THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

COMPSCI 210

C Programming Language

Functions and Pointers

Function

```
return-type function-name(argument declarations)
{
  local variable declarations
  statements
}
```

Function Definition (header + body)

- Smaller, simpler, subcomponent of program
- Provides abstraction
  - hide low-level details
  - give high-level structure to program, easier to understand overall program flow
  - enables separable, independent development
- C functions
  - zero or multiple arguments passed in
  - single result returned (optional)
  - return value is always a particular type
- Declaration (also called prototype) at the beginning of the Code.
  - `int Factorial(int n);`
- Function call -- used in expression
  - `a = x + Factorial(f + g);`
- In other languages, called procedures, subroutines, ...

Example: 01function1.c

Function Declarations

- Function Declarations
  - Before you use a function within a program, you must declare it.
    - Compiler must know return and arg types and number of args
  - Function declaration consists of only the header part of the function definition
    - Note: the declaration ends with a semi-colon
    - Definition might be in a different file, written by a different programmer
      - Include a header file with declarations only
    - Note: If a function definition occurs in a program source file before the function call, it serve as both the definition and the declaration

```
void show_message();
...
int main(){
  ...
  for (count=0;count<msgNum;count++)
    show_message();
  ...
}
void show_message(){
  printf("Show message \n");
}
```

Declare

Define

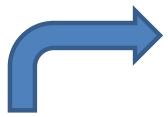
Must before main()

```
/* declare and define */
int Add(int n1, int n2) {
        return n1 + n2;
}
int main(){
  ...
  printf("%d\n", Add(x,result));
  ...
}
```

Example: checkRange.c, checkRange.asm

C Function to LC3

- A general transformation from C function to LC3 function is as follows:
  - Declare ranges: -999, +999
  - Use mulitple BR operators
  - Store output inside a Register



```
int checkRange(int input){
   if(input > -999 && input < 999){
      return 1; // true
   }
   else{
      return 0; // false
   }
}
```

```
RangeCheck    LD        R5,Neg999
              ADD       R4,R0,R5    ; Recall that R0 contains the
              BRp       BadRange    ; result being checked.
              LD        R5,Pos999
              ADD       R4,R0,R5
              BRn       BadRange
              AND       R5,R5,#0    ; R5 <-- success
              RET
BadRange      ST        R7,Save     ; R7 is needed by TRAP/RET
              LEA       R0,RangeErrorMsg
              TRAP      x22         ; Output character string
              LD        R7,Save
              AND       R5,R5,#0    ;
              ADD       R5,R5,#1    ; R5 <-- failure
              RET
Neg999        .FILL     #-999
Pos999        .FILL     #999
Save          .FILL     x0000
RangeErrorMsg .FILL     x000A
              .STRINGZ  "Error: Number is out of range."
```

Example: 02passByValue1.c

Pass by Value

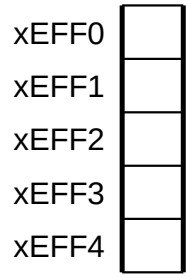
- Pass by value
  - C uses “call by value” to pass parameters between functions by duplicating the values
  - Changing the originally passed parameter in the calling function, does not change the variable value in the called function

```
void changer(int x){
  while(x){
    printf("changer: x=%d\n", x);
    x--;
  }
}
int main(){
  int i;
  i = 2;
  printf("before i=%d\n", i);
  changer(i);
  printf("after i=%d\n", i);
  ...
```

```
before i=2
changer: x=2
changer: x=1
after i=2
```

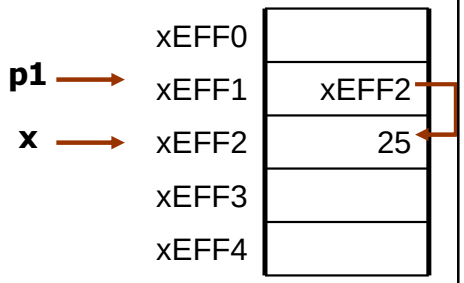
Pointers

Each memory location is labelled with an address, which arbitrarily selected to be in the xEFF0 range.
The actual address and unit are depends on the ISA of the computer.



