CS2100 COMPUTER ORGANISATION

http://www.comp.nus.edu.sg/~cs2100/

Lecture #4

Pointers and Functions

NUS National University of Singapore | School of Computing

Lecture #4: Pointers and Functions (1/2)

Lecture #4: Pointers and Functions (2/2)

1. Pointers (1/3)

- While C is a high-level programming language, it is usually considered to be at the lower end of the spectrum due to a few reasons, among which are:
  - It has pointers which allow direct manipulation of memory contents
  - It has a set of bit manipulation operators, allowing efficient bitwise operations
- In Lecture #2 slide 11, we say that a variable has
  - a name (identifier);
  - a data type; and
  - an address.

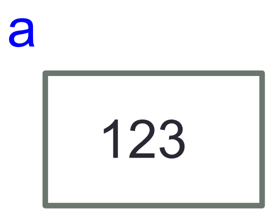
1. Pointers (2/3)

- A variable occupies some space in the computer memory, and hence it has an address.
- The programmer usually does not need to know the address of the variable (she simply refers to the variable by its name), but the system keeps track of the variable's address.

Data type Name

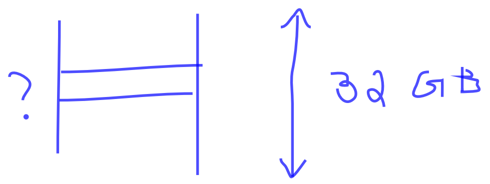
```
int a;
a = 123;
```

May only contain integer value



Where is variable a located in the memory?

1. Pointers (3/3)



- You may refer to the address of a variable by using the address operator & (ampersand)

Address.c

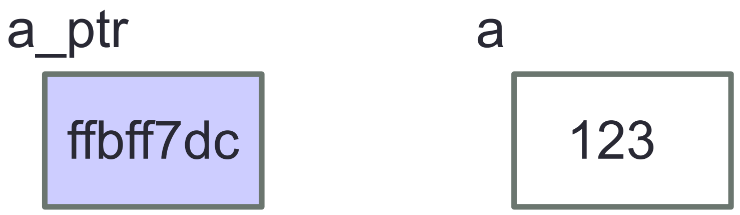
```
int a = 123;
printf("a = %d\n", a);
printf("&a = %p\n", &a);
```

```
a = 123
&a = ffbff7dc
```

- %p is used as the format specifier for addresses
- Addresses are printed out in hexadecimal (base 16) format
- The address of a variable varies from run to run, as the system allocates any free memory to the variable

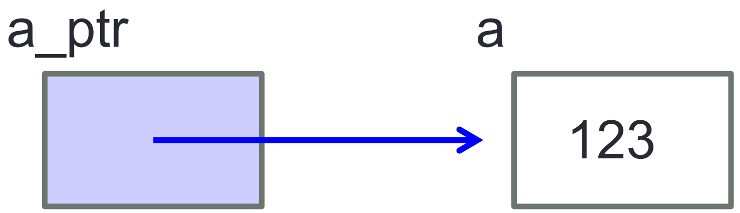
1.1 Pointer Variable

- A variable that contains the address of another variable is called a pointer variable, or simply, a pointer.
- Example: a pointer variable a_ptr is shown as a blue box below. It contains the address of variable a.



Assuming that variable a is located at address ffbff7dc.

- Variable a_ptr is said to be pointing to variable a.
- If the address of a is immaterial, we simply draw an arrow from the blue box to the variable it points to.



1.2 Declaring a Pointer

Syntax:

```
type *pointer_name;
```

- pointer_name is the name (identifier) of the pointer
- type is the data type of the variable this pointer may point to

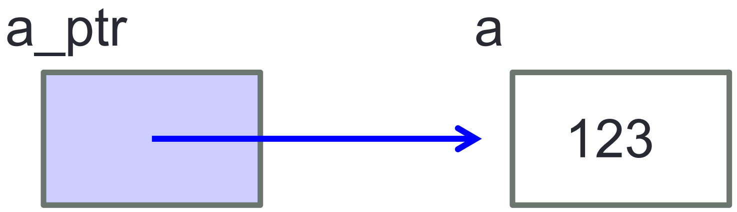
- Example: The following statement declares a pointer variable a_ptr which may point to any int variable
- Good practice to name a pointer with suffix _ptr or _p

```
int *a_ptr;
```

1.3 Assigning Value to a Pointer

- Since a pointer contains an address, only an address may be assigned to a pointer
- Example: Assigning address of a to a_ptr

```
int a = 123;
int *a_ptr; // declaring an int pointer

a_ptr = &a;
```



