Lecture 13:

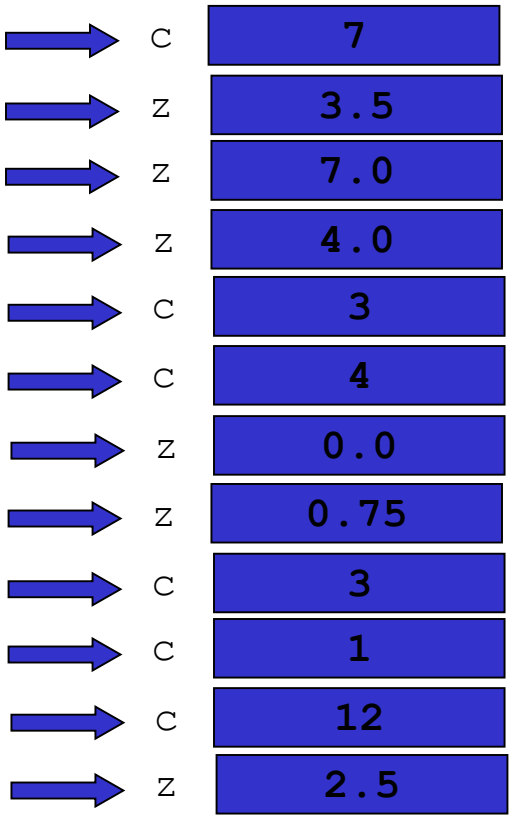
Pointer Arithmetic and Operations

Prof. Shervin Shirmohammadi
University of Ottawa

Arithmetic Expressions With Dereferenced Pointers

- Using the indirection operator, we can write arithmetic expressions that are similar to those written using variables.
  - E.g.:

```
int a=3, b=4, c, *aPtr=&a, *bPtr=&b, *cPtr=&c;
float x=1.0, y=2.5, z, *xPtr=&x, *yPtr=&y, *zPtr=&z;
*cPtr = *aPtr + *bPtr;
*zPtr = *xPtr + *yPtr;
*zPtr = *aPtr + *bPtr;
*zPtr = *xPtr + *aPtr;
*cPtr = *xPtr + *yPtr;
*cPtr = *xPtr + *aPtr;
*zPtr = *aPtr / *bPtr;
*zPtr = (float) *aPtr / *bPtr;
*cPtr = *aPtr % *bPtr;
*cPtr = *bPtr % *aPtr;
*cPtr = *aPtr * *bPtr;
*zPtr = *xPtr * *yPtr;
```



Loss of info!

Logical Expressions With Dereferenced Pointers

- Using the indirection operator, we can write logical expressions that are similar to those written using variables.
  - E.g.:
    - `int a = 0, *aPtr = &a, b = 1, *bPtr = &b;`

```
if (*aPtr > *bPtr)
printf("a is greater than b.\n");

if (!(*aPtr))
printf("a is false.\n");

if ((*aPtr + 2) > *bPtr)
printf("%d > %d\n", (*aPtr + 2), *bPtr);
```

- Pointers can be compared to other pointers, addresses and NULL
- Most useful comparisons include comparison to NULL and comparison to the addresses of elements in a table.

	`int a, *aPtr=NULL, b, *bPtr=&b;`
If address of a is not equal to null *Print "This is always true."*	`if(&a != NULL)` `  printf("This is always true.\n");`
If address of a equals address of b *Print "We might have problems"*	`if(&a == &b)` `  printf("We might have problems.\n");`
If aPtr is equal to null *Assign address of a to aPtr*	`if(aPtr == NULL)` `  aPtr=&a;`
If aPtr is not eqal to bPtr *Assign bPtr to aPtr*	`if(aPtr != bPtr)` `  aPtr=bPtr;`

