COMsW 1003-1

Introduction to Computer Programming in C

Lecture 11

Spring 2011

Instructor: Michele Merler

http://www1.cs.columbia.edu/~mmerler/comsw1003-1.html

Announcements

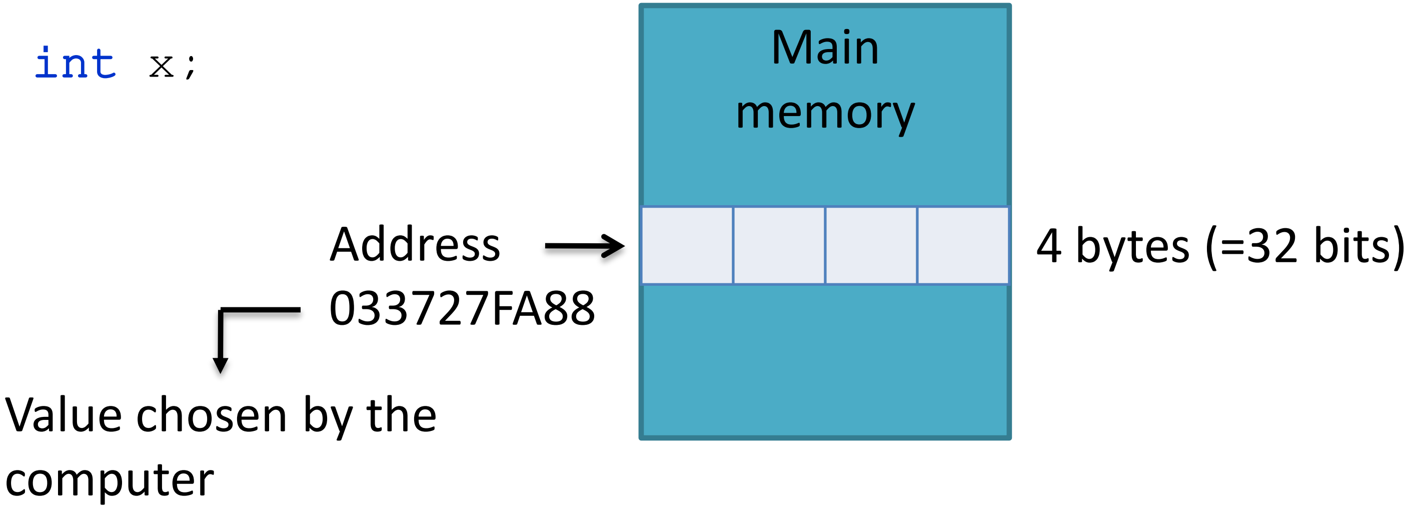
- Grades for Homework 1 posted on Courseworks
- Homework 2 is due next Monday at the beginning of class
- Bring the printout to class!

Pointers

Pointers

Remember what happens when we declare a variable: the computer allocates memory for it.