- We may initialise a pointer during its declaration:

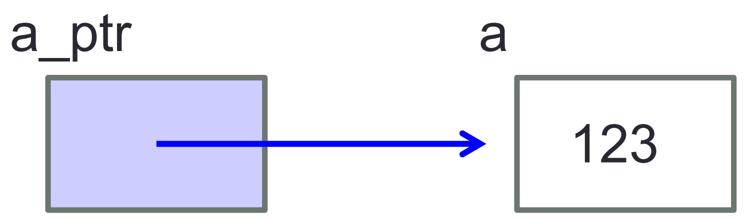
```
int a = 123;
int *a_ptr = &a; // initialising a_ptr
```

Visualization

```
int a = 123;
int *a_ptr;
a_ptr = &a;
```

address	name	value
…	…	…
ffbff7dc	a	123
…	…	…
ffbff7ff	a_ptr	ffbff7dc
…	…	…

1.4 Accessing Variable Through Pointer



- Once we make a_ptr points to a (as shown above), we can now access a directly as usual, or indirectly through a_ptr by using the indirection operator (also called dereferencing operator) *

```
printf("a = %d\n", *a_ptr);
```

≡

```
printf("a = %d\n", a);
```

```
*a_ptr = 456;
```

≡

```
a = 456;
```

Hence, *a_ptr is synonymous with a

1.5 Example #1

```
int i = 10, j = 20;
int *p; // p is a pointer to some int variable

p = &i; // p now stores the address of variable i
```

Important! Now `*p` is equivalent to `i`

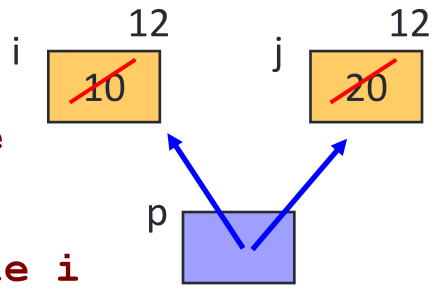
```
printf("value of i is %d\n", *p);
```

```
value of i is 10
```

```
// *p accesses the value of pointed/referred variable
*p = *p + 2; // increment *p (which is i) by 2
             // same effect as: i = i + 2;
p = &j; // p now stores the address of variable j
```

Important! Now `*p` is equivalent to `j`

```
*p = i;  // value of *p (which is j now) becomes 12
         // same effect as: j = i;
```

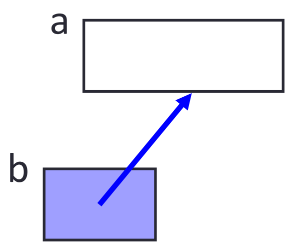


1.6 Example #2 (1/2)

Pointer.c

```
#include <stdio.h>

int main(void) {
   double a, *b;

   b = &a;
   *b = 12.34;
   printf("%f\n", a);

   return 0;
}
```

Can you draw the picture?
What is the output?



```
12.340000
```

What is the output if the **`printf()`** statement is changed to the following?

```
printf("%f\n", *b);
```

```
12.340000
```

```
printf("%f\n", b);
```

Compile with warning

```
printf("%f\n", *a);
```

Error

What is the proper way to print a pointer?
(Seldom need to do this.)

```
printf("%p\n", b);
```

Value in hexadecimal; varies from run to run.

```
ffbff6a0
```

1.6 Example #2 (2/2)

- How do we interpret the declaration?

```
double a, *b;
```

- The above is equivalent to

```
double a;  // this is straight-forward: a is a double variable
double *b;
```

- We can read the second declaration as
  - *b is a double variable, so this implies that ...
  - b is a pointer to some double variable
- The following are equivalent:

```
double a;
double *b;
b = &a;
```

```
double a;
double *b = &a;
```

But this is not the same as above (and it is not legal):

```
double a;
double b = &a;
```