Example

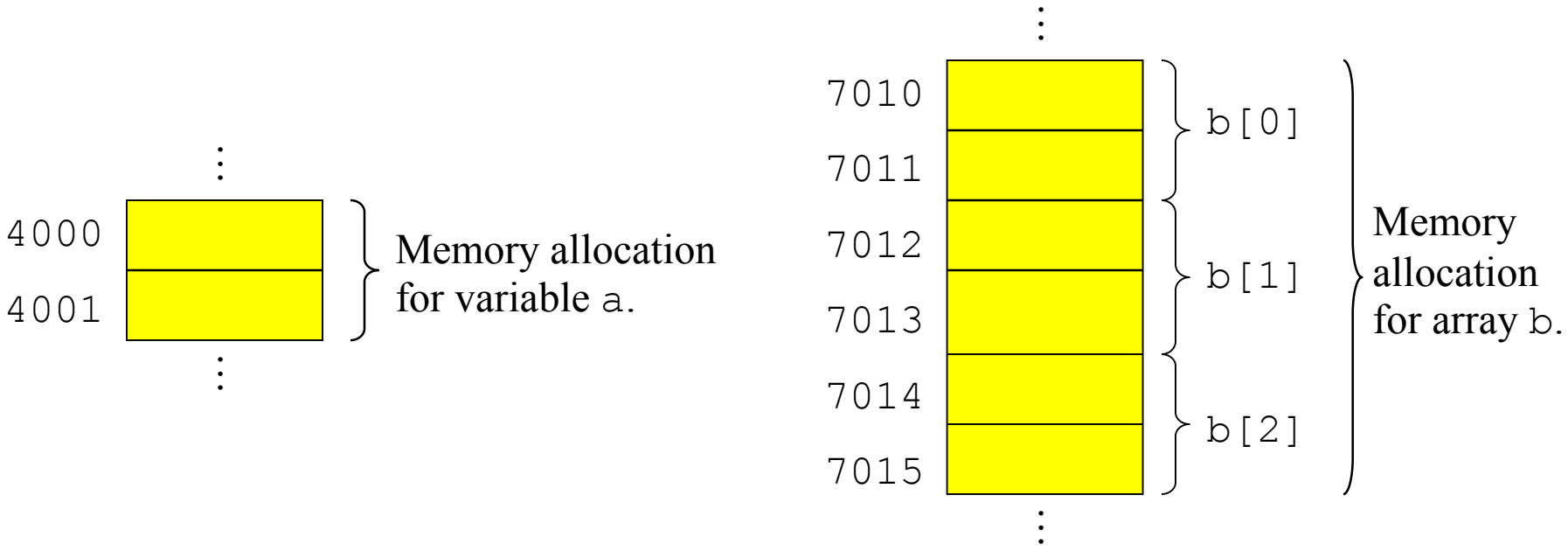
- Write a program to add two floats once by reference and once by value.

Pointers and Arrays

- Pointers and arrays are intimately related in C.
  - Any element in an array may be accessed via a pointer and using address arithmetic.
- The name of an array without the `[]` and the index **is in fact the address** to the first element in the array.
  - E.g.:
    - `int b[5], *bPtr;`
    - `bPtr = b;`
    - `bPtr = &b[0];`
  - The above assignments are equivalent: they both assign the address of the first element of array `b` to the pointer `bPtr`.
  - Note that the assignment such as `b = &a;` is not allowed. Why?
- Since array elements are guaranteed to be stored in contiguous memory elements, then it is possible to use address arithmetic with a pointer to access individual array elements.
  - Adding an integer to a pointer in fact adds the number `integer*sizeof(type)` to the pointer.
  - This operation is one type of *pointer arithmetic*.

Applying Pointer Arithmetic

- Arithmetic using pointers is in fact useful only to navigate through arrays since an array is the only data structure where individual array elements are guaranteed to be in contiguous memory locations.
  - Two variables of the same type defined in the same declaration are not guaranteed to be stored in contiguous memory locations.
  - Possible allocations for `int a, b[3];`



Pointer and Address Arithmetic

- Arithmetic operations with pointers are limited.