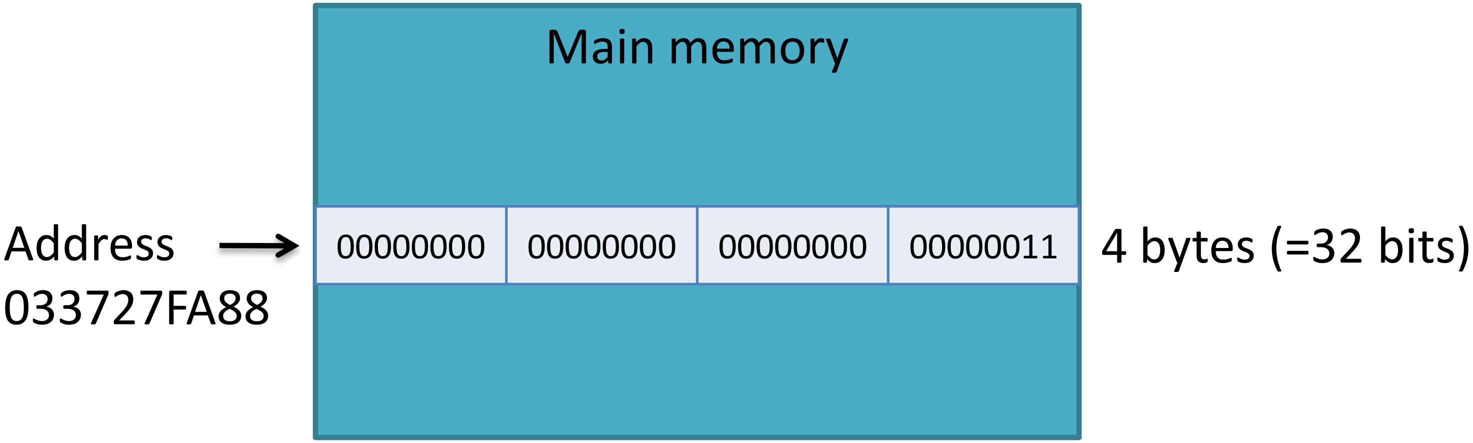
```
int x;
```



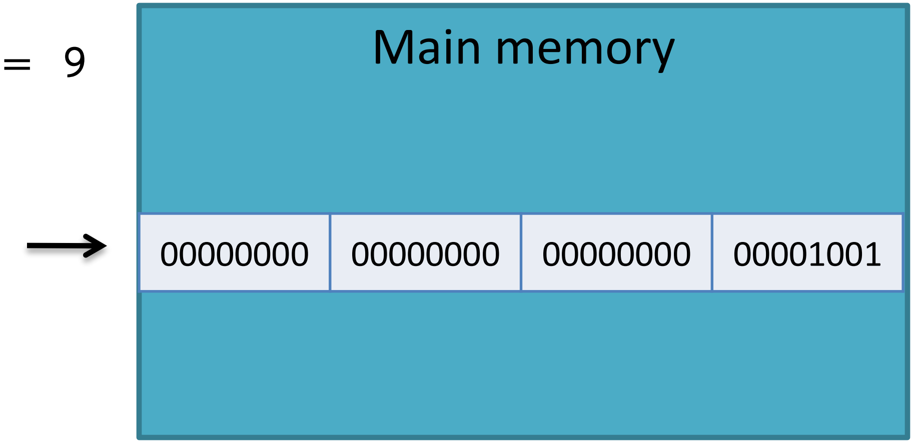
Pointers

When we assign a value to a variable, the computer stores that value at the address in memory that was previously allocated for that variable.

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



```
x *= 3; // x = 9
```



Pointers

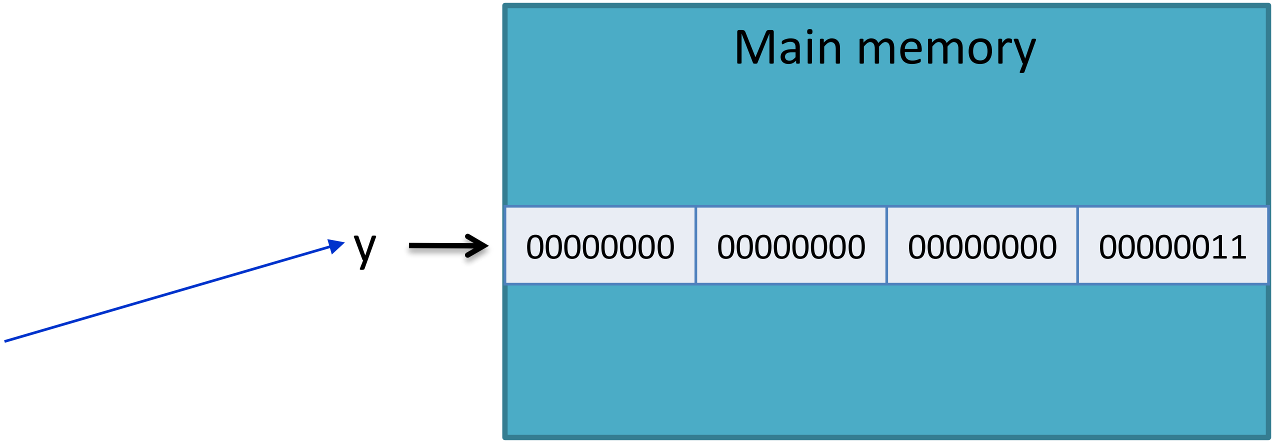
Pointers are variables for memory addresses.

They are declared using the * operator.

They are called pointers because they **point to the place in memory** where other variables are stored.