✗

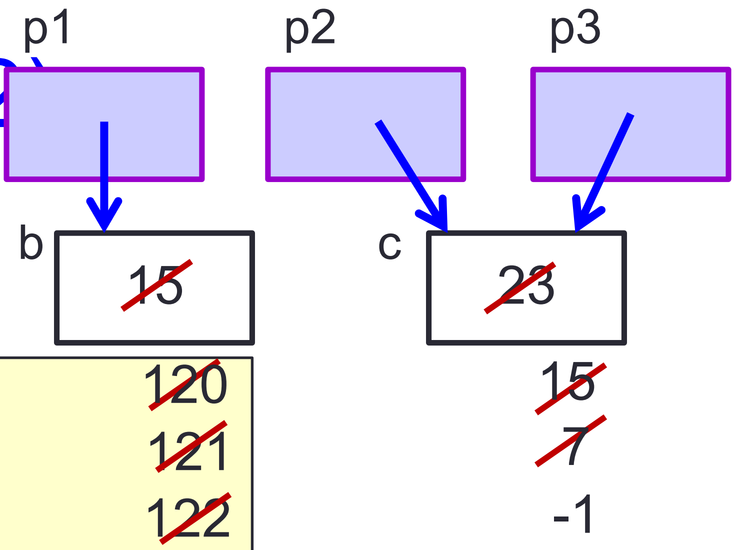
1.7 Tracing Pointers (1/2)

- Trace the code below manually to obtain the outputs.
- Compare your outputs with your neighbours.

TracePointers.c

```
int a = 8, b = 15, c = 23;
int *p1, *p2, *p3;

p1 = &b;
p2 = &c;
p3 = p2;
printf("1: %d %d %d\n", *p1, *p2, *p3);

*p1 *= a;
while (*p2 > 0) {
  *p2 -= a;
  (*p1)++;
}
printf("2: %d %d %d\n", *p1, *p2, *p3);
printf("3: %d %d %d\n", a, b, c);
```

1.7 Tracing Pointers (2/2)



```
int a = 8, b = 15, c = 23;
int *p1, *p2, *p3;

p1 = &b;
p2 = &c;
p3 = p2;
printf("1: %d %d %d\n", *p1, *p2, *p3);

*p1 *= a;
while (*p2 > 0) {
  *p2 -= a;
  (*p1)++;
}
printf("2: %d %d %d\n", *p1, *p2, *p3);
printf("3: %d %d %d\n", a, b, c);
```

```
1: 15 23 23
```

```
2: 123 -1 -1
```

```
3: 8 123 -1
```

1.8 Incrementing a Pointer

- If p is a pointer variable, what does p = p + 1 (or p++) mean?

Recall Lect#2 slide 15:
int takes up 4 bytes
float takes up 4 bytes
char takes up 1 byte
double takes up 8 bytes

```c
int a; float b; char c; double d;
int *ap; float *bp;
char *cp; double *dp;

ap = &a; bp = &b; cp = &c; dp = &d;
printf("%p %p %p %p\n", ap, bp, cp, dp);
```

```
ffbff0a4 ffbff0a0 ffbff09f ffbff090
```

+4 +4 +1 +8

```c
ap++; bp++; cp++; dp++;
printf("%p %p %p %p\n", ap, bp, cp, dp);
```

```
ffbff0a8 ffbff0a4 ffbff0a0 ffbff098
```

+12

```c
ap += 3;
printf("%p\n", ap);
```

+12

```
ffbff0b4
```

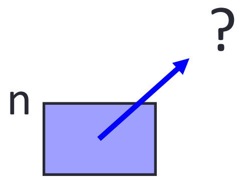
IncrementPointers.c

1.9 Common Mistake

CommonMistake.c

```
int *n;

*n = 123;
printf("%d\n", *n);
```

What's wrong with this?
Can you draw the picture?



- Where is the pointer n pointing to?
- Where is the value 123 assigned to?
- Result: Segmentation Fault (core dumped)
  - Remove the file "core" from your directory. It takes up a lot of space!

