Pointers

Outline

1. Pointers in C language
   - What is a pointer?
   - Pointer syntax

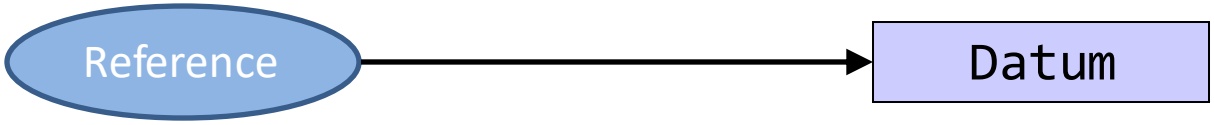
2. Passing data by reference via Pointers
   - How to pass multiple data from a function to its caller via parameters?
   - How to implement a function to swap the value of two variables?

3. Additional pointer concepts

1.1. What is a Reference?

- In programming, a *reference* is a value that refers to a variable or a datum in the memory. It enables a program to indirectly access a particular variable/datum.

- *Dereferencing* a reference ➔ Accessing a variable/datum through the reference



1.2. What is a Pointer?

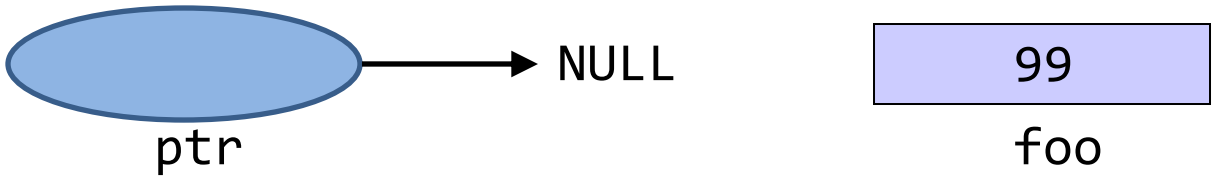
- A *pointer* in C language is a kind of reference.
- A pointer == A memory address
- A *pointer variable* is a variable that stores memory address.
- NULL is used when the pointer points to nothing.

1.3. Pointer Syntax

```
1   int foo = 99;
2   int *ptr = NULL;
3
4
5
6
7
8
9
10
11
```

Declare a pointer variable named `ptr`.

Initialize `ptr` to point to nothing (`NULL`).

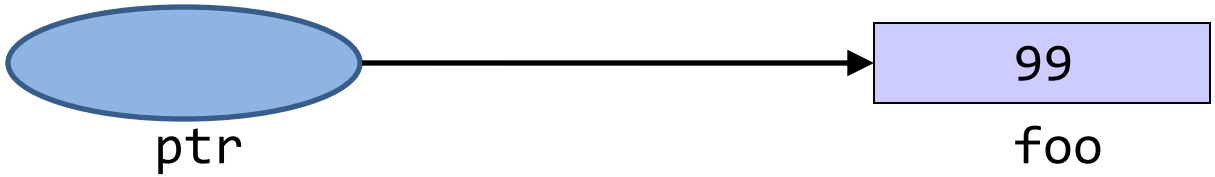


1.3. Pointer Syntax

```
1   int foo = 99;
2   int *ptr = NULL;
3
4   ptr = &foo;
5
6
7
8
9
10
11
```

& is the *address-of* operator, which yields the memory address of a variable.

In this example, the address of `foo` is assigned to `ptr`. The effect of this is like "making `ptr` points to `foo`".



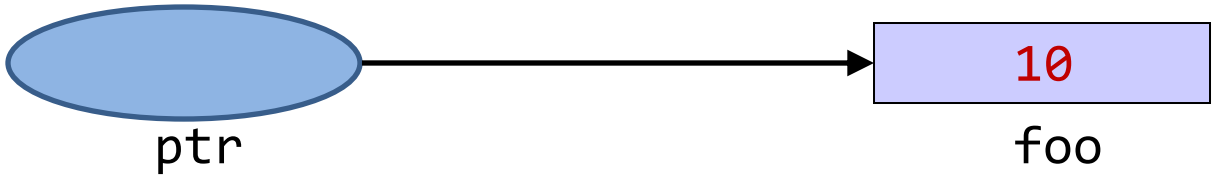
1.3. Pointer Syntax

```
1   int foo = 99;
2   int *ptr = NULL;
3
4   ptr = &foo;
5
6   *ptr = 10;
7
8
9
10
11
```

The unary operator * is a *dereference* operator.