How can we know what the address in memory of a variable is?
The & operator.

```
int x;
x = 3;

int *y;

y = &x;
```



Pointers - Syntax

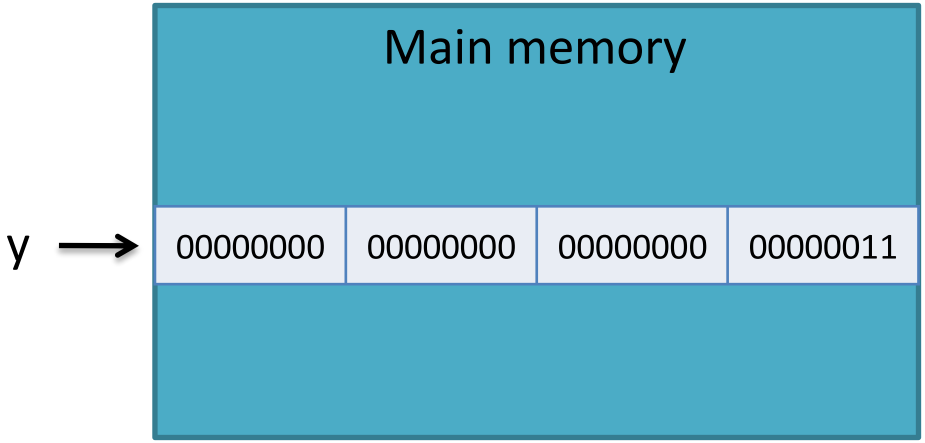
When we declare a pointer, we must specify the type of variable it will be pointing to

```
type *ptrName;
```

If we want to set a pointer to point to a variable, we must use the **&** operator

```
ptrName = &varName;
```

```
int x;
x = 3;

int *y;

y = &x;
```



Pointers : operators * and &

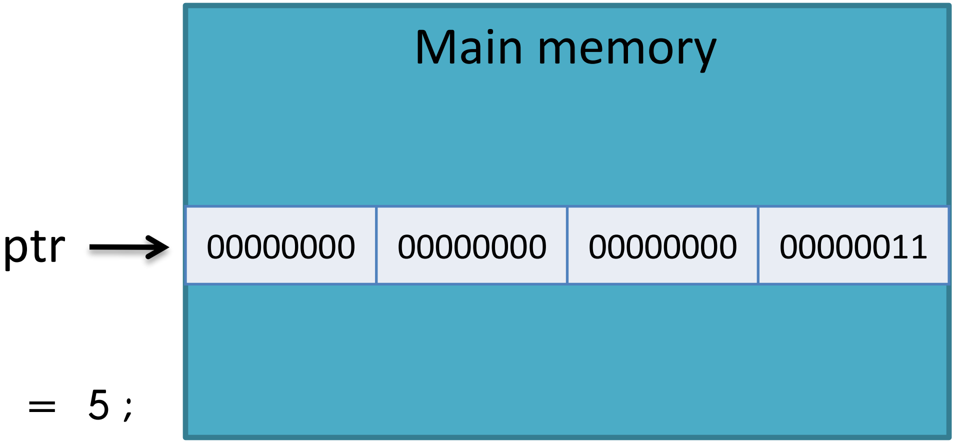
* **dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& **reference operator**: gives the address in memory of a variable (returns a pointer)

```
int x = 3;

int *ptr;

ptr = &x;

*ptr = 5; // x = 5;
```

Make ptr point to the address of x

Modify the value in address pointed by ptr



Main memory

ptr →

00000000	00000000	00000000	00000011

Pointers : operators * and &

* **dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& **reference operator**: gives the address in memory of a variable (returns a pointer)

```
int x = 3;

int *ptr;

ptr = &x;


*ptr = 5;    // x = 5;
```

Make ptr point to the address of x

Modify the value in address pointed by ptr

Code	Meaning
x	Variable of type **int**
ptr	**Pointer** to an element of type int
&x	Pointer to x
*ptr	Variable of type int

Pointers : operators * and &

* **dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& **reference operator**: gives the address in memory of a variable (returns a pointer)

```
int x;

int *ptr;
```

V

```
&x

*ptr
```

✗

```
&ptr // pointer to a pointer

*x   // x is not a pointer
```

Pointers : operators * and &

* **dereference operator** : gives the value in the memory pointed by a pointer (returns a value)

& **reference operator**: gives the address in memory of a variable (returns a pointer)

```
int x;

int *ptr;
```