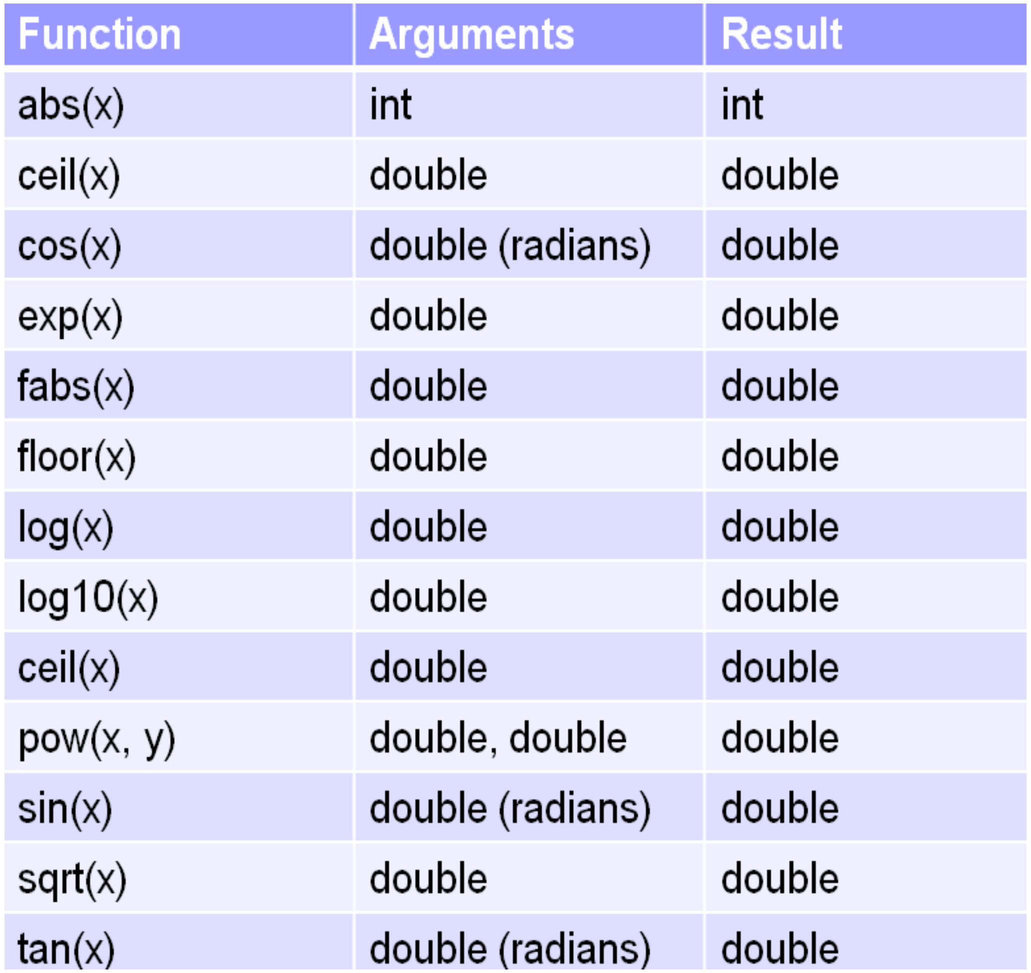
1.10 Why Do We Use Pointers?

- It might appear that having a pointer to point to a variable is redundant since we can access the variable directly
- The purpose of pointers is apparent later when we pass the address of a variable into a function, for example, in the following scenarios:
  - To pass the addresses of two or more variables to a function so that the function can pass back to its caller new values for the variables
  - To pass the address of the first element of an array to a function so that the function can access all elements in the array

2. Calling Functions (1/3)

- In C, there are many libraries offering functions for you to use.
- Eg: scanf() and printf() – requires to include <stdio.h>
- C provides many libraries, for example, the math library
- To use math functions, you need to
  - Include <math.h> AND
  - Compile your program with –lm option (i.e. gcc –lm …) in sunfire
- See table (next slide) for some math functions

2. Calling Functions (2/3)



Function	Arguments	Result
abs(x)	int	int
ceil(x)	double	double
cos(x)	double (radians)	double
exp(x)	double	double
fabs(x)	double	double
floor(x)	double	double
log(x)	double	double
log10(x)	double	double
ceil(x)	double	double
pow(x, y)	double, double	double
sin(x)	double (radians)	double
sqrt(x)	double	double
tan(x)	double (radians)	double

Function prototype:

double pow(double x, double y)