Pseudo-code	**C**	**Description**
Increment *pointerName*	++	Increments the address in *pointerName* by one element (i.e. the number of byes occupied by the type of variable referenced by *pointerName*)
Decrement *pointerName*	--	Decrements the address in *pointerName* by one element (i.e. the number of byes occupied by the type of variable referenced by *pointerName*)
pointerName + <integer number> *Eg:* ***aPtr+10***	*+ or +=*	Increments the address in *pointerName* by *<integer Number>* elements.
pointerName - <integer number> *Eg:* ***aPtr-3***	*- or -=*	Decrements the address in *pointerName* by *<integer Number>* elements.
Assign *pointerName1* ***to*** *pointerName2* *Eg:* ***Assign aPtr to bPtr***	=	The address in *ponterName1* is stored in *pointerName2.*

Pointer Arithmetic - Examples

Pseudo-code	C Instructions

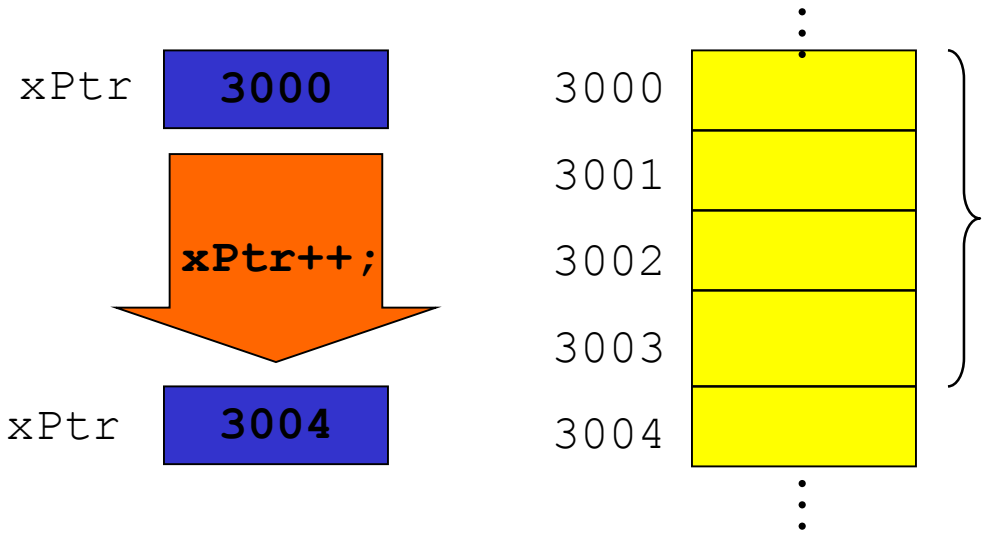
	`int a = 0, *aPtr = &a,`
	`int b = 1, *bPtr = &b;`
	`float x = 1.0, *xPtr = &x;`

Increment xPtr	`xPtr++; /*incr by sizeof(float)*/`
Decrement aPtr	`aPtr--; /*decr by sizeof(int))*/`
Assign aPtr-2 to aPtr	`aPtr -= 2;` `/*less 2*sizeof(int)*/`
Assign xPtr+2 to xPtr	`xPtr = xPtr + 2;` `/*plus 2*sizeof(float)*/`
Assign bPtr to aPtr	`aPtr = bPtr;`

Incrementing a pointer

Increment xPtr	`xPtr++; /* xPtr points to the next value of type float. */`

- `xPtr` is a pointer of type `float`, and contains the value 3000 (thus at address 3000 the first byte of a `float` value is stored).
- After the execution of "`xPtr++;`", `xPtr` contains the value 3004 since the `float` value occupies 4 bytes.
- What will be the value in `xPtr` after the execution of "`xPtr = xPtr +2;`"?



- Adding `1` to a pointer to an array positions the pointer to the next array element:
  - `int b[5], *bPtr = &b[0];`
  - `*(bPtr + 0) = 1;` → Stores `1` in `b[0]`
  - `*(bPtr + 1) = 10;` → Stores `10` in `b[1]`
- Parentheses are required in the above to force the precedence of the addition + over the indirection `*`.
- The **integer offset** corresponds directly to the array element that we wish to access.