This is weird but actually ok, we will see its meaning later

✓

```
&x

*ptr
```

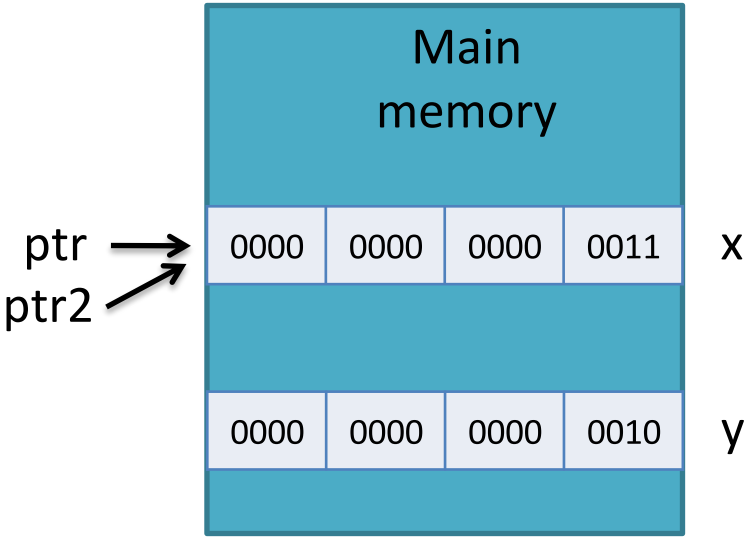
✗

```
&ptr// pointer to a pointer

*x    // x is not a pointer
```

Pointers

Multiple pointers can point to the same address

```
int x = 3, y = 2;

int *ptr = &x;

int *ptr2 = ptr;
```

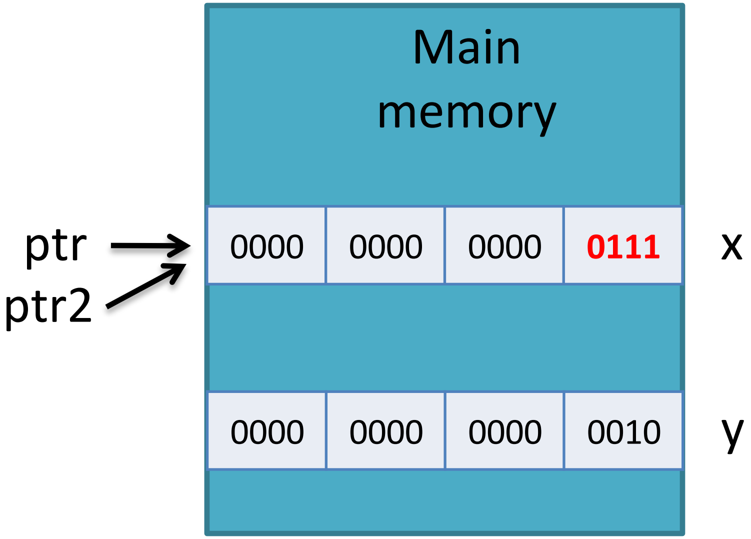


NOTE: first 4 bits omitted to save space

Pointers

Multiple pointers can point to the same address

```
int x = 3, y = 2;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;    // x = 7;
```

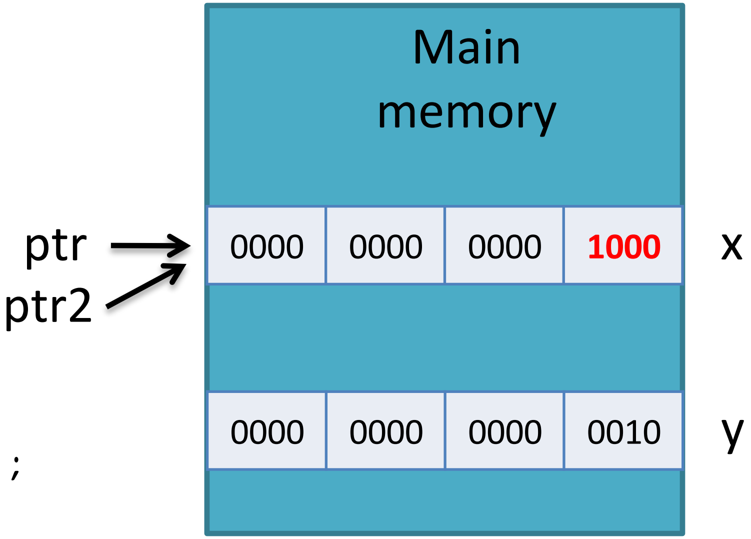


NOTE: first 4 bits omitted to save space

Pointers

Multiple pointers can point to the same address

```
int x = 3, y = 2;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;    // x = 7;
*ptr2 = *ptr2 + 1;   // x = 8;
```