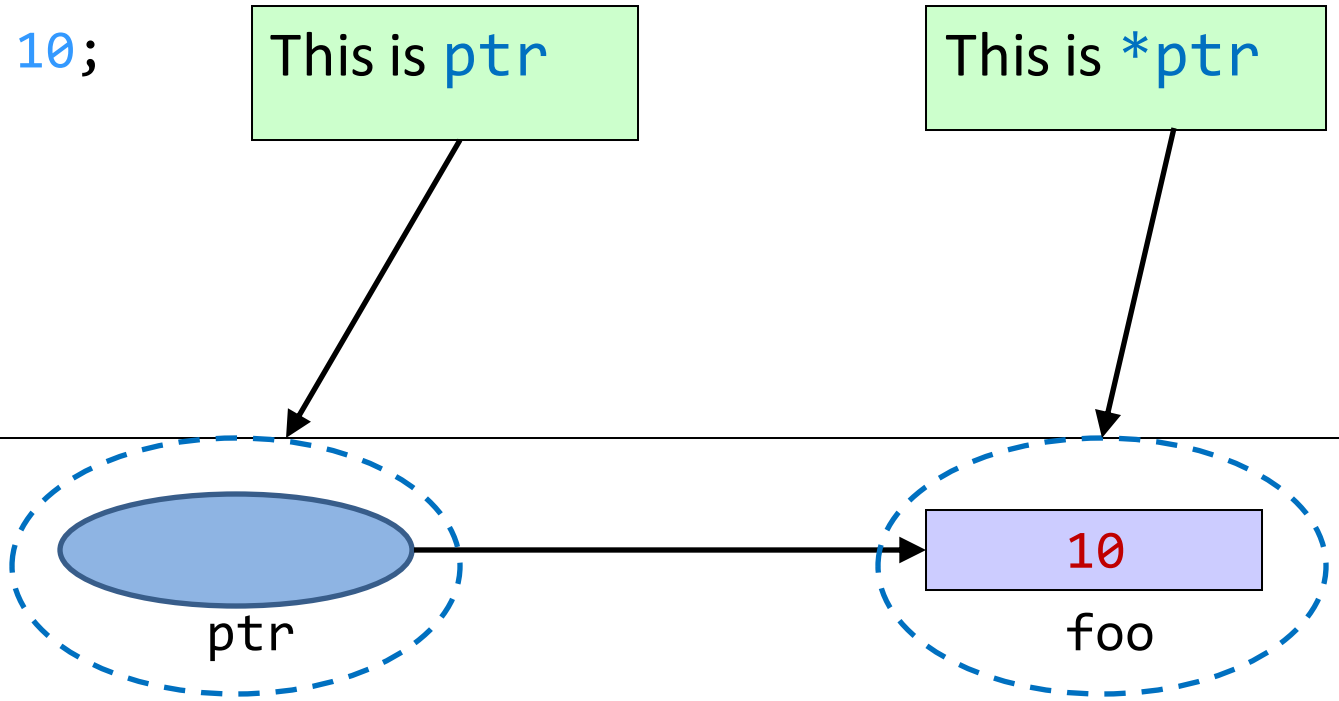
`ptr` holds the address of `foo`,
so `*ptr` (dereferencing the address) means "accessing `foo`"

In this example, 10 is assigned to `foo`.



1.3. Pointer Syntax

```
1   int foo = 99;
2   int *ptr = NULL;
3
4   ptr = &foo;
5
6   *ptr = 10;
7
8
9
10
11
```



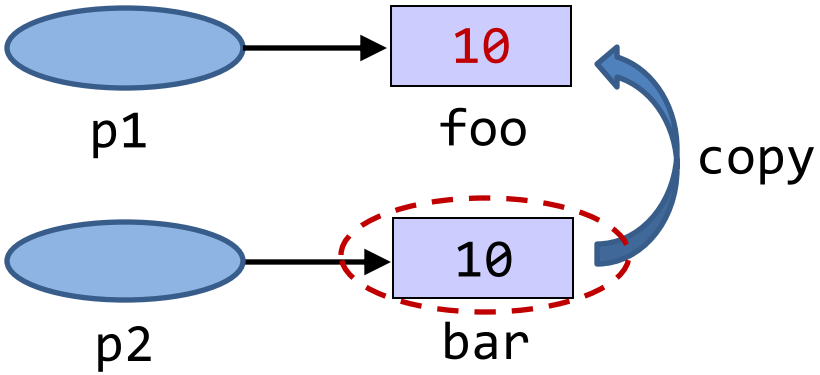
1.4. Copying datum at an address vs. copying address