function return type

2. Calling Functions (3/3)

To link to Math library

MathFunctions.c

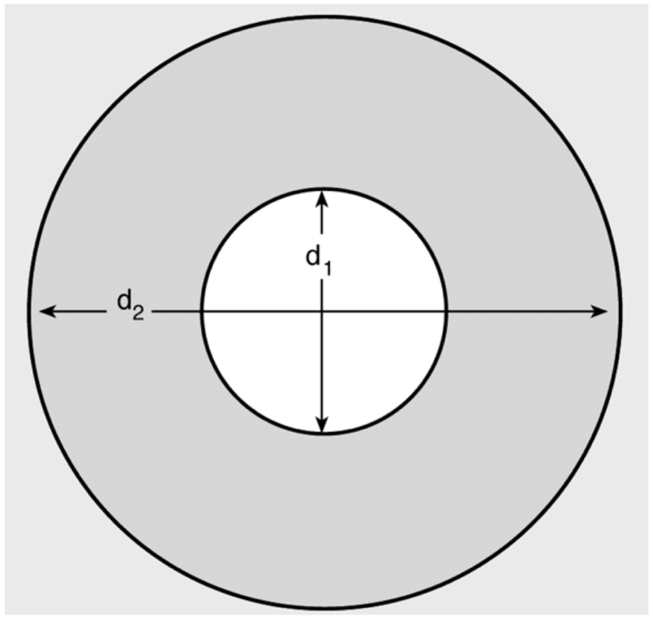
```
#include <stdio.h>
#include <math.h>

int main(void) {
   int   x, y;
   float val;

   printf("Enter x and y: ");
   scanf("%d %d", &x, &y);
   printf("pow(%d, %d) = %f\n", x, y, pow(x,y));

   printf("Enter value: ");
   scanf("%f", &val);
   printf("sqrt(%f) = %f\n", val, sqrt(val));

   return 0;
}
```

```
$ gcc -lm MathFunctions.c
$ a.out
Enter x and y: 3 4
pow(3,4) = 81.000000
Enter value: 65.4
sqrt(65.400002) = 8.087027
```

3. User-Defined Functions (1/6)

- We can define and use our own functions

Example: Compute the volume of a flat washer. Dimensions of a flat washer are usually given as an inner diameter, an outer diameter, and a thickness.



rim area = $\pi(d_2/2)^2 - \pi(d_1/2)^2$

3. User-Defined Functions (2/6)

Washer.c

```
#include <stdio.h>
#include <math.h>
#define PI 3.14159
int main(void) {
    double d1, // inner diameter
           d2, // outer diameter
           thickness, outer_area, inner_area, rim_area, volume;

    // read input data
    printf("Enter inner diameter, outer diameter, thickness: ");
    scanf("%lf %lf %lf", &d1, &d2, &thickness);

    // compute volume of a washer
    outer_area = PI * pow(d2/2, 2);
    inner_area = PI * pow(d1/2, 2);
    rim_area = outer_area - inner_area;
    volume = rim_area * thickness;

    printf("Volume of washer = %.2f\n", volume);
    return 0;
}
```

```
Enter …: 8.2 10.5 2.2
Volume of washer = 74.32
```

3. User-Defined Functions (3/6)

- Note that area of circle is computed twice. For code reusability, it is better to define a function to compute area of a circle.

```
double circle_area(double diameter) {
   return PI * pow(diameter/2, 2);
}
```

- We can then call/invoke this function whenever we need it.

`circle_area(d2)` → to compute area of circle with diameter d2

`circle_area(d1)` → to compute area of circle with diameter d1

3. User-Defined Functions (4/6)

WasherV2.c

```c
#include <stdio.h>
#include <math.h>
#define PI 3.14159

double circle_area(double);

int main(void) {
    // code similar to Washer.c; omitted here

    // compute volume of a washer
    rim_area = circle_area(d2) - circle_area(d1);
    volume = rim_area * thickness;

    printf("Volume of washer = %.2f\n", volume);
    return 0;
}

// This function returns the area of a circle
double circle_area(double diameter) {
    return PI * pow(diameter/2, 2);
}
```

Function prototype

Function definition

3. User-Defined Functions (5/6)

- It is a good practice to put function prototypes at the top of the program, <u>before</u> the main() function, to inform the compiler of the functions that your program may use and their return types and parameter types.
- A function prototype includes only the function's return type, the function's name, and the data types of the parameters (names of parameters are optional).
- Function definitions to follow <u>after</u> the main() function.
- Without function prototypes, you will get error/warning messages from the compiler.

3. User-Defined Functions (6/6)

- Let’s remove (or comment off) the function prototype for circle_area() in WashersV2.c
- Messages from compiler:

```
WashersV2.c: In function ‘main’:
WashersV2.c:19:2: warning: implicit declaration of function
‘circle_area’ [-Wimplicit-function-declaration]
  rim_area = circle_area(d2) - circle_area(d1);
  ^
WasherV2.c: At top level:
WashersV2.c:27:8: error: conflicting types for ‘circle-area’
:
```

- Without function prototype, compiler assumes the default (implicit) return type of int for circle_area() when the function is used in line 19, which conflicts with the function header of circle_area() when the compiler encounters the function definition later in line 27.

