Pointers

Famous feature in C: traditionally good for performance but highly error-prone (exam included)

Outline

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

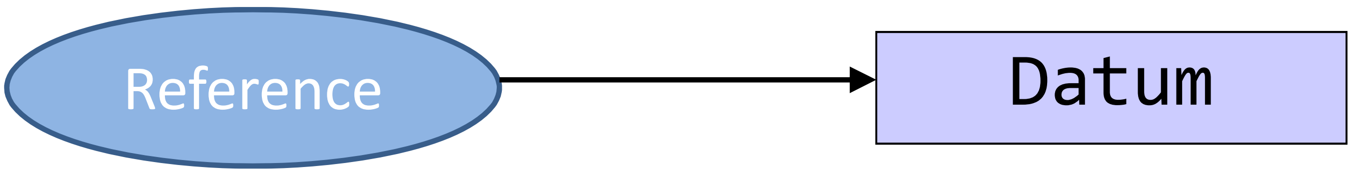
1.1. Concept: What is a Reference?

Two basic concepts:

- In programming, a *reference* allows us to **refer to a variable or a datum currently in memory**. It enables a program to indirectly access a particular variable/datum by using its reference.

- *How? By means of dereferencing* a reference:

  ➔ Access a variable/datum through the reference



1.2. What is a Pointer?

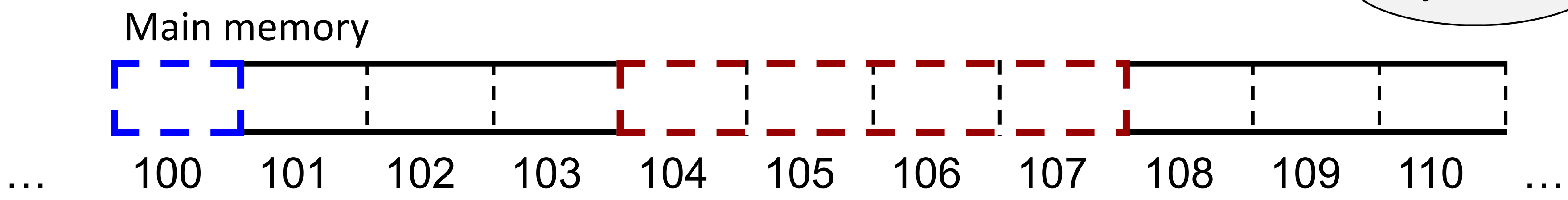
- A *pointer* in C language is one kind of reference.
- A *pointer variable* is a **variable that stores memory address**, i.e., the value stored is a memory address.
- Since it is a variable, it also has its own memory…

- Only *five basic syntax rules* to remember, but…

WARNING: don’t mix up with reference variables in C++; it is another way to create references. Although its effect is similar, it has a different mechanism & syntax from pointers!

Syntax 1: The “address-of” operator – &

Syntax 1




```
char ch ; // ch occupies 1 byte  (assume: location 100)
int  x  ; // x  occupies 4 bytes (assume: location 104)

// Print address of x as a hexadecimal number.
printf( "%p\n" , &x );    // %p – print its hex. value

// Print addresses of ch and x as decimal numbers.
printf( "%u %u\n" , (unsigned int) &ch ,
                    (unsigned int) &x  );
```

- **&**, when applied to a variable, yields the address of the variable.

```
00000068
100 104
```

Note: 6 x 16 + 8 = 104

Syntax 2: Creating Pointer Variables

- *Pointer variables* are variables that store memory addresses (or pointers).

```
int y ;
y = 5 ;

int * ptr ;  // Declare a pointer variable
ptr = & y ;  // Assign address of y to ptr
```

Syntax 2

The data type of `ptr` is `int *`.

- **`ptr`** is a variable that holds the address of an **`int`** variable.

Assigning Address of Variables to Pointers

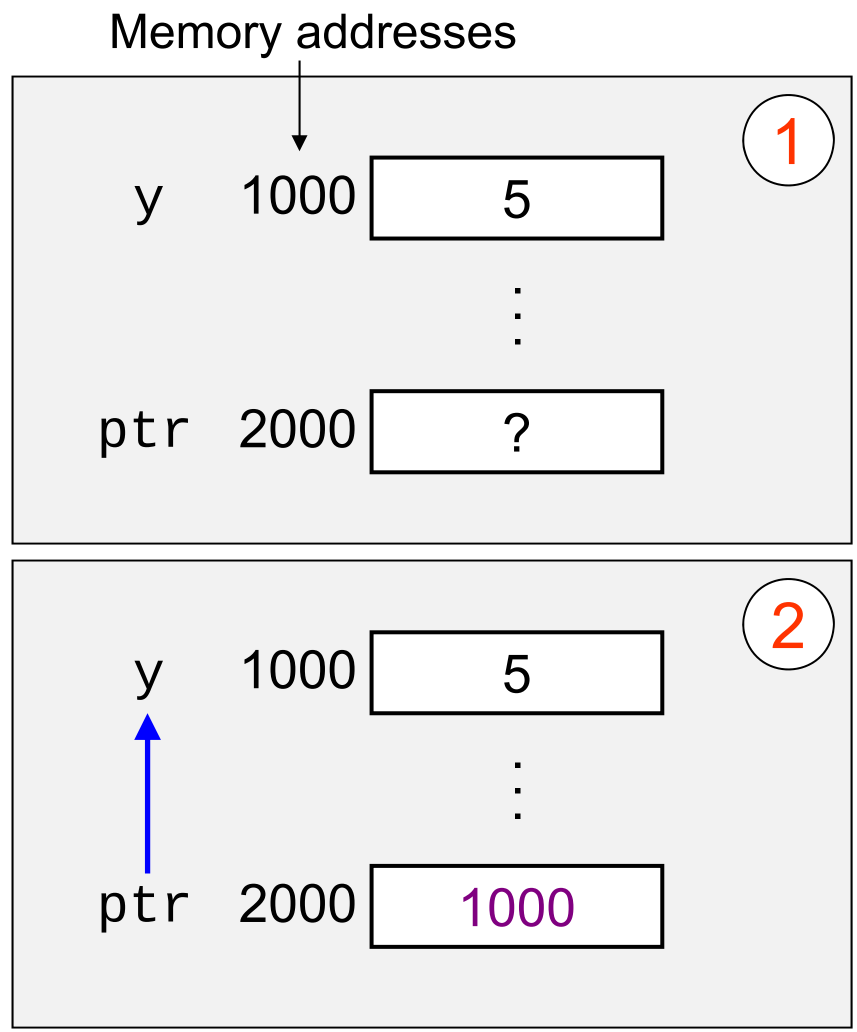
```
int y = 5;
int *ptr;    1
```

```
ptr = &y;    2
```

ptr stores the address of variable y.

or

ptr points to y.



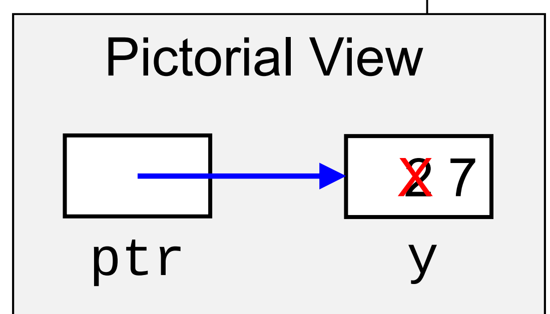
Syntax 3: The Dereference Operator – *

- *, when applied to a pointer variable in an expression, yields the variable that is pointed to by the pointer.

```
int y = 2;
int *ptr;

ptr = &y;    // ptr gets the address of y

// Prints the address of y in hexadecimal
printf( "%p" , ptr );

// Same as print out y;
printf( "%d" , *ptr );

// Same as y = 3 * y + 1;
*ptr = 3 * *ptr + 1;
```

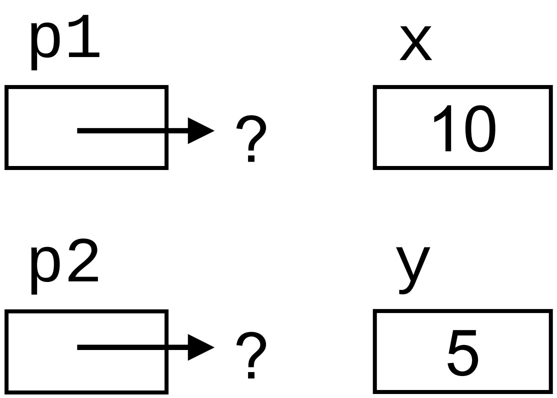
When used as a prefix in variable declaration, * indicates that the variable is a pointer

Syntax 3



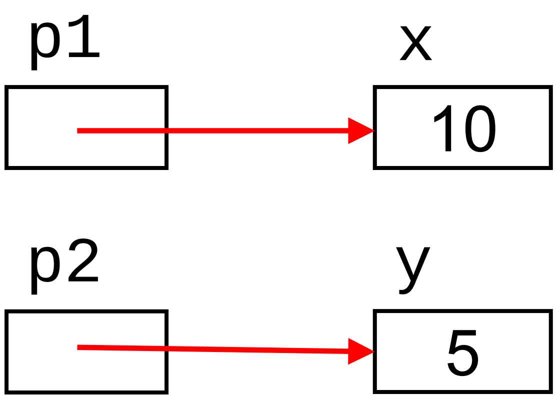
Example: Pointers to Access Variables

```
int x = 10 , y = 5 ;
int *p1 , *p2 ;
```



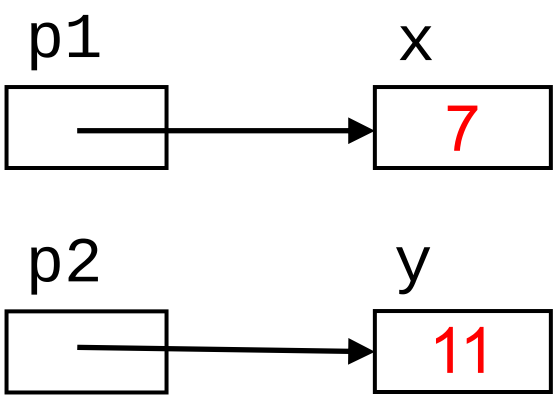
Example: Pointers to Access Variables

```
int x = 10 , y = 5 ;
int *p1 , *p2 ;

p1 = &x ;    // Set up p1 to point to x
p2 = &y ;    // Set up p2 to point to y
```