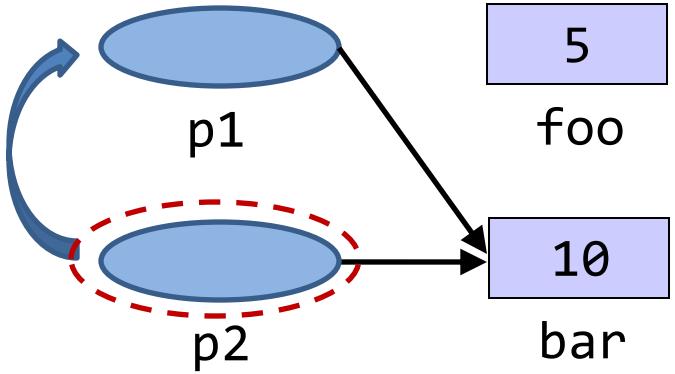
```
1  int foo = 5, bar = 10;
2  int *p1, *p2;
3
4  p1 = &foo;
5  p2 = &bar;
6  // Copying the value at an
7  // address
8  *p1 = *p2;
```

```
int foo = 5, bar = 10;
int *p1, *p2;

p1 = &foo;
p2 = &bar;

// Copying an address
p1 = p2;
```





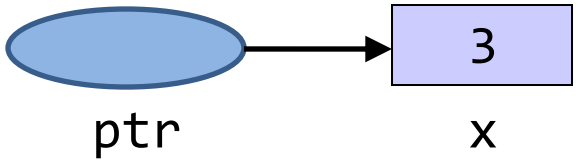
2. Pointers as Parameters

- Pointers are passed by value
  - The address stored in one pointer variable (the actual parameter) is copied to another pointer variable (the formal parameter).

- Passing pointers allow us to emulate the effect of "pass by reference".
  - When the callee receives the memory address, the callee can access the data stored at that memory address.

2.1. Pointers as Parameters

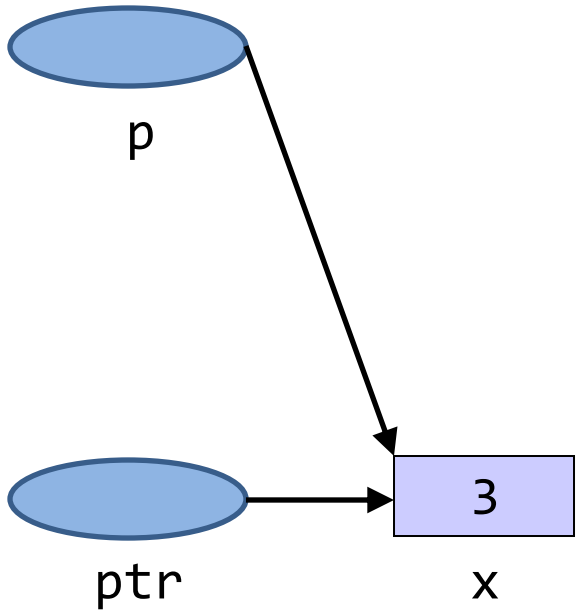
```
1   void foo(int *p) {
2     *p = 0;
3   }
4
5   int main(void) {
6     int x = 3, *ptr = &x;
7
8     foo(ptr);
9     printf("%d", x);  // Print 0
10
11    return 0;
12  }
```



(Line 6) Initially, **ptr** holds the address of **x**.

2.1. Pointers as Parameters

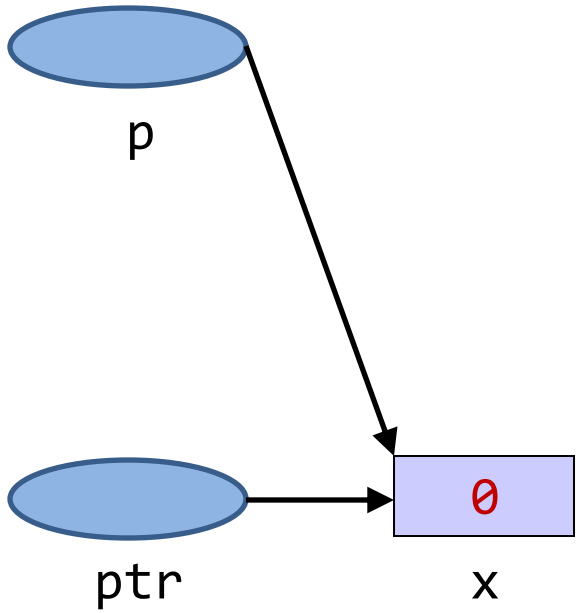
```
1   void foo(int *p) {
2     *p = 0;
3   }
4
5   int main(void) {
6     int x = 3, *ptr = &x;
7
8     foo(ptr);
9     printf("%d", x);  // Print 0
10
11    return 0;
12  }
```