4. Pass-by-Value and Scope Rule (1/4)

- In C, the actual parameters are passed to the formal parameters by a mechanism known as pass-by-value.

```
int main(void) {

  double a = 10.5, b = 7.8;
  printf("%.2f\n", sqrt_sum_square(3.2, 12/5);
  printf("%.2f\n", sqrt_sum_square(a, a+b);
  return 0;
}
```

```
double sqrt_sum_square(double x, double y) {

  double sum_square;
  sum_square = pow(x,2) + pow(y,2);
  return sqrt(sum_square);
}
```

a: 10.5 b: 7.8

Actual parameters:

3.2 and 2.0

10.5 and 18.3

Formal parameters:

x: 3.2 / 10.5 y: 2.0 / 18.3

4. Pass-by-Value and Scope Rule (2/4)

- Formal parameters are local to the function they are declared in.
- Variables declared within the function are also local to the function.
- Local parameters and variables are only accessible in the function they are declared – scope rule.
- When a function is called, an activation record is created in the call stack, and memory is allocated for the local parameters and variables of the function.
- Once the function is done, the activation record is removed, and memory allocated for the local parameters and variables is released.
- Hence, local parameters and variables of a function exist in memory only during the execution of the function. They are called automatic variables.
- In contrast, static variables exist in the memory even after the function is executed.

4. Pass-by-Value and Scope Rule (3/4)

- What's wrong with this code?

```c
int f(int);

int main(void) {
  int a;
  ...
}

int f(int x) {
  return a + x;
}
```

Answer:
Variable a is local to main(), not f(). Hence, variable a cannot be used in f().

4. Pass-by-Value and Scope Rule (4/4)

- Trace this code by hand and write out its output.

```
In main, before: a=2, b=3
In g, before: a=2, b=3
In g, after : a=102, b=203
In main, after : a=2, b=3
```

A void function is a function that does not return any value.

```c
#include <stdio.h>
void g(int, int);

int main(void) {
  int a = 2, b = 3;

  printf("In main, before: a=%d, b=%d\n", a, b);
  g(a, b);
  printf("In main, after : a=%d, b=%d\n", a, b);
  return 0;
}

void g(int a, int b) {
  printf("In g, before: a=%d, b=%d\n", a, b);
  a = 100 + a;
  b = 200 + b;
  printf("In g, after : a=%d, b=%d\n", a, b);
}
```