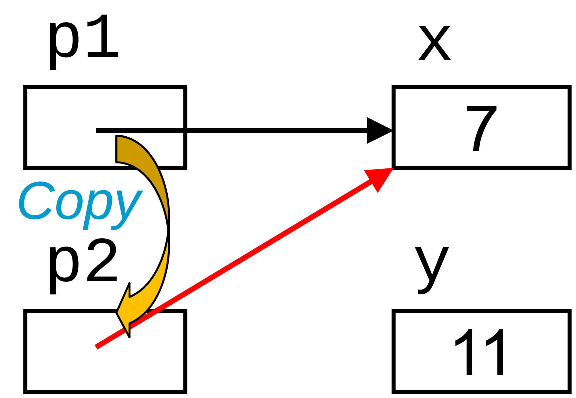
Example: Pointers to Access Variables

```
int x = 10 , y = 5 ;
int *p1 , *p2 ;

p1 = &x ;    // Set up p1 to point to x
p2 = &y ;    // Set up p2 to point to y

*p1 = 7  ;
*p2 = 11 ;
```



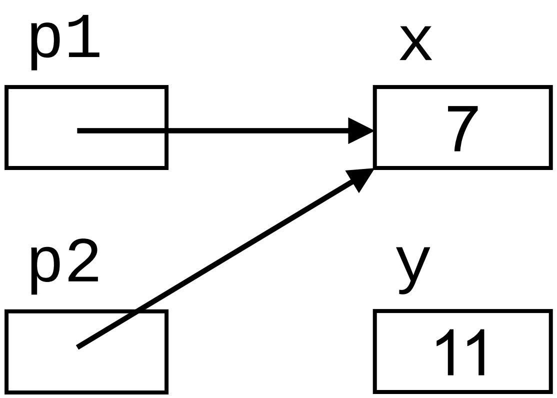
Example: Pointers to Access Variables

```
int x = 10 , y = 5 ;
int *p1 , *p2 ;

p1 = &x ;    // Set up p1 to point to x
p2 = &y ;    // Set up p2 to point to y

*p1 = 7  ;
*p2 = 11 ;

p2 = p1 ;  // Not the same as *p2 = *p1
           // Pointer assignment!!!
```



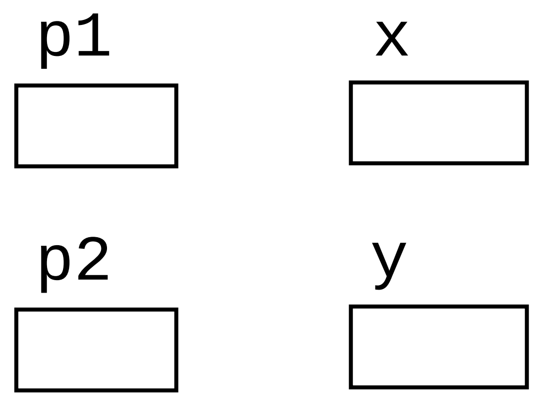
Example: Pointers to Access Variables

```
int x = 10 , y = 5 ;
int *p1 , *p2 ;

p1 = &x ;   // Set up p1 to point to x
p2 = &y ;   // Set up p2 to point to y

*p1 = 7  ;
*p2 = 11 ;

p2 = p1 ;  // Not the same as *p2 = *p1
           // Pointer assignment!!!

printf( "%d %d\n" , *p1 , *p2 );
           // What's the output?
```



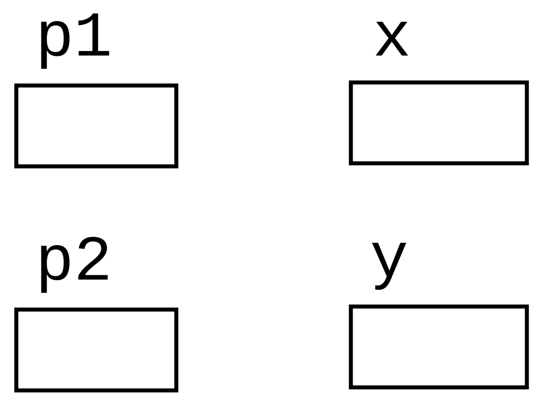
Exercise 1

```
int  x = 7 , y = 11 ;
int *p1 , *p2 ;

p1 = &x ;
p2 = &y ;

*p1 = 3 * *p2 + 2 ;
 y  = 10 ;
printf( "%d\n" , *p2 );
(*p1)++;
printf( "%d %d\n" , x , y );
```



Exercise 2

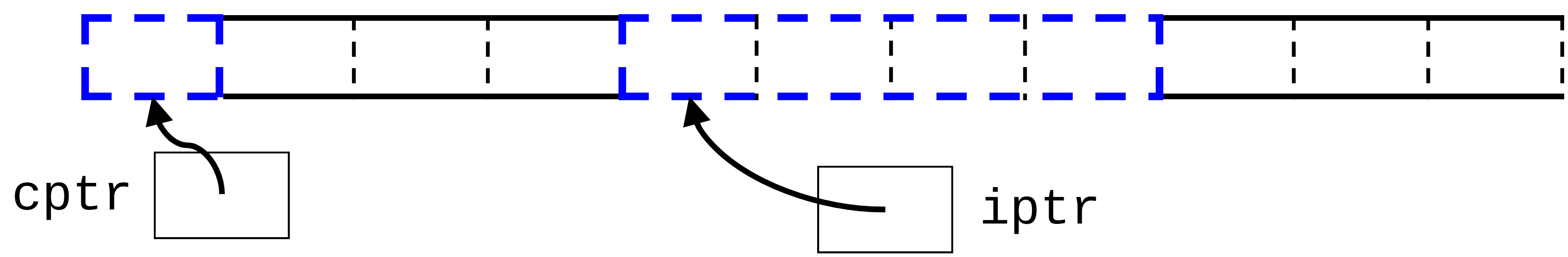
```
int  x = 7 , y = 11 ;
int *p1 , *p2 ;

p1 = &x ;
p2 = &y ;

*p1 = y ;
*p2 = x ;
printf( "%d %d\n" , x , y );
printf( "%d %d\n" , *p1 , *p2 );
```



Syntax 4: Pointers to Different Data Types

```
int    *iptr ;    // Pointer to integer
char   *cptr ;    // Pointer to char
double *dptr ;    // Pointer to double
```

- Pointer variables of different types all store memory addresses.
- The pointer type tells the computer the range of memory of the data stored at the location pointed to by the pointer.



Treat 1 byte here as character

Starting here, treat the next 4 bytes as integer

Syntax 4: Pointers to Different Data Types

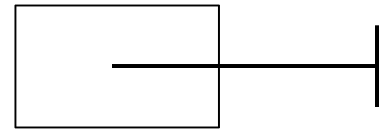
- Pointers of different types are <u>incompatible</u>
- But… explicitly converted by type casting

```
int    *iptr ;    // Pointer to integer
char   *cptr ;    // Pointer to char
double *dptr ;    // Pointer to double
…
iptr = cptr ;     // Compilation error
cptr = iptr ;     // Compilation error
…
cptr = (char *) iptr ;  // Explicitly converted
// No warning, but usually doesn't make sense
```

Syntax 5: “NULL” pointer

```
int y , *ptr1 ;
double *ptr2 ;
ptr1 = &y    ;   // ptr1 points to y
ptr1 = NULL ;   // ptr1 stores address 0
ptr2 = NULL ;   // ptr2 stores address 0
```

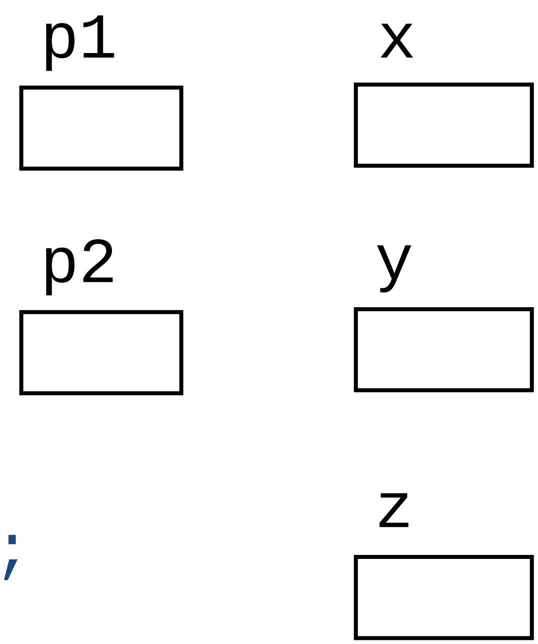
ptr2



- **NULL** is a predefined constant representing memory address 0 ... So, don’t do “`*ptr1 = 2;`”
- **NULL** indicates that the pointer is "*not pointing to anything*." (No data can be stored at location 0.)
- **NULL** is compatible to any pointer type.

Exercise 3

```
int x = 7, y = 11, z = 3, *p1, *p2;
p2 = &x;
p2 = &y;

*p2 = 5  ;
 p1 = p2 ;
 p2 = &z ;
 y  = 6  ;
printf( "%d %d\n" , *p1 , *p2 );

 z  = *p1 ;
*p2 =  x  ;
printf( "%d %d\n" , x , y );
```



Outline

~~1. Pointers in C language~~

- ~~What is a pointer?~~
- ~~Pointer syntax~~

2. Applications: 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

2. Pointers as Function Parameters

- In C, function parameter is **always passed by value**

Concept: Pass by value

```c
void foo( int val )
{
  val = 0 ;
}

int main( void )
{
  int x = 3 ;

  foo( x );
  printf( "%d" , x );  // print what?
  return 0 ;
}

```

Concept: Pass by reference

```
void foo( int & val )
{
  val = 0 ;
}

int main( void )
{
  int x = 3 ;

  foo( x );
  printf( "%d" , x );  // print what?
  return 0 ;
}
```

This is C++ !!!