- Pointer
  - Address of a variable in memory
  - Allows us to indirectly access variables
    - in other words, we can talk about its ***address*** rather than its ***value***
  - Example:
    - The variable x is stored at the address xEFF2.
      - The content is 25.
    - The pointer (p1) is stored at the address of xEFF1.
      - The content is xEFF2 (address of x)

```
int x = 25;
int *p1;
p1 = &x;
```



  - & Operator:
  - &p returns the address of variable z
  - Must be applied to a memory object, such as a variable. In other words, &3 is not allowed.
- * Operator
  - *p returns the value pointed to by p
  - Can be applied to any expression. All of these are legal

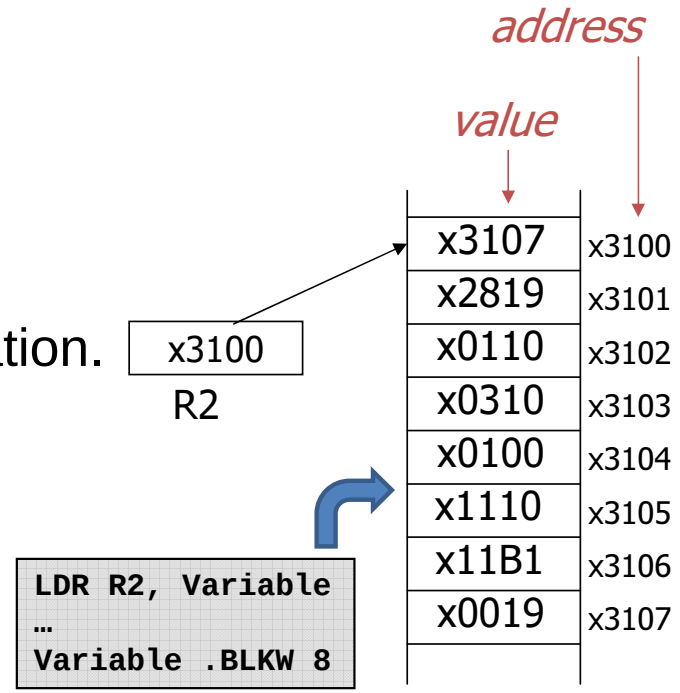
```
*var //contents of mem loc pointed to by var
**var //contents of mem loc pointed to by memory
location pointed to by var
*3 //contents of memory location 3
```

Address vs. Value

- Sometimes we want to deal with the address of a memory location, rather than the value it contains.

- Recall example from Chapter 6: adding a column of numbers.
  - R2 contains address of first location.
  - Read value, add to sum, and increment R2 until all numbers have been processed.

- R2 is a pointer -- it contains the address of data we're interested in.