(Line 8) During the function call, value of **ptr** (address of **x**) is copied to **p**. In effect, **p** points to **x.**

2.1. Pointers as Parameters

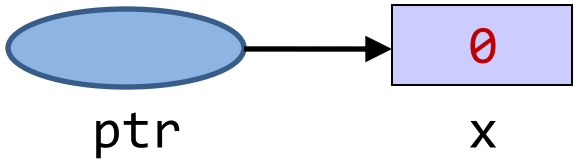
```
1   void foo(int *p) {
2     *p = 0;
3   }
4
5   int main(void) {
6     int x = 3, *ptr = &x;
7
8     foo(ptr);
9     printf("%d", x);  // Print 0
10
11    return 0;
12  }
```



(Line 2) In the function **foo()**, since **p** is pointing at **x**, ***p** is equivalent to **x**. As a result, even though **foo()** cannot access **x** in **main()** directly, it is able to modify **x** through the pointer **p**.

2.1. Pointers as Parameters

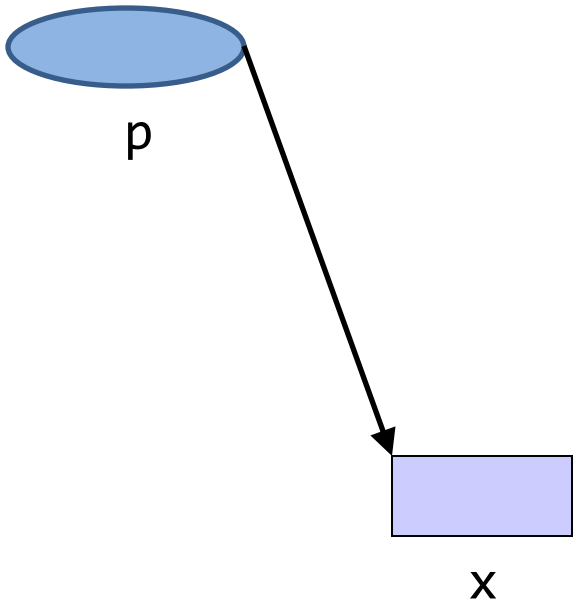
```
1   void foo(int *p) {
2     *p = 0;
3   }
4
5   int main(void) {
6     int x = 3, *ptr = &x;
7
8     foo(ptr);
9     printf("%d", x);  // Print 0
10
11    return 0;
12  }
```



(Line 9) When **`foo()`** finishes and execution returns to **`main()`**, the value of **`x`** in **`main()`** has already been changed to **`0`**.

2.1. Pointers as Parameters

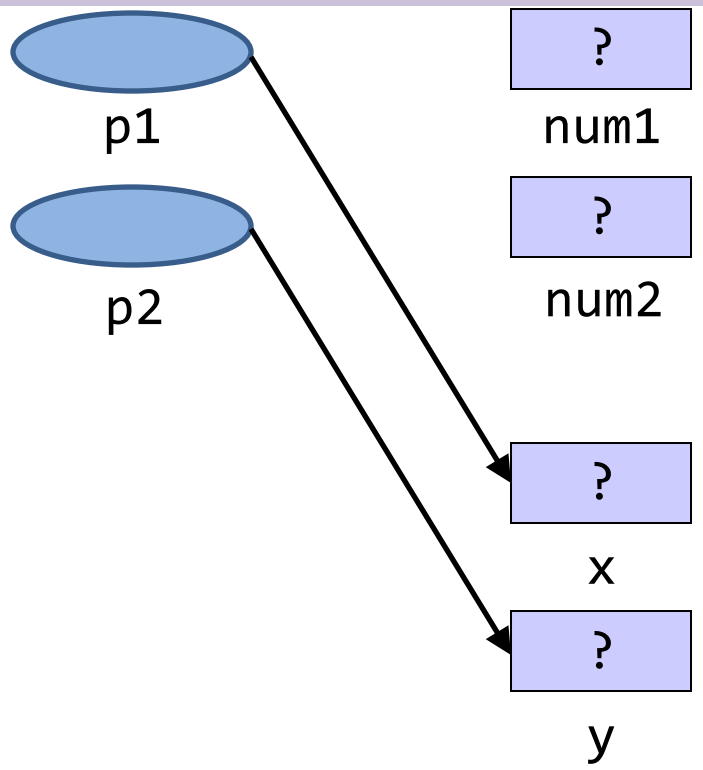
```
1   void foo(int *p) {
2     *p = 0;
3   }
4
5   int main(void) {
6     int x = 3;
7
8     foo(&x);
9     printf("%d", x);  // Print 0
10
11    return 0;
12  }
```



We can also pass the address of a variable directly (instead of first assigning the address to a pointer variable) to a function that accepts an address as its parameter.