PassByValue.c

main

addr	name	val
_	a	2
_	b	3

g

addr	name	val
_	a	102
_	b	203

4.1 Consequence of Pass-by-Value

- Can this code be used to swap the values in **`a`** and **`b`**?

```
In main, before: a=2, b=3
In main, after : a=2, b=3
```

```c
#include <stdio.h>
void swap(int, int);

int main(void) {
  int a = 2, b = 3;

  printf("In main, before: a=%d, b=%d\n", a, b);
  swap(a, b);
  printf("In main, after : a=%d, b=%d\n", a, b);
  return 0;
}

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

SwapIncorrect.c

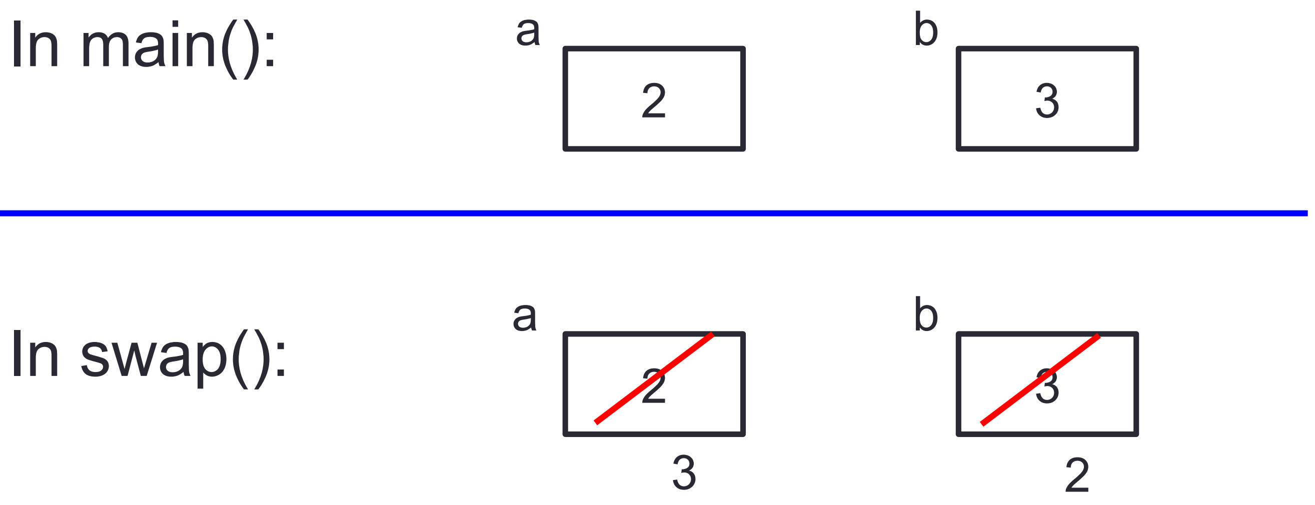
No

5. Function with Pointer Parameters (1/3)

- A function may not return any value (called a void function), or it may return a value.
- All parameters and variables in a function are local to the function (scope rule).
- Arguments from a caller are passed by value to a function's parameters.
- How do we then allow a function to return more than one value, or modify values of variables defined outside it?
- An example is swapping two variables. How can we write a function to do that? The previous slide shows a negative example.

5. Function with Pointer Parameters (2/3)

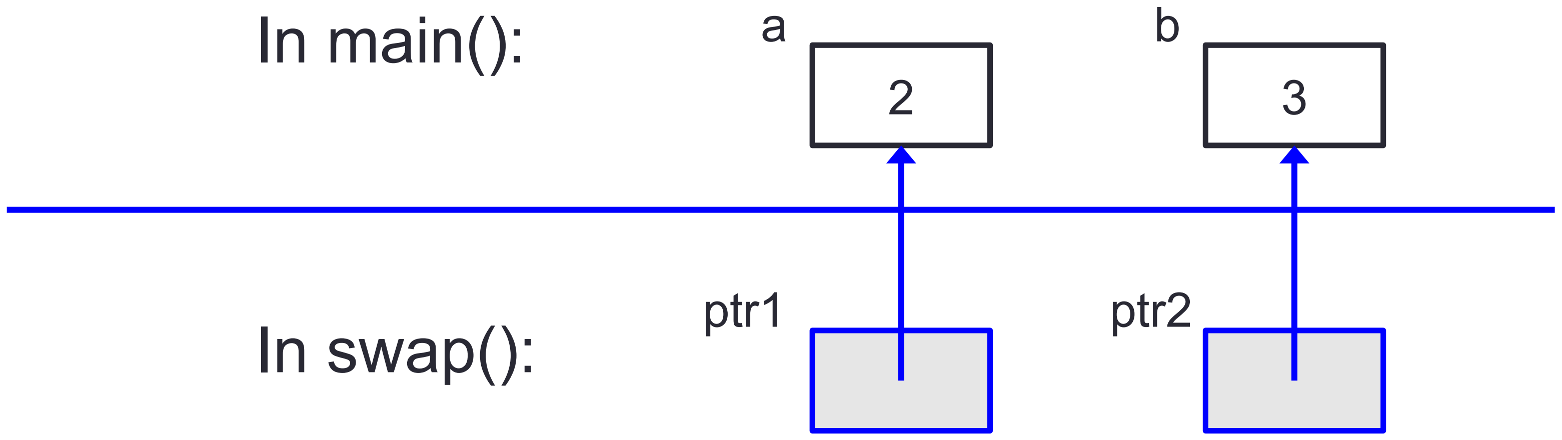
- What happens in SwapIncorrect.c?
- It's all about pass-by-value and scope rule!



- No way for swap() to modify the values of variables that are outside its scope (i.e. a and b), unless...

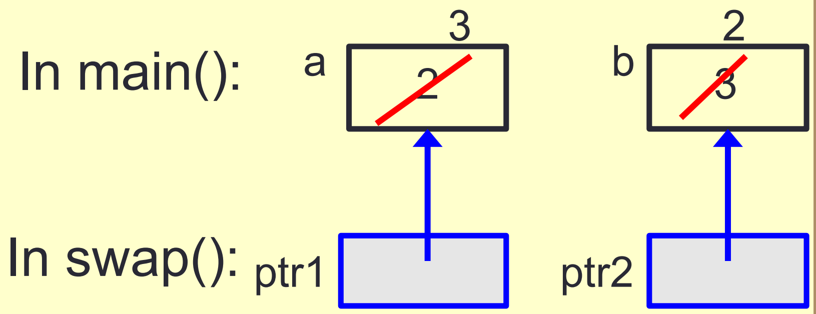
5. Function with Pointer Parameters (3/3)

- The only way for a function to modify the value of a variable outside its scope, is to find a way for the function to access that variable
- Solution: Use pointers!



5.1 Function To Swap Two Variables