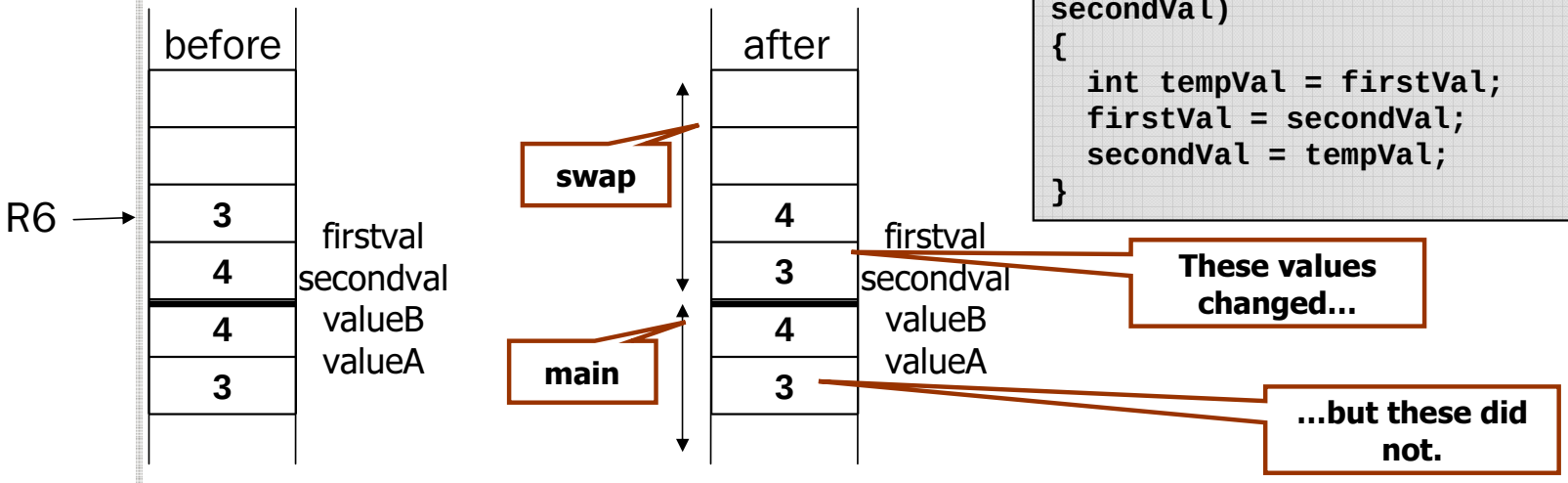


Example: swap.c

Another Need for Addresses

- Consider the following function that's supposed to swap the values of its arguments.
  - Swap needs addresses of variables outside its own activation record.

```
void Swap(int firstVal, int
secondVal)
{
  int tempVal = firstVal;
  firstVal = secondVal;
  secondVal = tempVal;
}
```



Pointers in C

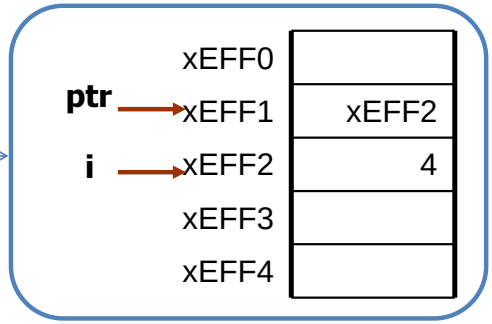
- C lets us talk about and manipulate pointers as variables and in expressions.
- Declaration
- `int *p;  /* p is a pointer to an int */`
- A pointer in C is always a pointer to a particular data type: `int*`, `double*`, `char*`, etc.
- `type *var;`
  `type* var;`
- Either of these work -- whitespace doesn't matter. Type of variable is `int*` (integer pointer), `char*` (char pointer), etc.
- Example:

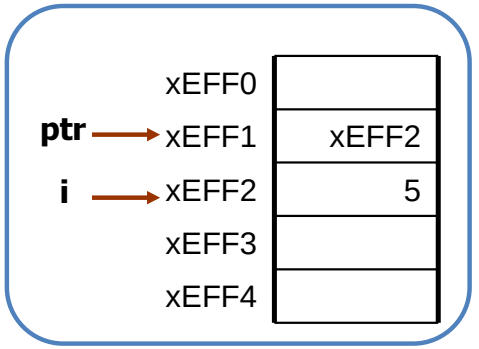
```
int i;
int *ptr;

i = 4;

ptr = &I;
*ptr = *ptr + 1;
```

C pointers





LC3 translation

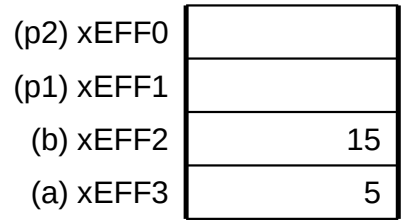
```
; i = 4;
AND  R0, R0, #0  ; clear R0
ADD  R0, R0, #4  ; put 4 in R0
STR  R0, R5, #0  ; store in i

; ptr = &i;
ADD  R0, R5, #0  ; R0 = R5 + 0 (addr of i)
STR  R0, R5, #-1 ; store in ptr

; *ptr = *ptr + 1;
LDR  R0, R5, #-1 ; R0 = ptr
LDR  R1, R0, #0  ; load contents (*ptr)
ADD  R1, R1, #1  ; add one
STR  R1, R0, #0  ; store result where R0
points
```

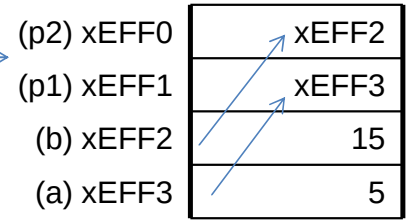
Example 1

```
int a=5, b=15;
int *p1, *p2;
```



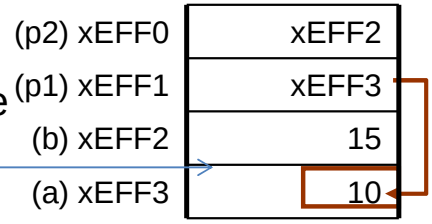
- pointer = &(var)
  - Pointer stores the address of the variable

```
p1 = &a;
p2 = &b;
```



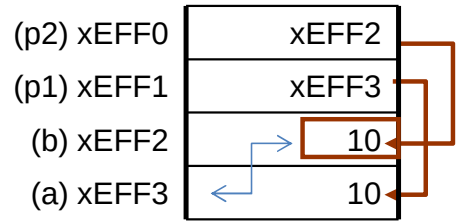
- *(pointer) = new value
  - Change the value pointed by the pointer to a new value
    - Note: A pointer must have a value before you can dereference it

```
*p1 = 10;
```



