer* to the array!

Not convinced?

Not convinced?

- Replace the previous function with this one instead and try it out!

Not convinced?

- Replace the previous function with this one instead and try it out!

```
int my_second_sum(int *array){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

Not convinced?

- Replace the previous function with this one instead and try it out!

```
int my_second_sum(int *array){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

- The parameter declaration `int array[]` in the function definition is *syntactic sugar* for `int *array`.

Not convinced?

- Replace the previous function with this one instead and try it out!

```
int my_second_sum(int *array){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

- The parameter declaration `int array[]` in the function definition is *syntactic sugar* for `int *array`.
- However, `int p[]` makes it clear we are passing an array of integers while `int *p` … not so much.

Not convinced?

- Replace the previous function with this one instead and try it out!

```
int my_second_sum(int *array){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + array[i];
  return sum;
}
```

- The parameter declaration `int array[]` in the function definition is *syntactic sugar* for `int *array`.
- However, `int p[]` makes it clear we are passing an array of integers while `int *p` … not so much.

This is called pointer/array duality in C.

Pointer/array duality

Pointer/array duality

- In fact `arr[3]` is syntactic sugar for `*(arr + 3)` !!

Pointer/array duality

- In fact `arr[3]` is syntactic sugar for `*(arr + 3)` !!

```
int my_third_sum(int *arr){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + *(arr + i);
  return sum;
}
```

would also work just fine!

Pointer/array duality

- In fact `arr[3]` is syntactic sugar for `*(arr + 3)` !!

```
int my_third_sum(int *arr){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + *(arr + i);
  return sum;
}
```

would also work just fine!

- So is there a difference between `cptr` and `arr` in the below?

```
char arr[10];
char *cptr;
cptr = arr;
```

Pointer/array duality

- In fact `arr[3]` is syntactic sugar for `*(arr + 3)` !!

```
int my_third_sum(int *arr){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + *(arr + i);
  return sum;
}
```

would also work just fine!

- So is there a difference between `cptr` and `arr` in the below?

```
char arr[10];
char *cptr;
cptr = arr;
```

- Try doing:

```
cptr = cptr + 1;
arr = arr + 1;
```

Pointer/array duality

- In fact `arr[3]` is syntactic sugar for `*(arr + 3)` !!

```
int my_third_sum(int *arr){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum = sum + *(arr + i);
  return sum;
}
```

would also work just fine!

- So is there a difference between `cptr` and `arr` in the below?

```
char arr[10];
char *cptr;
cptr = arr;
```

- Try doing:

```
cptr = cptr + 1;
arr = arr + 1;
```

What gives?

Next time

- More pointer/array duality
- Arrays in LC3
- Variable length arrays
- Strings
- Multi-dimensional arrays

Good luck on the exam!