```
#include <stdio.h>

void swap(int *, int *);

int main(void) {
   int a, b;

   printf("Enter two integers: ");
   scanf("%d %d", &var1, &var2);

   swap( &a, &b );

   printf("var1 = %d; var2 = %d\n", var1, var2);
   return 0;
}

void swap(int *ptr1, int *ptr2) {
   int temp;
   temp = *ptr1; *ptr1 = *ptr2; *ptr2 = temp;
}
```

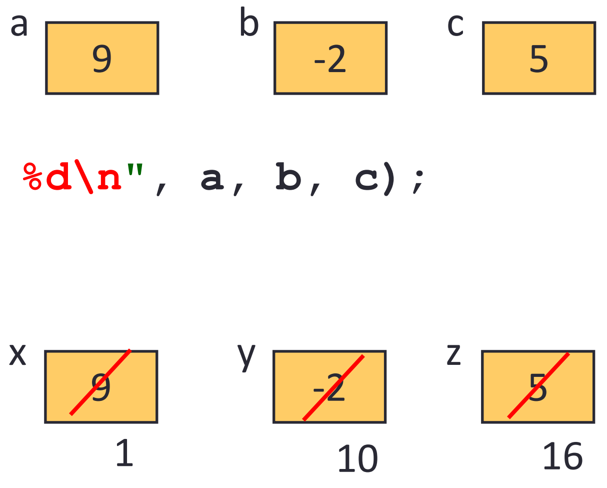


SwapCorrect.c

5.2 Examples (1/4)

Example1.c

```c
#include <stdio.h>
void f(int, int, int);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(a, b, c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int x, int y, int z) {
    x = 3 + y;
    y = 10 * x;
    z = x + y + z;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
}
```

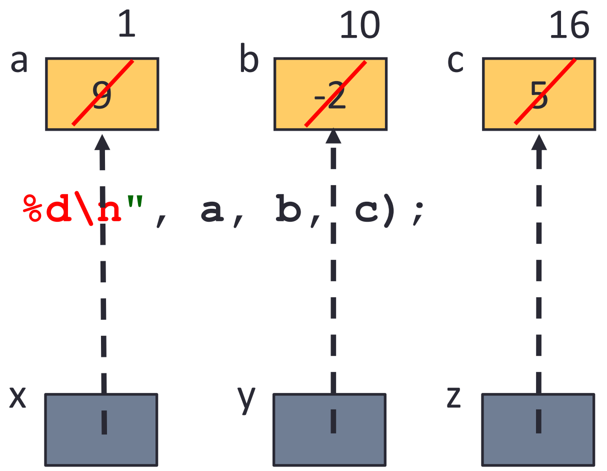


```
x = 1, y = 10, z = 16
a = 9, b = -2, c = 5
```

5.2 Examples (2/4)

Example2.c

```
#include <stdio.h>
void f(int *, int *, int *);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int *x, int *y, int *z)
{
    *x = 3 + *y;
    *y = 10 * *x;
    *z = *x + *y + *z;
    printf("*x = %d, *y = %d, *z = %d\n", *x, *y, *z);
}
```



*x is a, *y is b, and *z is c!

```
*x = 1, *y = 10, *z = 16
a = 1, b = 10, c = 16
```

5.2 Examples (3/4)

Example3.c

```
#include <stdio.h>
void f(int *, int *, int *);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int *x, int *y, int *z)
{
    *x = 3 + *y;
    *y = 10 * *x;
    *z = *x + *y + *z;
    printf("x = %d, y = %d, z = %d\n", x, y, z);
}
```

Compiler warnings, because x, y, z are NOT integer variables! They are addresses (or pointers).

5.2 Examples (4/4)

Example4.c

```
#include <stdio.h>
void f(int *, int *, int *);

int main(void) {
    int a = 9, b = -2, c = 5;
    f(&a, &b, &c);
    printf("a = %d, b = %d, c = %d\n", a, b, c);
    return 0;
}

void f(int *x, int *y, int *z)
{
    *x = 3 + *y;
    *y = 10 * *x;
    *z = *x + *y + *z;
    printf("x = %p, y = %p, z = %p\n", x, y, z);
}
```

Use %p for pointers.

Addresses of variables a, b and c.
(Values change from run to run.)

```
x = ffbff78c, y = ffbff788, z = ffbff784
a = 1, b = 10, c = 16
```

End of File