If you pass a variable into a function such that the local variable inside the function is a reference of your variable. That means the function can modify the variable outside!!!

BUT... C language doesn't have this!
C only has "pass by value"

Beyond this course on C:
https://www.tutorialspoint.com/cplusplus/cpp_function_call_by_reference.htm

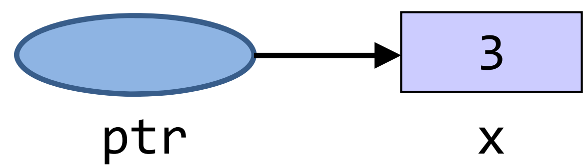
2. Pointers as Function Parameters

- In C, function parameter is **always passed by value**
- A pointer itself is a variable, so passing by value means?
  - It means that the address stored in a pointer variable (the actual parameter) is copied to another pointer variable, as its contents (the formal parameter).
- Passing pointers to a function **emulates** the effect of "pass by reference" (since it is a memory address)
  - When the callee receives the memory address, it can access (read & modify) the data stored at that memory address

Again, C only supports "pass by value" but not "pass by reference". Don't mix up with "pass by reference" in C++ (which is based on the reference variables).

2.1. Pointers as Param. (Emulate pass by ref.)

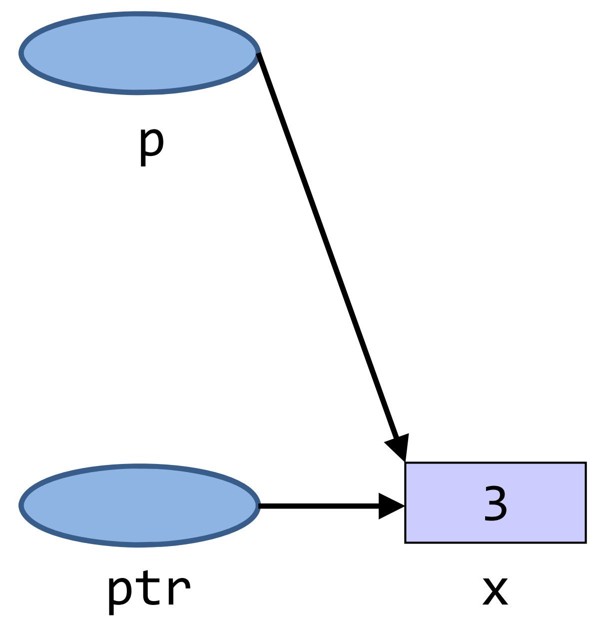
```c
1  void foo( int * p )
2  {
3    *p = 0 ;
4  }
5
6  int main( void )
7  {
8    int   x   =   3 ;
9    int * ptr = & x ;
10
11   foo( ptr );
12   printf( "%d" , x );  // print 0
13   return 0 ;
14 }
```



(Line 9) Initially, **ptr** holds the address of **x**.

2.1. Pointers as Param. (Emulate pass by ref.)

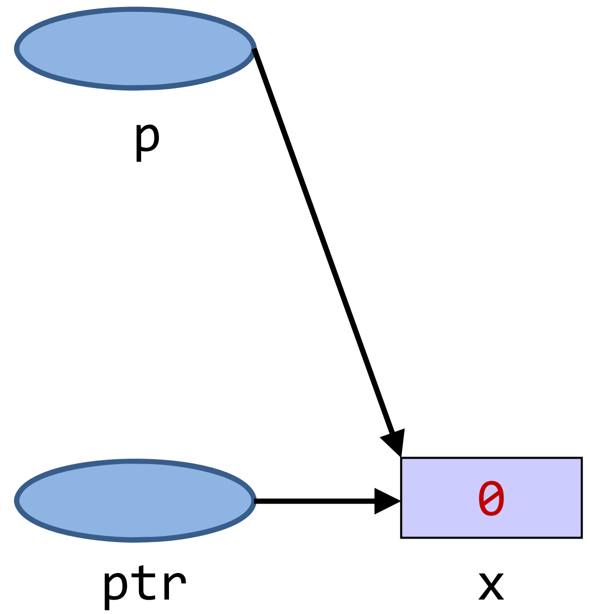
```c
1   void foo( int * p )
2   {
3     *p = 0 ;
4   }
5
6   int main( void )
7   {
8     int    x    =    3 ;
9     int * ptr = & x ;
10
11    foo( ptr );
12    printf( "%d" , x );  // print 0
13    return 0 ;
14  }
```



(Line 11) During the function call, the value of **`ptr`** (address of **`x`**) is copied to variable **p** inside function **`foo`**. In effect, **p** points to **`x`.**

2.1. Pointers as Param. (Emulate pass by ref.)

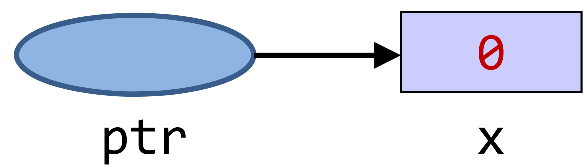
```
1  void foo( int * p )
2  {
3    *p = 0 ;
4  }
5
6  int main( void )
7  {
8    int   x   =   3 ;
9    int * ptr = & x ;
10
11   foo( ptr );
12   printf( "%d" , x );  // print 0
13   return 0 ;
14 }
```



(Line 3) Inside the function call, since **p** points to **x**, ***p** is just **x**. Hence, even though **foo()** cannot directly access **x** in **main()**, it can modify **x** through **p**.

2.1. Pointers as Param. (Emulate pass by ref.)

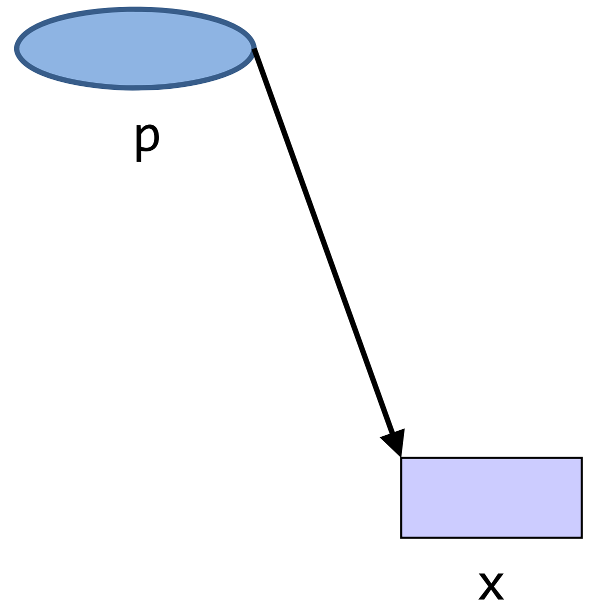
```
1  void foo( int * p )
2  {
3    *p = 0 ;
4  }
5
6  int main( void )
7  {
8    int   x   =   3 ;
9    int * ptr = & x ;
10
11   foo( ptr );
12   printf( "%d" , x );  // print 0
13   return 0 ;
14 }
```



(Line 12) After **`foo()`** finishes and the execution returns back to **`main()`**, the value of **`x`** in **`main()`** has already been changed to **`0`**.

2.1. Pointers as Param. (Emulate pass by ref.)

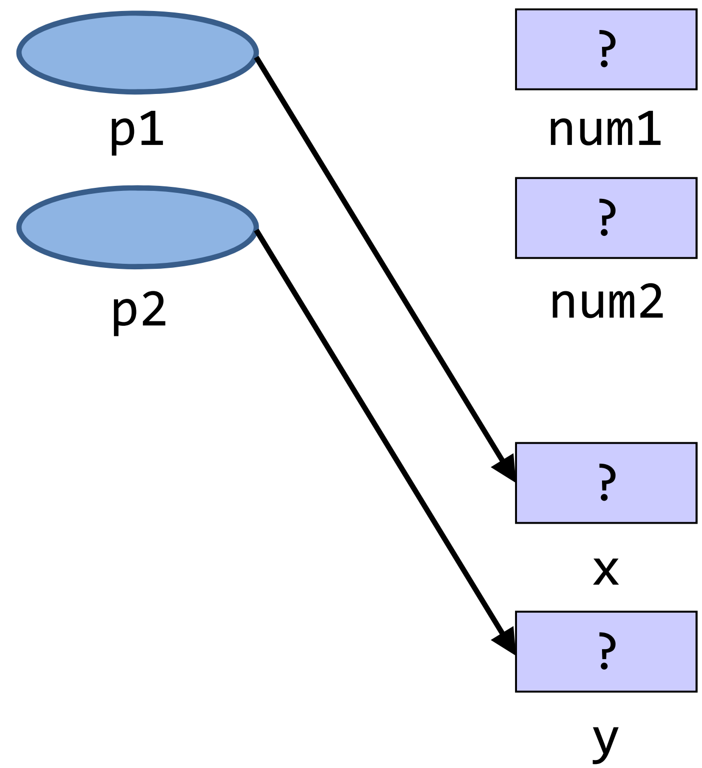
```c
1  void foo( int * p )
2  {
3    *p = 0 ;
4  }
5
6  int main( void )
7  {
8    int   x   =   3 ;
9    //int * ptr = & x ;
10
11   foo( & x );
12   printf( "%d" , x );  // print 0
13   return 0 ;
14 }
```



(Line 11) We can also pass the address of a variable directly (instead of first assigning its memory 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