2.2. Passing multiple values to a caller via parameters

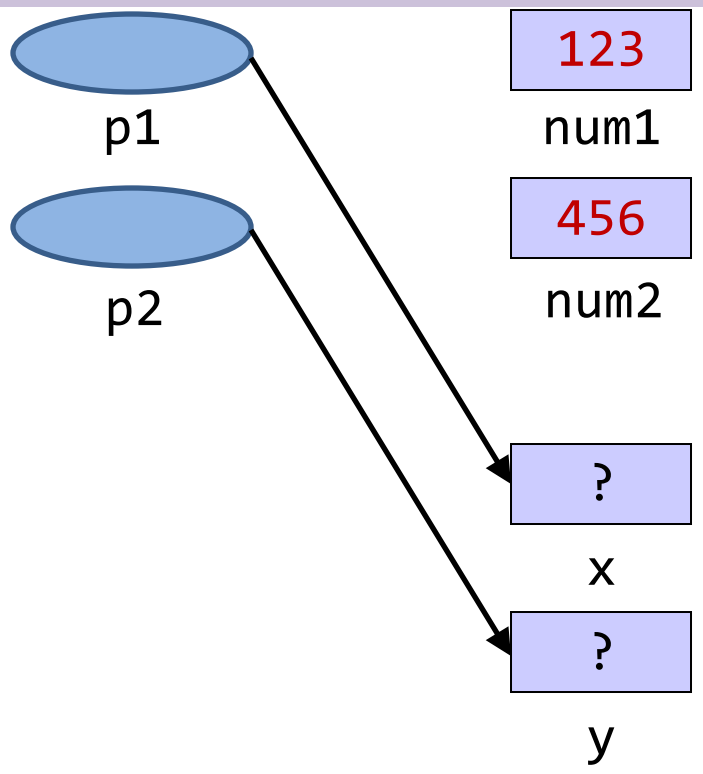
```
1  void readTwoInt(int *p1, int *p2) {
2      int num1, num2;
3      scanf("%d%d", &num1, &num2);
4      *p1 = num1;
5      *p2 = num2;
6  }
7
8  int main(void) {
9    int x, y;
10   readTwoInt(&x, &y);
11   return 0;
12 }
```



At the start of the function call, **p1** and **p2** point to **x** and **y** respectively.

2.2. Passing multiple values to a caller via parameters

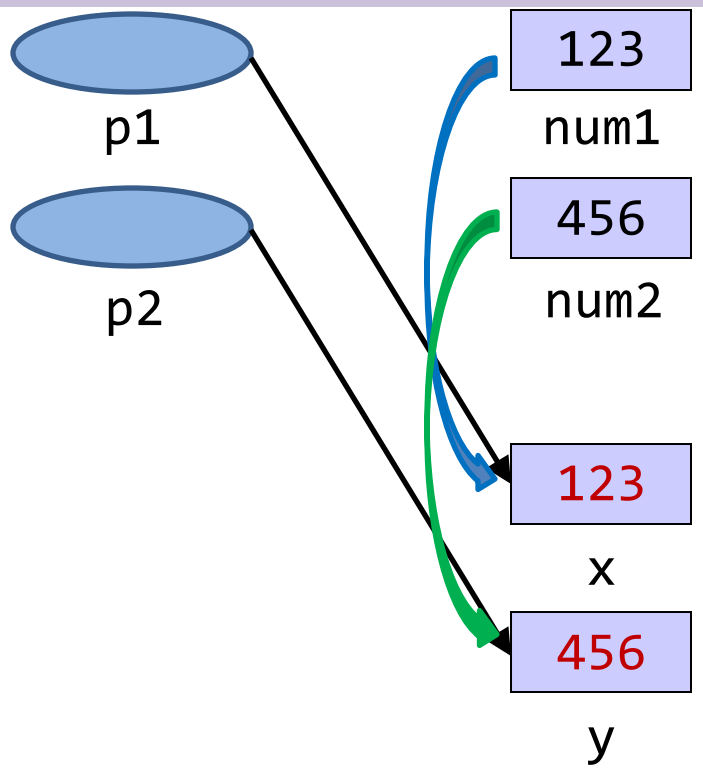
```
1  void readTwoInt(int *p1, int *p2) {
2      int num1, num2;
3      scanf("%d%d", &num1, &num2);
4      *p1 = num1;
5      *p2 = num2;
6  }
7
8  int main(void) {
9    int x, y;
10   readTwoInt(&x, &y);
11   return 0;
12 }
```



Suppose the input values are 123 and 456.
Through the addresses of **num1** and **num2**, **scanf()** is able to "dereference the addresses" and store the input values into **num1** and **num2**.