- We can also add an index to a pointer.
  - Based on the previous example:
    - `bPtr[2] = 1;` → Stores `1` in `b[2]`
  - We do not use the indirection operator `*` when accessing array elements using the `[]` and an index.
  - The `[]` performs the necessary address arithmetic and indirection automatically.

- We thus have four ways in C of accessing an array element:

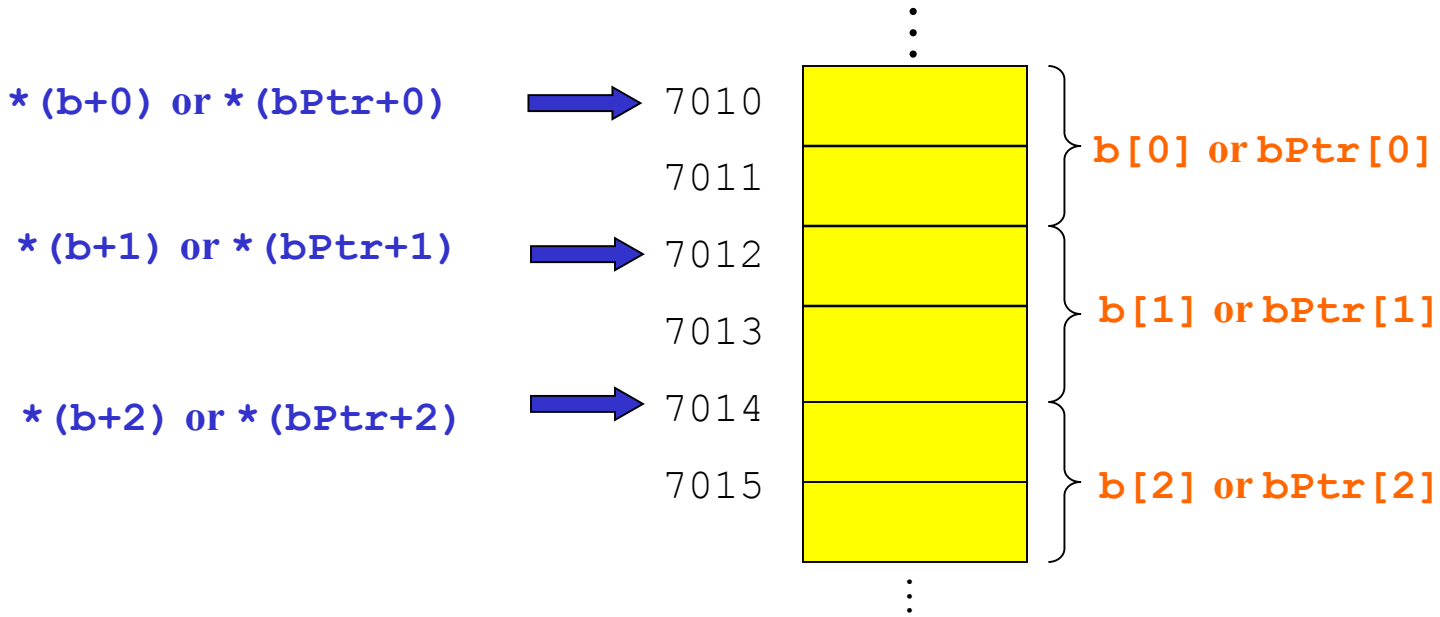
Best style! – `b[2] = 1;` → Stores `1` in `b[2]`

– `bPtr[2] = 1;` → Stores `1` in `b[2]`

– `*(bPtr + 2) = 1;` → Stores `1` in `b[2]`

– `*(b + 2) = 1;` → Stores `1` in `b[2]`

- E.g.:
  - Assume that the following declaration results in array `b` being stored from address `7010` onwards:
    - `int b[3], *bPtr;`
    - `bPtr = &b[0];`
  - The array name `b` or the pointer `bPtr` followed by an integer index placed between `[]` can be used to access an element of the array.
  - Since `b` is the address of the first element of the array then the indirection of **(b plus an offset)** can also be used to access an element of the array.



Example

- Write a program that prints all elements of an array using a pointer to that array.