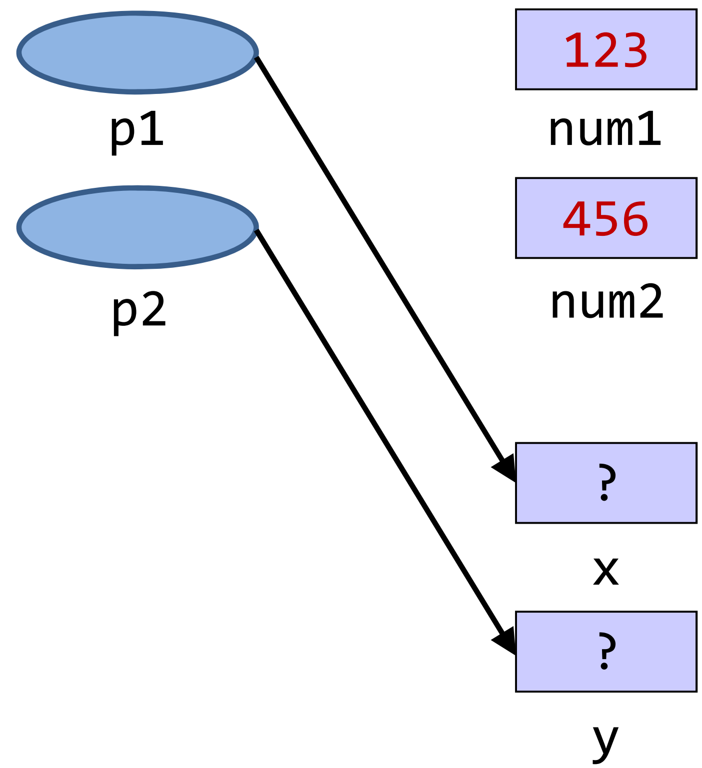
```c
void readTwoInt( int * p1 , int * p2 )
{
    int num1 , num2 ;
    scanf( "%d%d" , & num1 , & num2 );
    *p1 = num1 ;
    *p2 = num2 ;
}
int main( void )
{
    int x , y ;
    readTwoInt( & x , & y );
    return 0 ;
}
```



When the function call starts, **p1** and **p2** point to **x** and **y**, respectively.

2.2. Passing multiple values to a caller via parameters

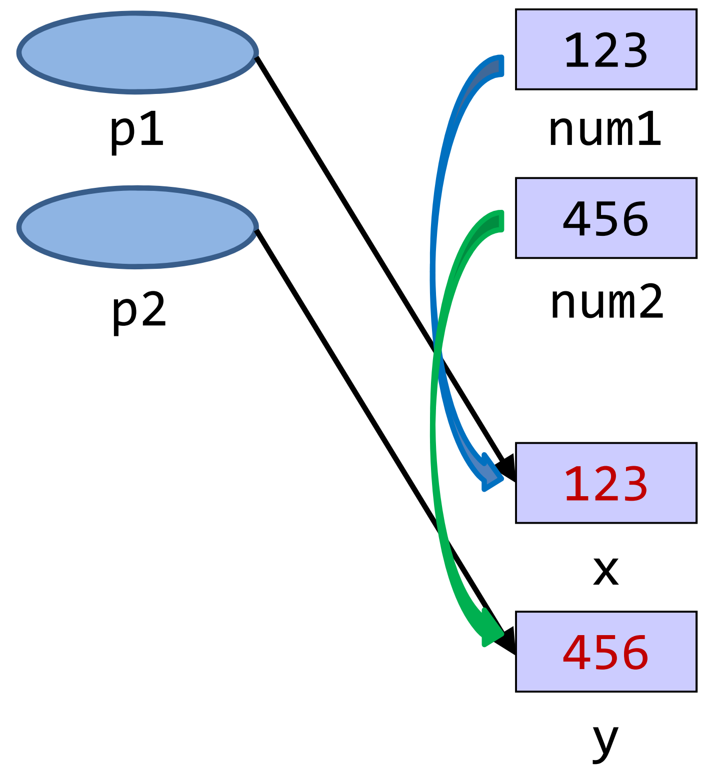
```c
void readTwoInt( int * p1 , int * p2 )
{
    int num1 , num2 ;
    scanf( "%d%d" , & num1 , & num2 );
    *p1 = num1 ;
    *p2 = num2 ;
}
int main( void )
{
    int x , y ;
    readTwoInt( & x , & y );
    return 0 ;
}
```



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

2.2. Passing multiple values to a caller via parameters

```
void readTwoInt( int * p1 , int * p2 )
{
   int num1 , num2 ;
   scanf( "%d%d" , & num1 , & num2 );
   *p1 = num1 ;
   *p2 = num2 ;
}
int main( void )
{
   int x , y ;
   readTwoInt( & x , & y );
   return 0 ;
}
```



Through **p1** and **p2**, **readTwoInt()** is able to copy the result 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

```
void readTwoInt( int * p1 , int * p2 )
{
   //int num1 , num2 ;
   scanf( "%d%d" , p1 , p2 );
   //*p1 = num1 ;
   //*p2 = num2 ;
}
int main( void )
{
   int x , y ;
   readTwoInt( & x , & y );
   return 0 ;
}
```

Think about this!

Now, you should understand why calling `scanf` usually has &

Since **p1** and **p2** are storing the address of **x** and **y**, we can pass the addresses to **`scanf()`** directly (i.e., no need **&**). This way, **`scanf()`** will store the input directly in **x** and **y**.

2.3. Swapping the value of two variables

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

main function

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

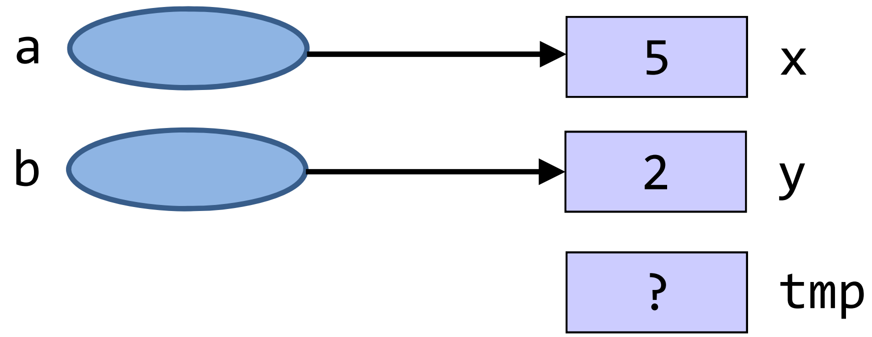
```c
// Version 2
void swap( int * a , int * b )
{
  int *tmp = a ;
  a = b   ;
  b = tmp ;
}
```

```c
// Version 3
void swap( int a , int b )
{
  int tmp = a ;
  a = b   ;
  b = tmp ;
}
```

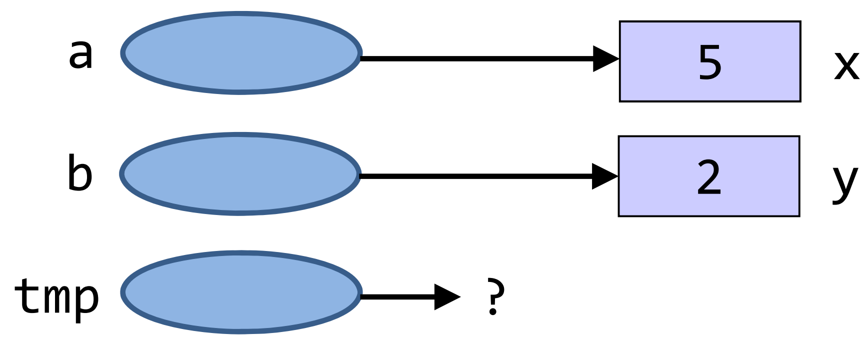
Which version of **`swap()`** can correctly swap the value of **x** and **y** in **`main()`**?

2.3. Swapping the value of two variables

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



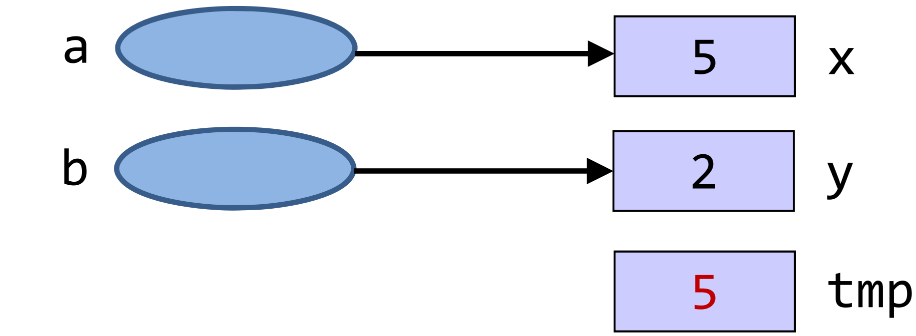

```
// Version 2
void swap( int * a , int * b )
{
   int *tmp = a ;
   a = b    ;
   b = tmp ;
}
```



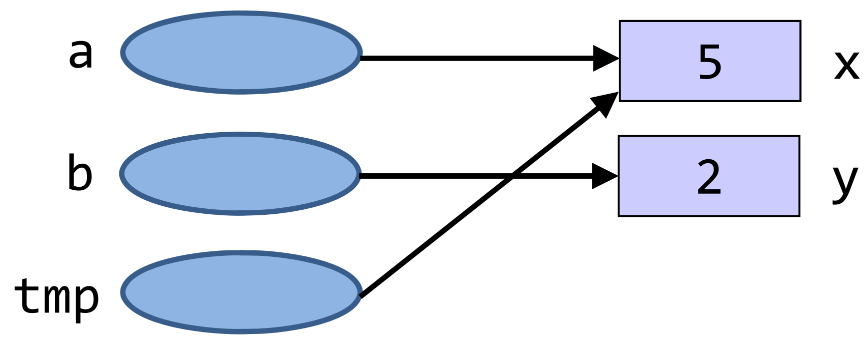
When the function call just starts…

2.3. Swapping the value of two variables

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

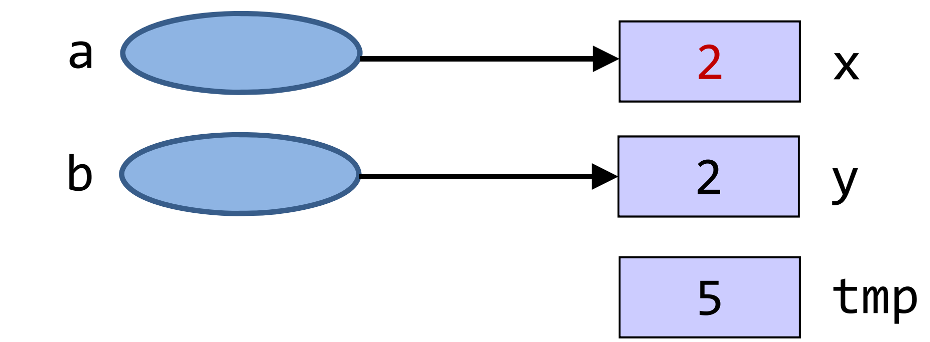



```
// Version 2
void swap( int * a , int * b )
{
  int *tmp = a ;
  a = b    ;
  b = tmp ;
}
```