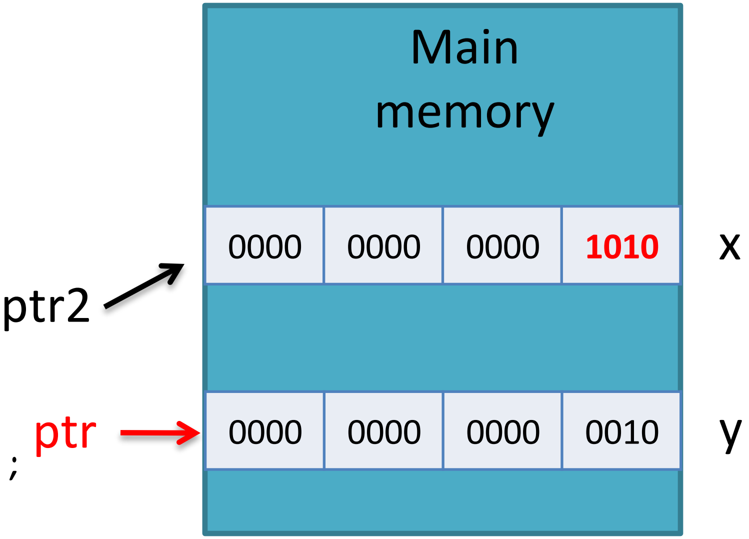


NOTE: first 4 bits omitted to save space

Pointers

Multiple pointers can point to the same address

```
int x = 3, y;

int *ptr = &x;

int *ptr2 = ptr;

*ptr = 7;    // x = 7;
*ptr2 = *ptr2 + 1;  // x = 8;

ptr = &y;

*ptr2 = 10; // x = 10;
```



Main memory

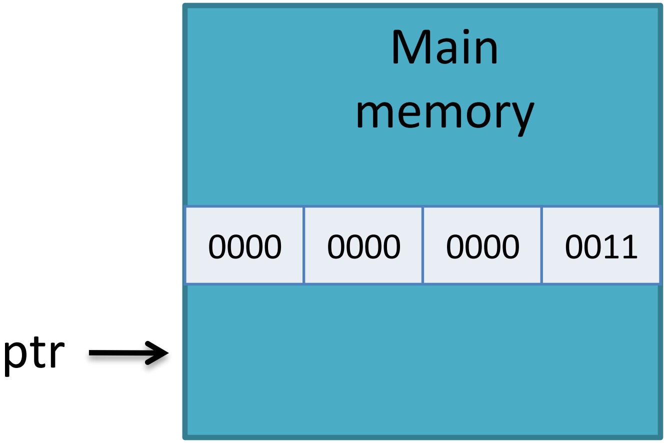
ptr2	0000	0000	0000	1010	x
ptr	0000	0000	0000	0010	y

Ptr2 is still pointing to x, even if ptr changed

Pointers

Be careful when using incremental operators!

```
int x = 3;

int *ptr = &x;

*ptr++;    // x = ?
```



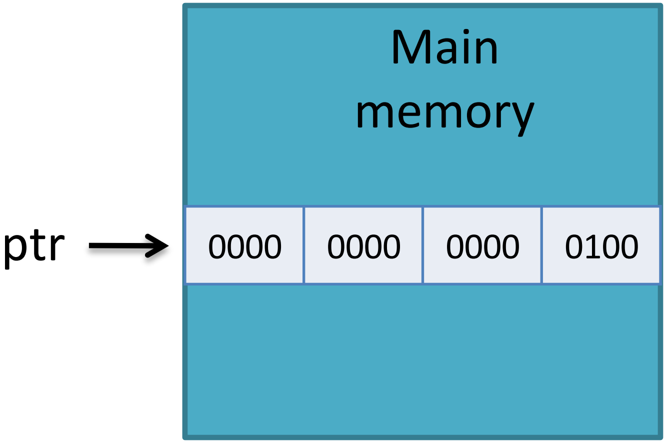
0000	0000	0000	0011

In this case I am incrementing `ptr`, NOT the value of the variable pointed by it!