2.2. Passing multiple values to a caller via parameters

```
1   void readTwoInt(int *p1, int *p2) {
2       int num1, num2;
3       scanf("%d%d", &num1, &num2);
4       *p1 = num1;
5       *p2 = num2;
6   }
7
8   int main(void) {
9     int x, y;
10    readTwoInt(&x, &y);
11    return 0;
12  }
```



Through **p1** and **p2**, **readTwoInt()** is able to copy the values from **num1** and **num2** to **x** and **y**, thus achieving the effect of passing two integers back to **main()**.

2.2. Passing multiple values to a caller via parameters

```
1   void readTwoInt(int *p1, int *p2) {
2       scanf("%d%d", p1, p2);
3   }
4
5   int main(void) {
6     int x, y;
7     readTwoInt(&x, &y);
8     return 0;
9   }
10
11
12
```

Since **p1** and **p2** are storing the addresses of **x** and **y**, we can pass the addresses to **scanf()** directly (i.e., without **&**). This way, **scanf()** will store the input directly into **x** and **y**.

2.3. Swapping the value of two variables

```c
// Version 1
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

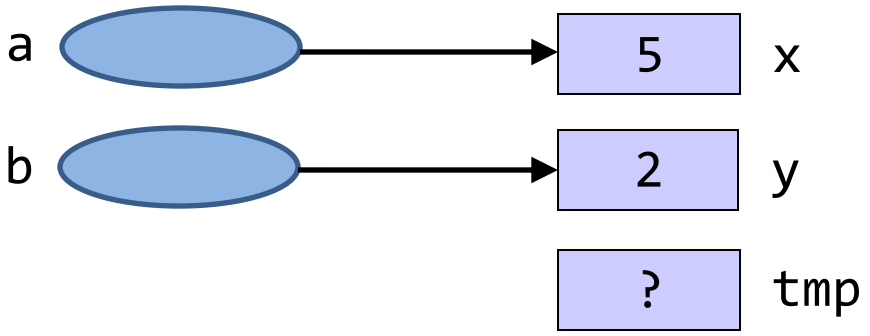
```c
// Version 2
void swap(int *a, int *b) {
  int *tmp = a;
  a = b;
  b = tmp;
}
```

```c
int main(void) {
  int x = 5, y = 2;
  swap(&x, &y);
  return 0;
}
```

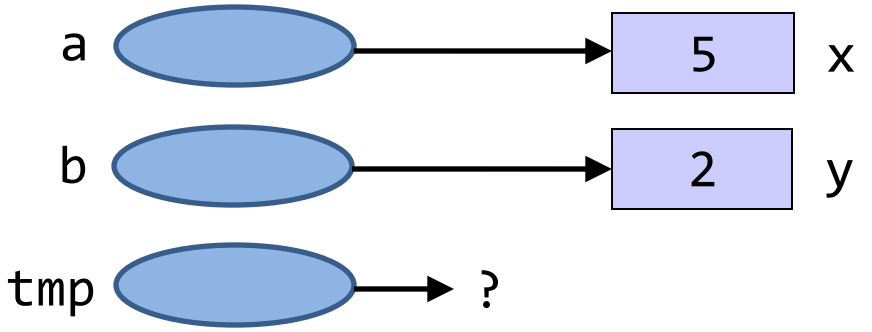
Which version of **`swap()`** would correctly swap the value of **`x`** and **`y`** in **`main()`**?

2.3. Swapping the value of two variables

```
// Version 1
void swap(int *a, int *b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```



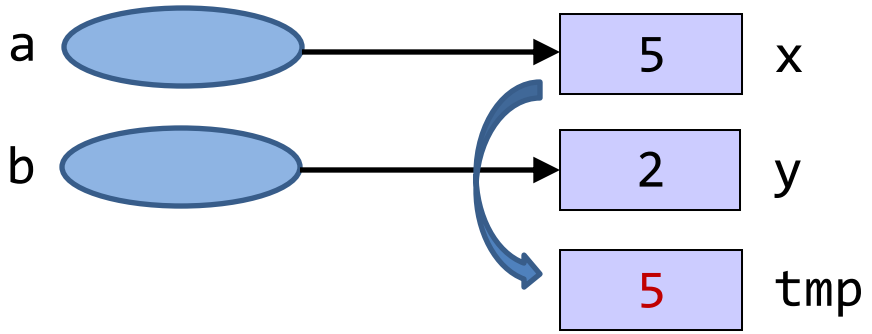

```
// Version 2
void swap(int *a, int *b) {
   int *tmp = a;
   a = b;
   b = tmp;
}
```