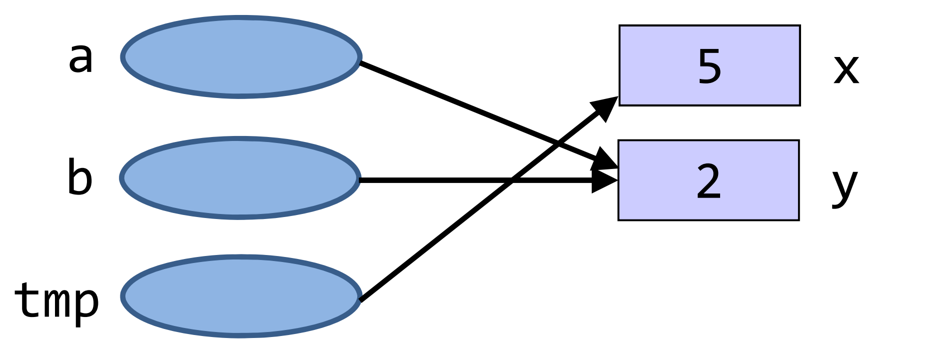


2.3. Swapping the value of two variables

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

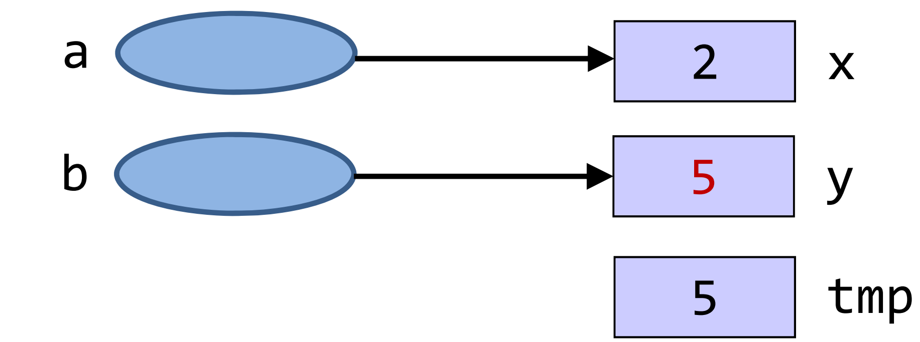



```
// Version 2
void swap( int * a , int * b )
{
  int *tmp = a ;
  a = b    ;
  b = tmp ;
}
```

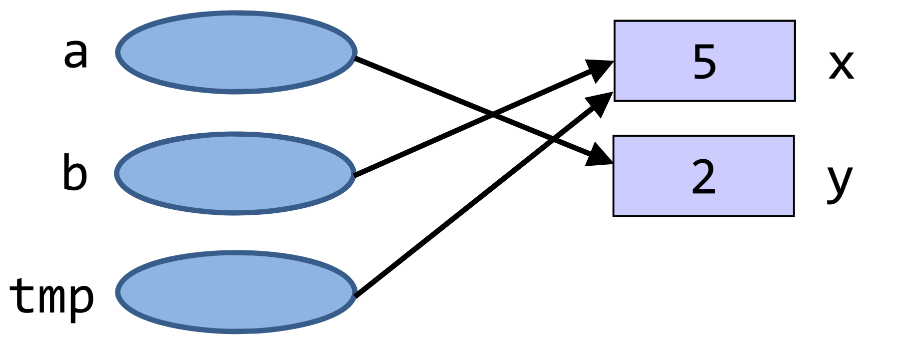


2.3. Swapping the value of two variables

```
// 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.

Pass by value (C) VS. Pass by reference (C++)

```c
void foo( int val )
{
  val = 0 ;
}

int main( void )
{
  int x = 3 ;

  foo( x );
  printf( "%d" , x );
  return 0 ;
}
```

C: pass by value (val is a local variable)

```c
void foo( int * p )
{
  *p = 0 ;
}

int main( void )
{
  int x = 3 ;

  foo( & x );
  printf( "%d" , x );
  return 0 ;
}
```

C: use pointer to emulate pass by ref.

```cpp
void foo( int & val )
{
  val = 0 ;
}

int main( void )
{
  int x = 3 ;

  foo( x );
  printf( "%d" , x );
  return 0 ;
}
```

C++: pass by ref. (val is the same variable as x)

Outline

~~1. Pointers in C language~~

- ~~What is a pointer?~~
- ~~Pointer syntax~~

~~2. Applications: 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

3. Additional Pointer Concepts

1. Basic Pitfalls
2. Pointer to Pointer
3. Pointer and Array
4. Pointer Arithmetic
5. Dynamic memory allocation
6. Pointers and Multi-dimensional arrays
7. Dangling Pointers and Memory Leakage
8. Pointers and Structure
9. Constant Pointers

3.1. Basic 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.

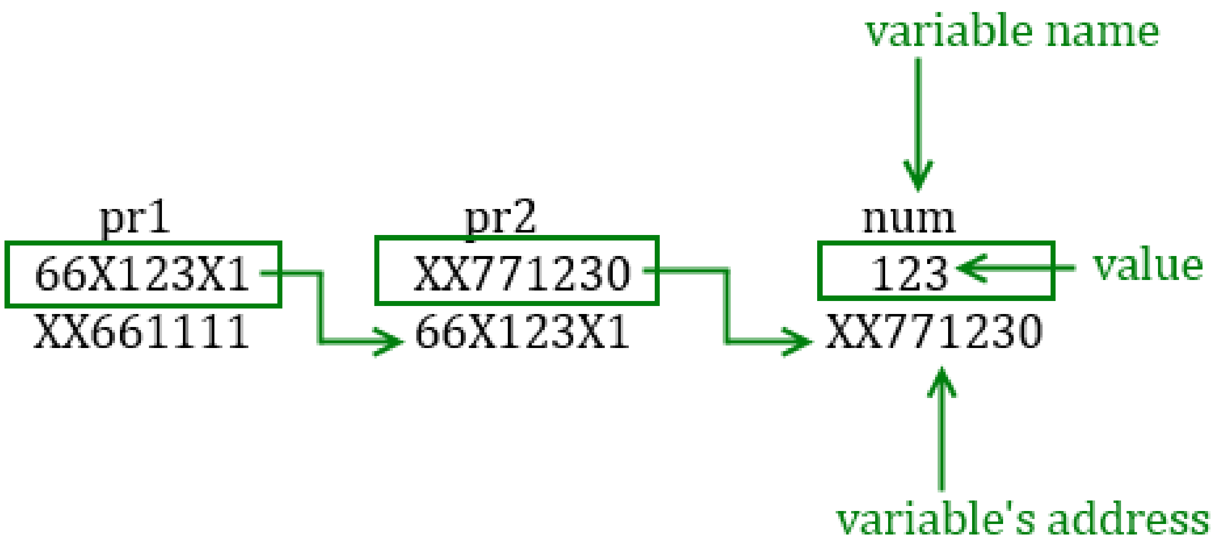
`int *p , foo = 10 ;`
`…`
`*p = & foo ;`
	Compile-time error (incompatible types).
	The type of the left operand, *p, is int.
	The type of the right operand, &foo, is an address.

`int foo = 10 ;`
`int *p;`
`*p = (int) & foo ;`
	Invalid… and it doesn’t make sense

`int *p;` `p = (int *) 100 ;`	Valid… but what does it mean? Danger ahead!!!

3.2. Pointer to Pointer

```
int   a     =   100  ;
int   b     =   102  ;
int  *ptr1 = & a     ;
int **ptr2 = & ptr1 ;
```



3.2. Pointer to Pointer

```
int    a     =   100   ;
int    b     =   102   ;
int   *ptr1 = & a     ;
int **ptr2 = & ptr1 ;

printf( "%d %d\n" , a , b ) ;
printf( "%d %d\n" , *ptr1 , **ptr2 ) ;
*ptr1 = 104 ;
printf( "%d %d\n" , a , b ) ;
printf( "%d %d\n" , *ptr1 , **ptr2 ) ;
ptr1 = & b ;
printf( "%d %d\n" , a , b ) ;
printf( "%d %d\n" , *ptr1 , **ptr2 ) ;
**ptr2 = 106 ;
printf( "%d %d\n" , a , b ) ;
printf( "%d %d\n" , *ptr1 , **ptr2 ) ;
```

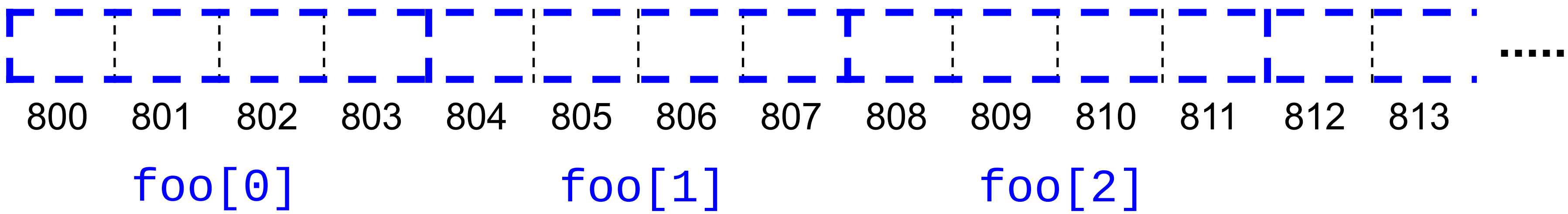
- What is the data type of “ptr2”?
- What will print out from the program above?