Pointers

Be careful when using incremental operators!

```
int x = 3;

int *ptr = &x;

(*ptr)++;    // x = 4;
```



0000	0000	0000	0100

Pointers and Arrays

- When set a pointer to an array, the pointer points to the **first element** in the array

```c
float arr[3] = {1, 2, 5};
float *pa;

pa = arr;
pa = &arr[0];
```

} These two notations are equivalent

- C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to

```
arr[0]  ⟷  *pa
arr[1]  ⟷  *(pa+1)
arr[2]  ⟷  pa[2]
```

Pointers and Arrays

- When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};
float *pa;

pa = arr;
pa = &arr[0];
```

} These two notations are equivalent

- C automatically keeps pointer arithmetic in terms of the size of the variable type being pointed to

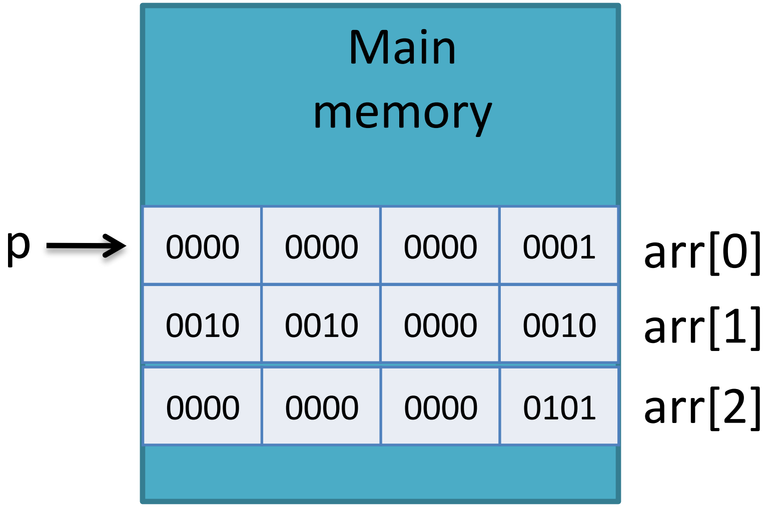
```
arr[0]  ⟷  *pa
arr[1]  ⟷  *(pa+1)
arr[2]  ⟷  pa[2]
```

→ Once we have set a pointer to the beginning of one array, we can use it as if it were the array itself!

Pointers and Arrays

When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;
```



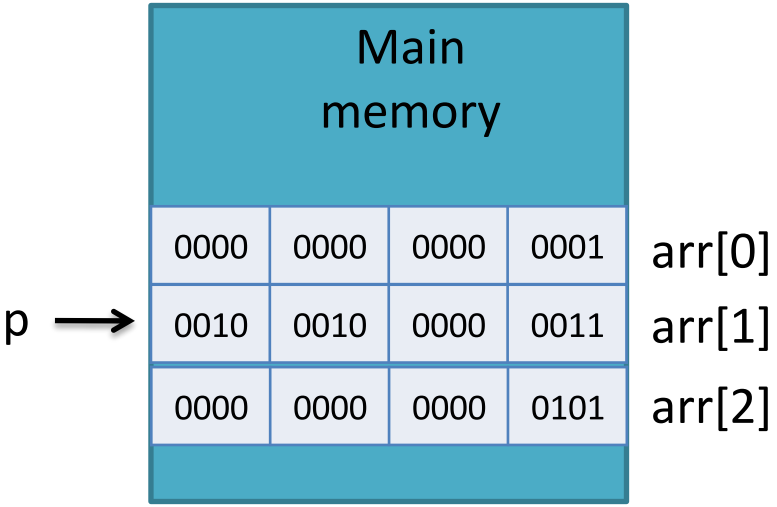
Main memory

p →	0000	0000	0000	0001	arr[0]
	0010	0010	0000	0010	arr[1]
	0000	0000	0000	0101	arr[2]

Pointers and Arrays

When set a pointer to an array, the pointer points to the **first element** in the array

```
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;

p++;

*p = 3; // arr[1] = 3;
```