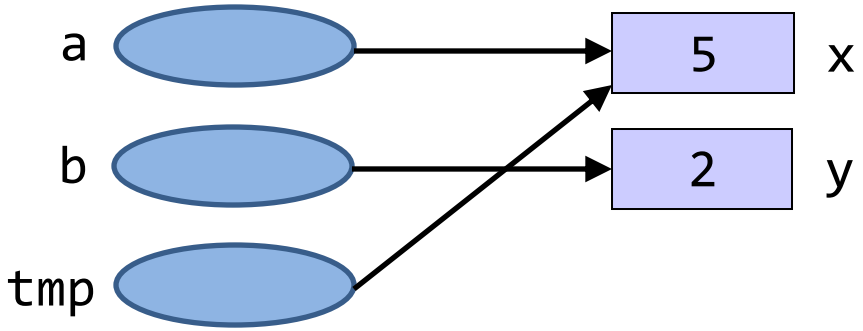
At the start of the function call…

2.3. Swapping the value of two variables

```
// Version 1
void swap(int *a, int *b) {
  int tmp = *a;
  *a = *b;
  *b = tmp;
}
```

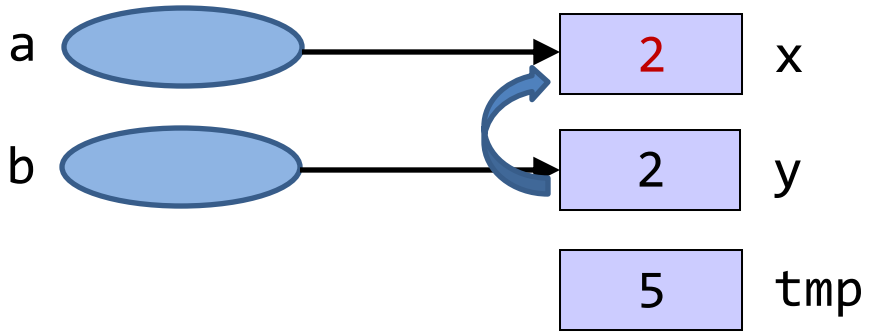



```
// Version 2
void swap(int *a, int *b) {
  int *tmp = a;
  a = b;
  b = tmp;
}
```

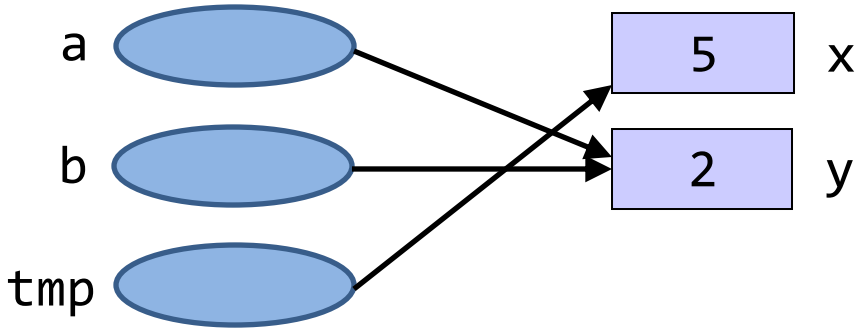


2.3. Swapping the value of two variables

```
// Version 1
void swap(int *a, int *b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

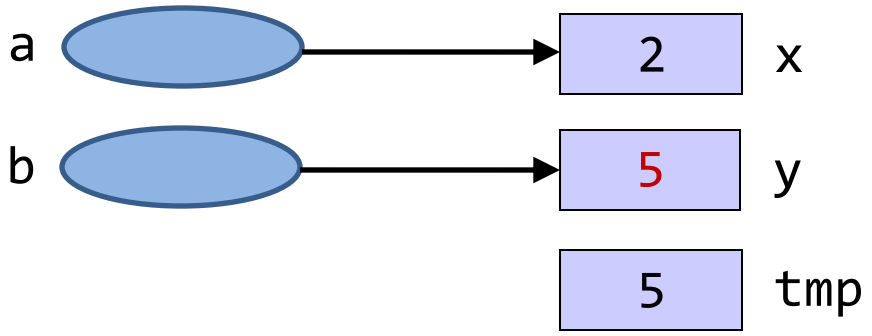