```
100 102
100 100
104 102
104 104
104 102
102 102
104 106
106 106
```

3.3. Pointer and Array

- First, let's look at how an array is stored in memory:

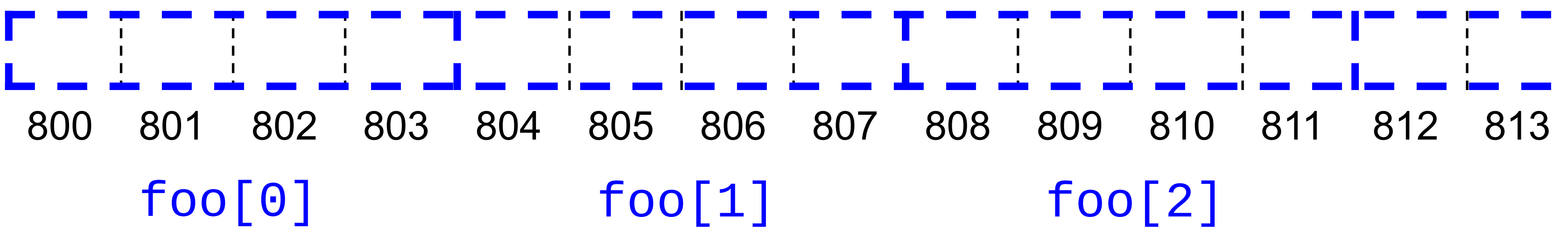
```
int foo[10] ; // Note: consider int as 4-byte
```



- What is the address of `foo[0]`?
- What is the address of array `foo`?

Note: Address of an array == Address of its first element (also known as the *base address*)

How are 1-D arrays stored in memory?



- What is the address of `foo[2]`?
- What is the address of `foo[9]`?
- What is the address of `foo[100]`?

sizeof(`int`) = 4

- Address of `foo[idx]` = base address + idx $\times$ 4
  - Given: each `int` data has 4 bytes

Note: <u>Array size</u> (num. of elements when you define an array) plays no role in determining the address of each array element.

C Representation of 1-D Arrays

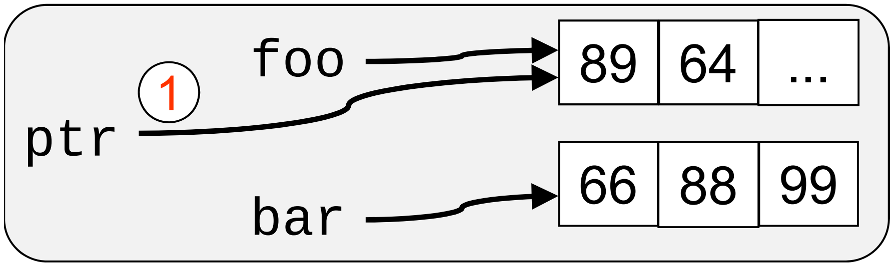
- An array in C (and also C++) is represented using its base address.
- E.g.,

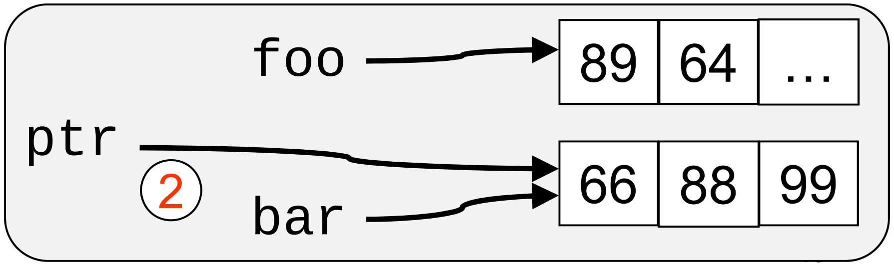
```
int    arr1[10], arr2[100];
double arr3[2];
printf( "%p\n" , arr1 );  // print address of arr1
printf( "%p\n" , arr2 );  // print address of arr2
printf( "%p\n" , arr3 );  // print address of arr3
```

Pointer Variables and Arrays

- An array variable can be seen as a pointer variable storing the base address of the array.
  - So we can assign an array variable to a pointer variable.

```
int *ptr ;
int  foo[5] = { 89, 64, 71, 928, 4 };
int  bar[3] = { 66, 88, 99 };
1 ptr = foo;  // ptr gets the base address of array foo
......
2 ptr = bar;  // ptr gets the base address of array bar
```





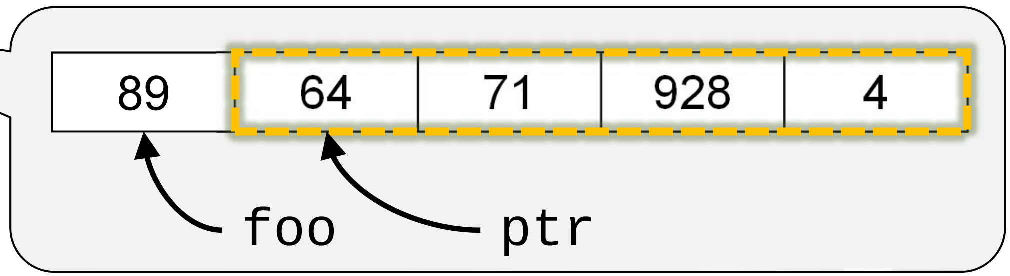
Pointer Variables and Arrays

- A pointer can be used as if it is an array.

```
int *ptr ;
int  foo[5] = { 89 , 64 , 71 , 928 , 4 };
int  bar[3] = { 66 , 88 , 99 };

ptr = foo ;  // ptr gets the base address of array foo

// ptr can now be used as an array
// The value stored in ptr is used as the base address
printf( "%d\n" , ptr[0] );  // prints
printf( "%d\n" , ptr[3] );  // prints

ptr = bar ;  // ptr gets the base address of array bar
printf( "%d\n" , ptr[1] );  // prints
```

Pointer Variables and Arrays

```
int *ptr , foo[5] = { 89 , 64 , 71 , 928 , 4 };

// ptr gets the address of foo[1]
ptr = & foo[1];
```




```
// ptr can be treated as an array whose base
// address is the address of foo[1].

// The value stored in ptr is used as the base address
printf( "%d\n" , ptr[0]+ptr[3] ); //
printf( "%d\n" , ptr[-1]        ); //
printf( "%d\n" , ptr[-2]        ); //
printf( "%d\n" , ptr[ 4]        ); //
```

Pointer Variables and Arrays

- How about the other way around?
  - Can we assign a memory address to an array variable?
  - An **array variable** should always point to the **same base address**!

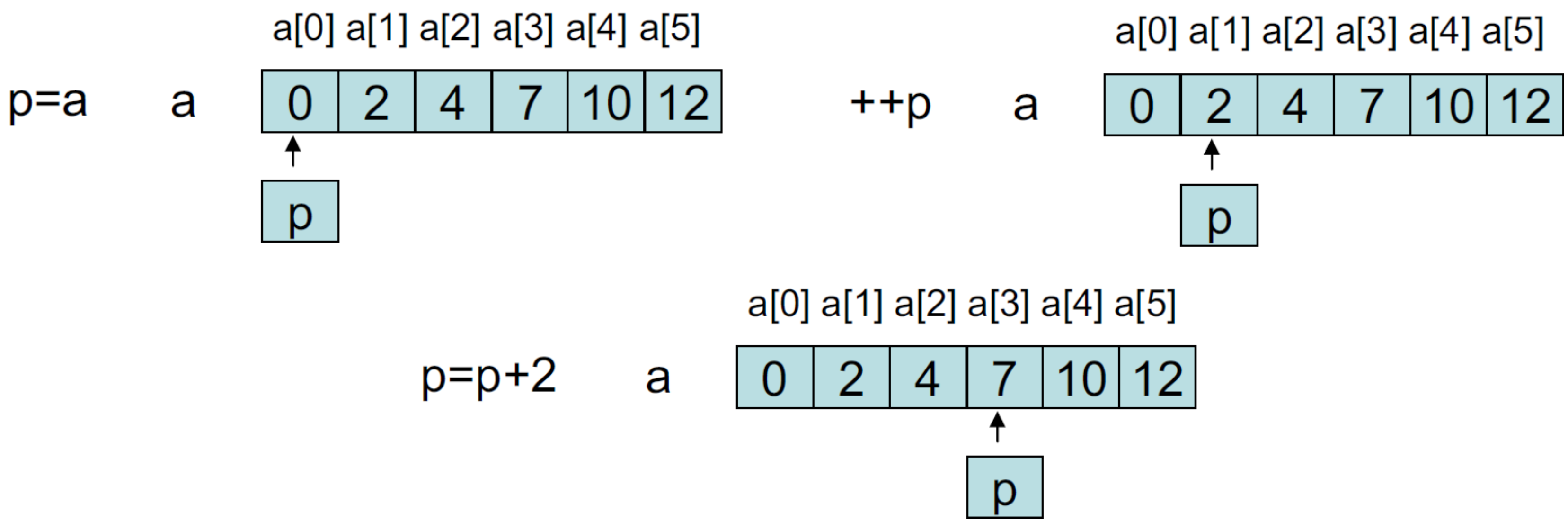
```
int *ptr ;
int  foo[5] = { 89, 64, 71, 928, 4 };
int  bar[3] = { 66, 88, 99 };
bar = foo;  // base address of array foo to bar?
            // it is a compilation error!!!
……
```

Note: the array name is just like a constant pointer!!!
(see later section in this lecture)

3.4. Pointer arithmetic

- Arithmetic on pointers has a different meaning than arithmetic on “numbers”. Adding an integer i to p says that p should be advanced i data items

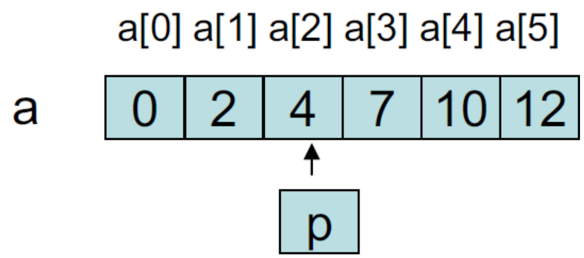
```
SomeType * p ;    // Set p to be a pointer to some type
p = p + i ;       // increment p by i*sizeof(SomeType) bytes
```



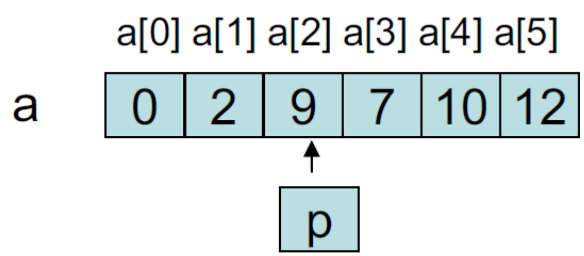
3.4. Pointer arithmetic

- More examples

p=a+2

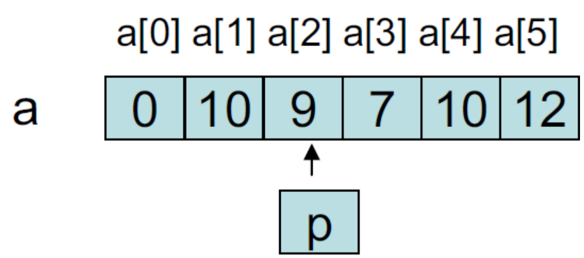


p=(p+1)+2



But… Beware of
Array out of bound error

(p-1)=(p+2)



3.5. Dynamic Memory Allocation (malloc)