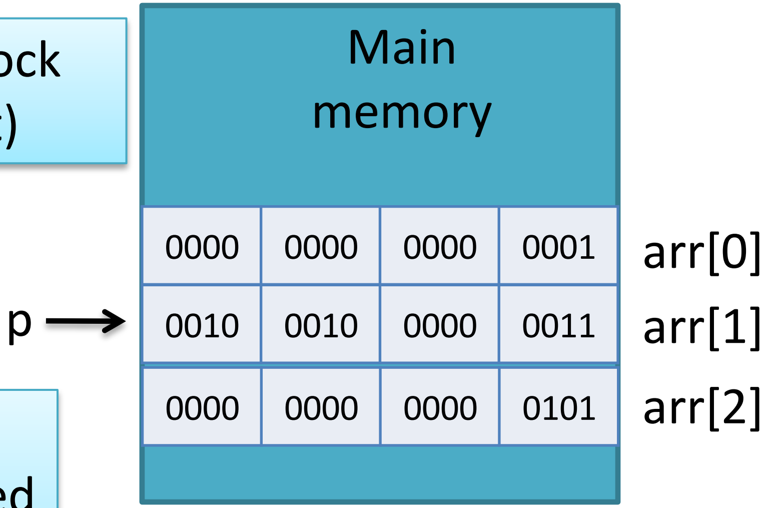
Pointers and Arrays

When set a pointer to an array, the pointer points to the first element in the array

```
float arr[3] = {1, 2, 5};

float *p = arr;

*p = 5; // arr[0] = 5;

p++;

*p = 3; // arr[1] = 3;
```

Note that for arrays, we do not need the reference & operator

p jumps in memory a block of 4 bytes (size of a float)



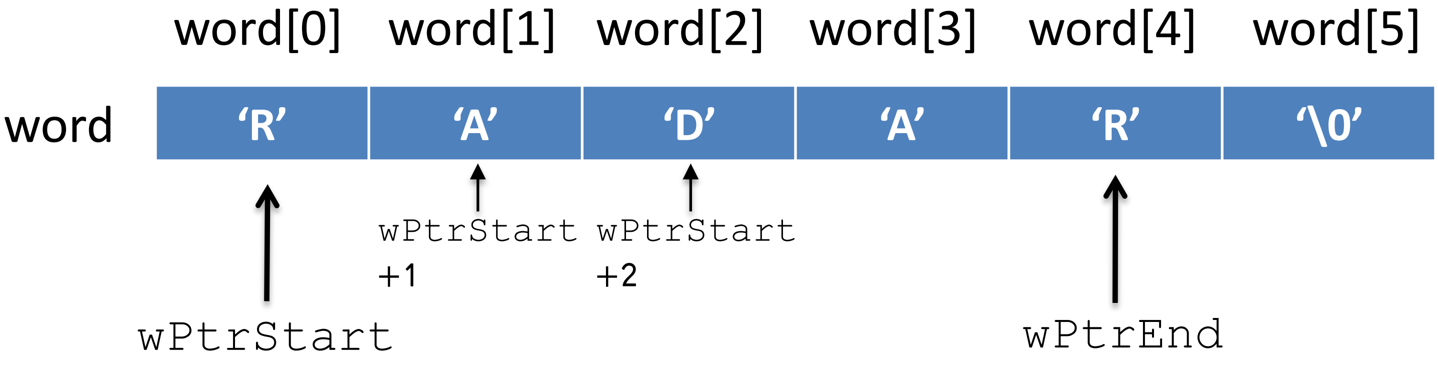
Main memory

0000	0000	0000	0001	arr[0]
0010	0010	0000	0011	arr[1]
0000	0000	0000	0101	arr[2]

Remember: an array is a set of elements of the same type allocated **contiguously** in memory!

Pointers and Arrays

palindrome.c



	word[0]	word[1]	word[2]	word[3]	word[4]	word[5]
word	‘R’	‘A’	‘D’	‘A’	‘R’	‘\0’