- *(pointer1) = *(pointer2)
  - Copy the value pointed from pointer2 to pointer1

```
*p2 = *p1;
```

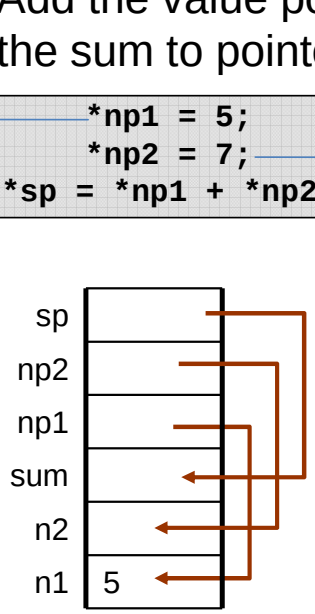


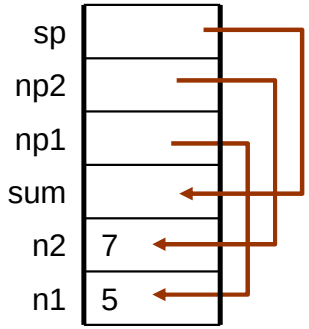
Example 2

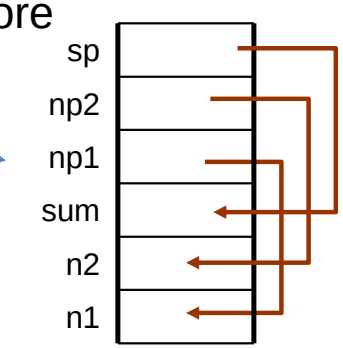
- *(pointer1) = *(pointer2) + *(pointer3)
  - Add the value pointed by pointer2 and pointer 3 and store the sum to pointer1

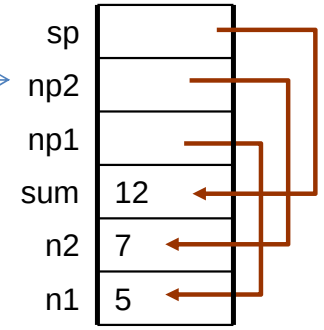
```
np1 = &n1;
np2 = &n2;
sp = &sum;
```

```
*np1 = 5;
*np2 = 7;
*sp = *np1 + *np2;
```



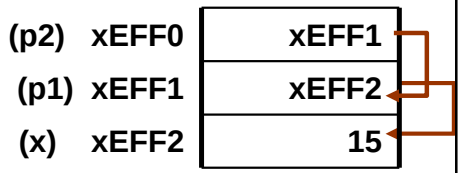






- Pointer to pointer

```
int x=15;
int *p1;
int **p2;
p1 = &x;
p2 = &p1;
```




```
int *x;
*x=3;
```

Error

Stdlib.h Atoi.c

int atoi (const char * str);

Convert string to integer

Parses the C string *str* interpreting its content as an integral number, which is returned as an int value.

1. Discards whitespace characters until the first non-whitespace character is found.
2. Starting from this character, takes an optional initial + or – sign
3. Takes as many numerical digits as possible, and interprets them as a numerical value.
4. Additional non numerical characters are ignored.
5. If no integral number characters encountered, the function does not perform conversion and returns 0!!
6. Instead use strtol

Stdlib.h Atoi.c

Main idea:

s pointer to string: CONST char *s;

1. Ignore white space characters
   - (c = *s) == 0x20
2. Ignore Tab character
   - (c = *s) == 0x09
3. Test for sign
   - (c = *s) == ‘+' || c == ‘-'
4. Read characters that represent decimal integer
   - (c = *s) >= '0' && c <= '9'
5. Convert ascii code to integer
   - **For** loop using length of string /* Use **strlen**
   - **Strlen** is a string function that determines the length of a C character string
   - value=10*value+c-'0'; /* ascii code 0x30 e.g. 48
6. Update character to which s points
   - s++;

Stdlib.h: use of atoi.c

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

  int main(void)
  {
    char *str = “210";
    int n = atoi(str);
    printf("The string %s contains the integer value: %d\n",str,n);

    char *str2 = “210forever";
    n = atoi(str2);
    printf("The string %s contains the integer value: %d\n",str2,n);
    return 0;
  }
```

Output:

The string 210 contains the integer value: 210

The string 210 contains the integer value: 210