```
// static allocation of 100 integers (stack memory)
int arr1[ 100 ] ;

// initialize a pointer (good habit to set it first to NULL)
int *arr2 = NULL ;

// dynamic allocation of 100 integers
arr2 = (int *) malloc( sizeof(int) * 100 );
...
free( arr2 ); // after you are done
arr2 = NULL ;
```

- Keywords in C (this is heap memory):
  - “malloc” allocates memory and returns base address
  - “free” deallocates memory and gives it back to OS

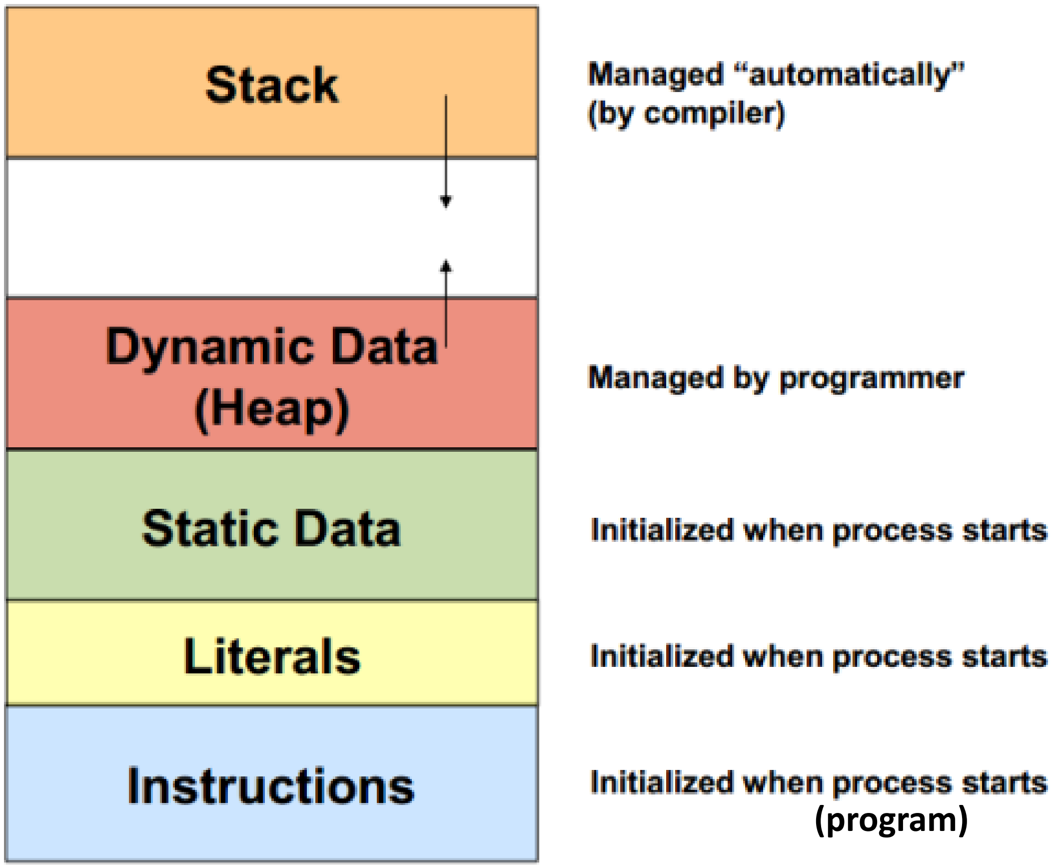
3.5. Dynamic Memory Allocation (malloc)

```c
// static allocation of 100 integers (stack memory)
int arr1[ 100 ] ;

// initialize a pointer (good habit to set it first to NULL)
int *arr2 = NULL ;

// dynamic allocation of 100 integers
int n ;
scanf( "%d" , & n );
arr2 = (int *) malloc( sizeof(int) * n );
... // still need free()
```

- Note:
  - Variable “n” can take any positive integer value -> dynamic alloc.
  - We usually need to type cast the return pointer accordingly, e.g., (int *)

3.5. Dynamic Memory Allocation (malloc)

When you run a program, the program is loaded into the computer memory:




```
// static allocation
// -> stack memory
int arr1[ 100 ] , data ;

// initialize a pointer int
*arr2 = NULL ;

// dynamic allocation
// -> heap memory
arr2 = (int *)
  malloc( sizeof(int) * 100 );
...
```

3.6 Pointers and Multi-dimensional arrays

Static allocation of 2D array

- **When you create a 2D array**

```
int board[3][4] = { {1,2,3,4}, {5,6,7,8}, ... };
```

`board[0][0]`	`board[0][1]`	`board[0][2]`	`board[0][3]`
`board[1][0]`	`board[1][1]`	`board[1][2]`	`board[1][3]`
`board[2][0]`	`board[2][1]`	`board[2][2]`	`board[2][3]`

Memory Addresses in static 2D Array

```
int board[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };

printf("Test Part 1:\n");
printf("- board       = %p\n" , &(board)       );
printf("- board[0]    = %p\n" , &(board[0])    );
printf("- board[0][0] = %p\n" , &(board[0][0]) );
printf("- board[1]    = %p\n" , &(board[1])    );
printf("- board[1][0] = %p\n" , &(board[1][0]) );
```

```
Test Part 1:
- board       = 0060FEA0
- board[0]    = 0060FEA0
- board[0][0] = 0060FEA0
- board[1]    = 0060FEB0
- board[1][0] = 0060FEB0
```

What is 0060FEB0 - 0060FEA0?

16 bytes

```
printf("Test Part 2:\n");
printf("- diff[0][1]-[0][0] = %d\n" , &(board[0][1]) - &(board[0][0]) );
printf("- diff[1][0]-[0][0] = %d\n" , &(board[1][0]) - &(board[0][0]) );
printf("- diff[1]-[0]       = %d\n" , &(board[1])    - &(board[0])    );


printf("Test Part 3 (char-based):\n");
printf("- diff[0][1]-[0][0] = %d\n",(char*)&(board[0][1])-(char*)&(board[0][0]));
printf("- diff[1][0]-[0][0] = %d\n",(char*)&(board[1][0])-(char*)&(board[0][0]));
printf("- diff[1]-[0]       = %d\n",(char*)&(board[1])-(char*)&(board[0]));
```

Memory Addresses in static 2

```
Test Part 1:
- board       = 0060FEA0
- board[0]    = 0060FEA0
- board[0][0] = 0060FEA0
- board[1]    = 0060FEB0
- board[1][0] = 0060FEB0
```

```
Test Part 2:
- diff[0][1]-[0][0] = 1
- diff[1][0]-[0][0] = 4
- diff[1]-[0]       = 1
```

```
Test Part 3 (char-based):
- diff[0][1]-[0][0] = 4
- diff[1][0]-[0][0] = 16
- diff[1]-[0]       = 16
```

```
int board[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,

printf("Test Part 1:\n");
printf("- board       = %p\n" , &(board)       );
printf("- board[0]    = %p\n" , &(board[0])    );
printf("- board[0][0] = %p\n" , &(board[0][0]) );
printf("- board[1]    = %p\n" , &(board[1])    );
printf("- board[1][0] = %p\n" , &(board[1][0]) );


printf("Test Part 2:\n");
printf("- diff[0][1]-[0][0] = %d\n" , &(board[0][1]) - &(board[0][0]) );
printf("- diff[1][0]-[0][0] = %d\n" , &(board[1][0]) - &(board[0][0]) );
printf("- diff[1]-[0]       = %d\n" , &(board[1])    - &(board[0])    );


printf("Test Part 3 (char-based):\n");
printf("- diff[0][1]-[0][0] = %d\n",(char*)&(board[0][1])-(char*)&(board[0][0]));
printf("- diff[1][0]-[0][0] = %d\n",(char*)&(board[1][0])-(char*)&(board[0][0]));
printf("- diff[1]-[0]       = %d\n",(char*)&(board[1])-(char*)&(board[0]));
```

(int *)

(int *)

(int *[4])

Passing a 2D array into a function

- Given

```
int data[3][4];
```

data[0][0]	data[0][1]	data[0][2]	data[0][3]
data[1][0]	data[1][1]	data[1][2]	data[1][3]
data[2][0]	data[2][1]	data[2][2]	data[2][3]

```
int func1( int board[3][4] ); // ok
int func2( int board[][4]  ); // ok
int func3( int board[][]   ); // not ok!!!
```

- When passing `data` into a function, we have