```
// Version 2
void swap(int *a, int *b) {
   int *tmp = a;
   a = b;
   b = tmp;
}
```

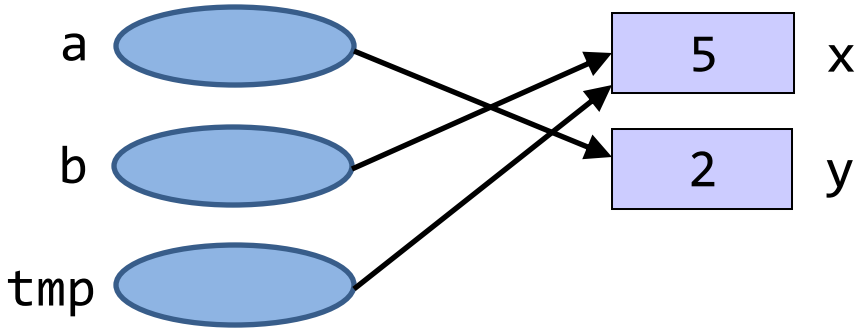


2.3. Swapping the value of two variables

```
// Version 1
void swap(int *a, int *b) {
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```




```
// Version 2
void swap(int *a, int *b) {
   int *tmp = a;
   a = b;
   b = tmp;
}
```



At the end of the function call, version 1 swaps the value between **x** and **y**. Version 2 only swaps the pointers within **`swap()`**; it leaves **x** and **y** unchanged.

3. Additional Pointer Concepts

- Differentiate * in declaration and expression
- Pointers Types
- Pitfalls
- Value of a Pointer

3.1. Differentiate * in declaration and expression

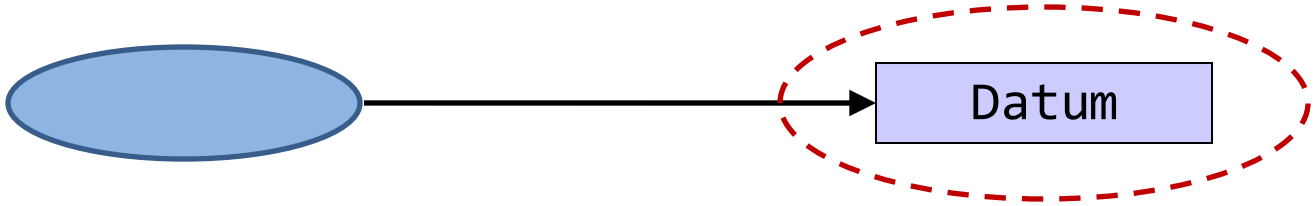
- In a variable declaration, * is used to declare that a variable is a pointer variable.

```
int foo;
int *ptr;
```

- In an expression, * acts as the dereference operator.

```
ptr = &foo;
*ptr = 999;
printf("%d", *ptr);
```

3.2. Pointers Types



How does a program know what type of datum this is?

```
int  *iptr;
char *cptr;
```

- The type of a pointer variable tells a program how to treat the datum when the pointer is dereferenced.

3.2. Pointers Types

Updated

```
int *iptr, intVar;
char *cptr, charVar;

iptr = &intVar;       // OK
cptr = &charVar;      // OK

cptr = &intVar;       // Warning: cptr should not store
                      // the address of an int variable.

cptr = iptr;          // Warning: cptr should not store
                      // the address of an int variable.
```

- Pointers of different types do not mix well.

3.3. Pitfalls

```
int *p, foo = 10;
*p = 10;
```

p is not initialized, that means it can point to any memory location. Modifying the content at an "unauthorized" location is dangerous and will likely cause the program to crash.

```
int *p, foo = 10;
…
*p = &foo;
```

Compile-time error (incompatible types).

The type of the left operand, **`*p`**, is `int`.
The type of the right operand, **`&foo`**, is an address.

3.4. Value of a Pointer (Memory Address)

```
int *p, foo = 100;

p = &foo;
printf("%d\n", *p);  // This prints 100
printf("%d\n", p);   // This prints the address of foo as
                     // a decimal number.
```

- Memory address is just an integer.

- You can print the address but the address value is system dependent and does not carry much information.

Summary

- Understand what a pointer is
- Understand the pointer syntax
- Know how to implement a function with pointers as parameters (to allow the function to pass data back to its caller via parameters)

Reading Assignment

- C: How to Program, 8th ed, Deitel and Deitel
- Chapter 7 C Pointers
  - Sections 7.1 – 7.3: Pointer Basics and Pointer Operators
  - Section 7.4: Passing Arguments to Functions by (Address)