```
func1( data );
```

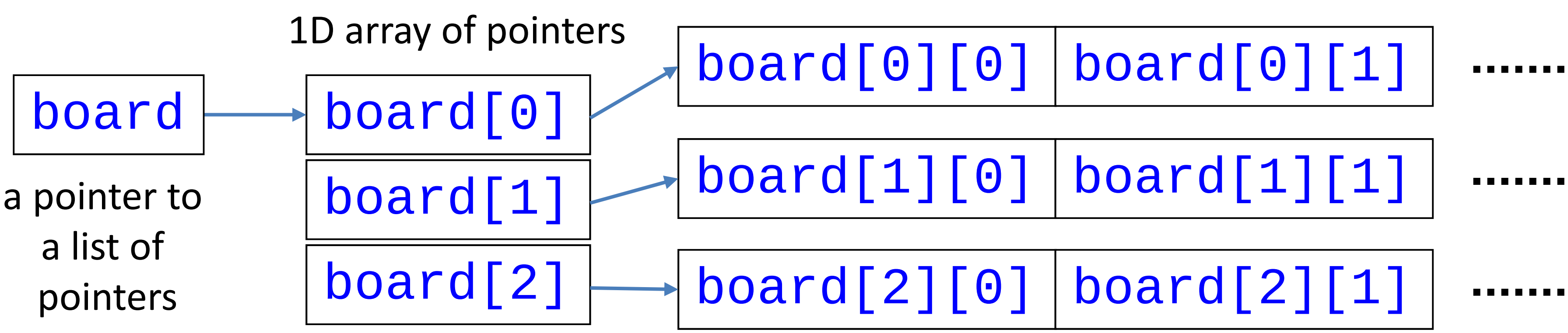
- Inside `func3()`, `board` is just a local variable that holds a memory address. If we say `data[1][0]`, `func3` cannot locate it, since `func3` doesn't know how many bytes in a row

3.6 Pointers and Multi-dimensional arrays

Dynamic allocation of 2D array

```
int board[3][4] ;
```

- We have **two steps:**
  - First, create a 1D array of pointers, each pointing to a row of data: `board[0] , … , board[2]`
  - Then, for each pointer "`board[i]`", create a 1D array of data:



Question: how many 1D arrays altogether?

Dynamic allocation of a 4x3 array

```
int ** arr2D = NULL ; // a pointer to a list of pointers

// 1) allocate an array of pointers
arr2D = (int **) malloc( sizeof(int *) * 3 );

// 2) allocate an array of data for each arr2D[i]
for ( int i = 0 ; i < 3 ; i ++ )
   arr2D[i] = (int *) malloc( sizeof(int) * 4 );

... // how to free?
```

- Two stages of dynamic memory allocation of 2D arrays:
  - Allocate the master array: arr2D
  - Allocate each data array: arr2D[i]

Dynamic allocation of a 4x3 array

```
...

// 1) free each arr2D[i]
for ( int i = 0 ; i < 3 ; i ++ )
   free( arr2D[i] );

// 2) free the master array of pointers
free( arr2D );
arr2D = NULL ;
```

- Two stages of memory de-allocation for 2D array:
  - Free each data array: arr2D[i]
  - Free the master array: arr2D

Note: in reversed order of memory allocation!

3.7. Two Pitfalls for Dynamic Memory Alloc.

- Dangling Pointers
- Memory Leakage

Pitfall #1: Dangling Pointer

- Be careful that when you free p, you are not releasing the memory that some other pointer **q is *still* pointing to**.

```
int *p ;        // p is an integer pointer variable
int *q ;        // q is an integer pointer variable
p = (int *) malloc( sizeof(int) * 10 ) ;
q = p ;
free(p);        // what happen to q? It becomes a dangling pointer
p = q = NULL ; // better to set them as NULL for sanity
```

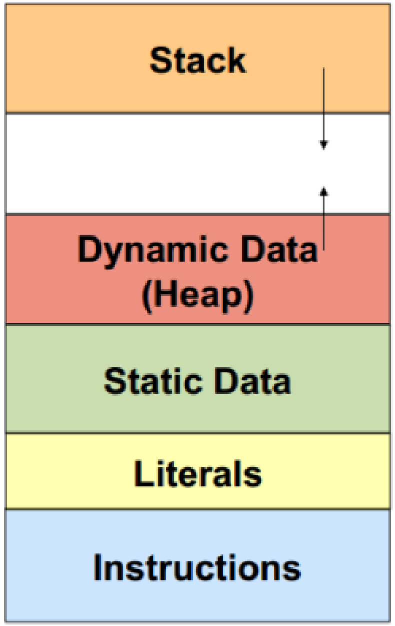
If you access q[0] after free(p), what will happen?

This is like "**Array out of bound error**"

Pitfall #2: Memory Leakage

- In Line 5, the memory for "q" can never be deallocated (or freed) and is lost, i.e., never returned to the memory heap.

```
int *p ;        // p is an integer pointer variable
int *q ;        // q is an integer pointer variable
p = (int *) malloc( sizeof(int) * 10 ) ;
q = (int *) malloc( sizeof(int) * 10 ) ;
q = p ;         // what happen at this moment?
// SHOULD free(q) before Line 5!!!
```



What if memory leakage keeps happening?

Out of heap memory -> Program crash!

A similar situation is that you call **malloc** but didn't **free**!!!

What kind of pitfall ?

```
int * compute()
{
    int data[5];
    ...
    return data ;
}
```

Discussion:

- What is the life time of data?
- What may happen if the caller of “compute” use the pointer to access the data?

```
int * compute()
{
    int *p ;
    p = (int *) \
       malloc( sizeof(int) * 5 ) ;
    ...
    return data ;
}
```

Discussion:

- What is the life time of p?
- What is the life time of the dynamically-allocated data?
- What should the caller do after using such data?
- What if the caller forget to do so?

Note: Avoid doing these!

The caller func. should prepare & pass an array into compute!

3.8. Pointers and Structure

Revisit Lecture “Structure”

- Pass a structure to a func. (by pointer) – P.22-23 & P.26-28
- A dynamic array of structure data – P.28-30

3.9. Constant Pointers

What is a constant?

const int const_data = 1 ;

Must initialize & cannot be changed

There are three cases:

- A pointer to a constant
- A pointer that is a constant
- A pointer that is a constant itself points to a constant

```
// a pointer to a constant (data cannot be changed)
const int * p1 ;

// a pointer that is a constant (pointer itself cannot be changed)
// an array is like a pointer that is a constant!   int data[10];
int * const p2 = & value ;

// the pointer cannot be changed and the thing it points to
// also cannot be changed
const int * const p3 = & const_data ;
```

3.9. Constant Pointers

Two fundamental concepts

- A data that is a constant cannot be changed, so
  - When you create a constant, you must assign a value to it, e.g.,

```
const int FULL_MARKS = 100 ;
```

- If pi is defined as a pointer to a const, this means that *pi can not be assigned to (else compilation error)

3.9. Constant Pointers

- Examples

```
int j = 10 ; const int i = 5 ;
const int * pi ;        // a pointer to a constant
pi = &i ;  pi = &j ;    // ok: pi can change
pi = &i ; *pi = 10 ;    // error: *pi can not be assigned to
pi = &j ; *pi = 10 ;    // error: ok for “&j” but *pi no change!
int *qi ;  qi = &i ;    // error: qi is not a pointer to const
```

```
int          i = 5   ;
int * const ri  ???  ;      // const, so must be assigned
printf( "%d\n" , *ri );     // ok
*ri = 10 ;                  // ok
int j ;
ri = & j ;             // compile-time error; cannot change ri
```

3.9. Constant Pointers

Acceptable software engineering practice demands that you make the following const:

- data that you don't intend to change:

```
const double PI = 3.1415927 ; // const must be initiaized
const Date   handover = { 1 , 7 , 1997 };
```

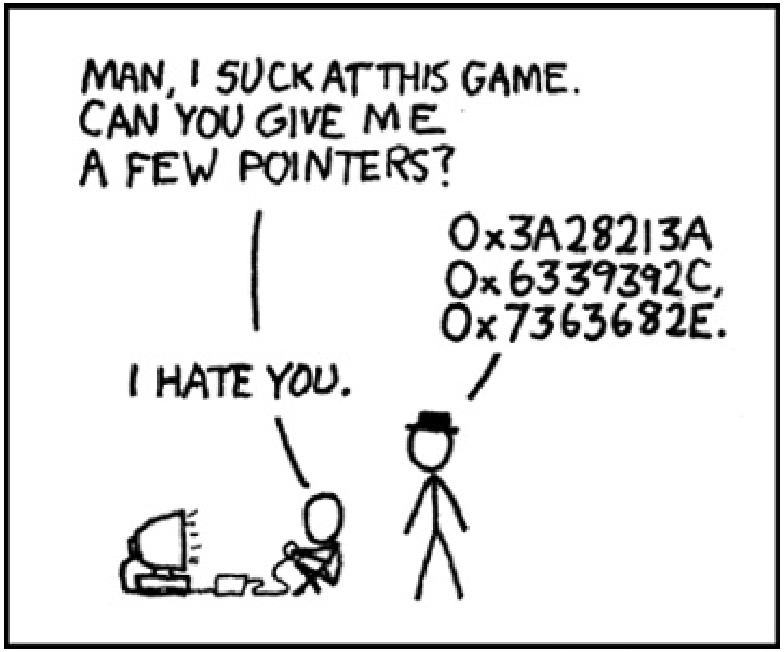
- function arguments that you don't intend to change, particularly those with pointers:

```
void print_height( const Student * ptr )
{
    print( "height: %f\n" , ptr->height ) ;
}
```

https://en.wikipedia.org/wiki/Const_(computer_programming)#C_convention

Summary

- Concepts of computer memory and how data are stored in the memory
- Know how to declare pointer variables and use the address-of operator (**&**) and the dereference operator (`*`)
- Understand the difference between `p1 = p2;` and `*p1 = *p2;`



FakePlus.com

- Understand passing pointers to functions (note: safer to use reference variable in C++)
- Understand pointer to pointer, pointer vs array, and dynamic memory allocation.