wPtrStart

wPtrStart +1

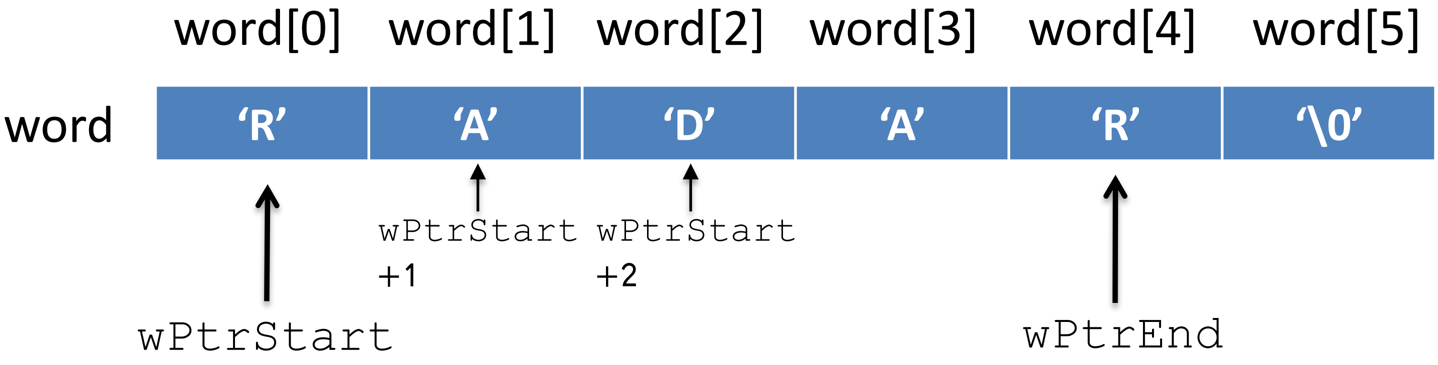
wPtrStart +2

wPtrEnd

```
char *wPtrStart = word;
char *wPtrEnd = wPtrStart + strlen(word)-1;

for( i=0 ; (i < strlen(word)/2) && (flag == 1) ; i++ ){

    if( *wPtrStart != *wPtrEnd ){
       flag = 0;
    }

    wPtrStart++;
    wPtrEnd--;
}
```

Pointers and Arrays

palindrome.c



	word[0]	word[1]	word[2]	word[3]	word[4]	word[5]
word	‘R’	‘A’	‘D’	‘A’	‘R’	‘\0’

wPtrStart

wPtrStart +1

wPtrStart +2

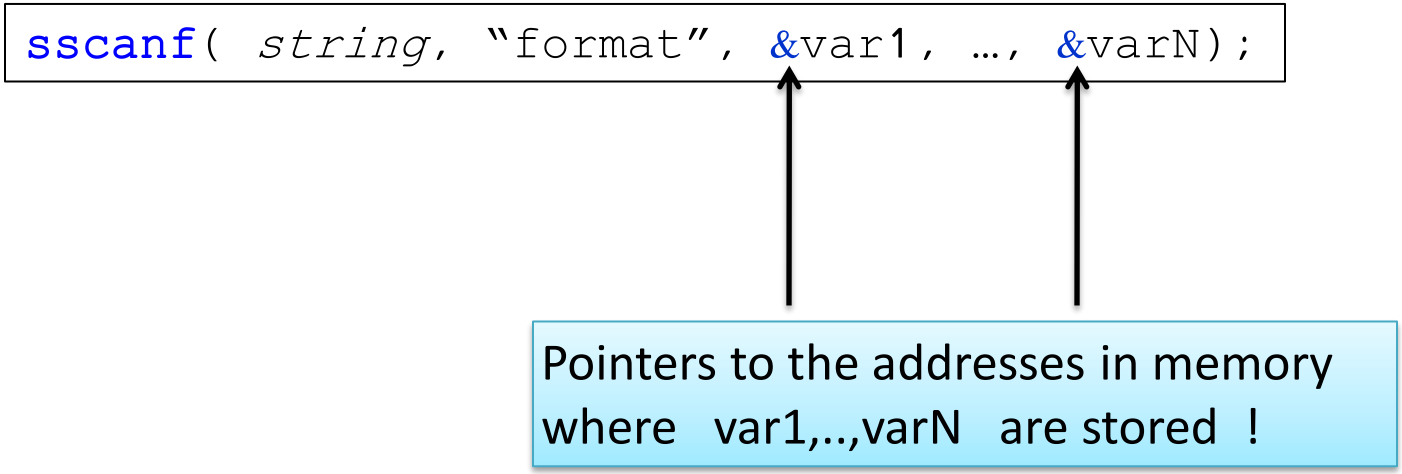
wPtrEnd

```
char *wPtrStart = word;
char *wPtrEnd = wPtrStart + strlen(word)-1;

for( i=0 ; (i < strlen(word)/2) && (flag == 1) ; i++ ){

    if( *wPtrStart != *wPtrEnd ){
       flag = 0;
    }

    wPtrStart++;
    wPtrEnd--;
}
```

When we increment or decrement, the pointers move by 1 byte (pointers to char)

Pointers : operators * and &

Now we know exactly what happens in sscanf !



```
sscanf( string, "format", &var1, …, &varN);
```

Pointers to the addresses in memory where var1,..,varN are stored !

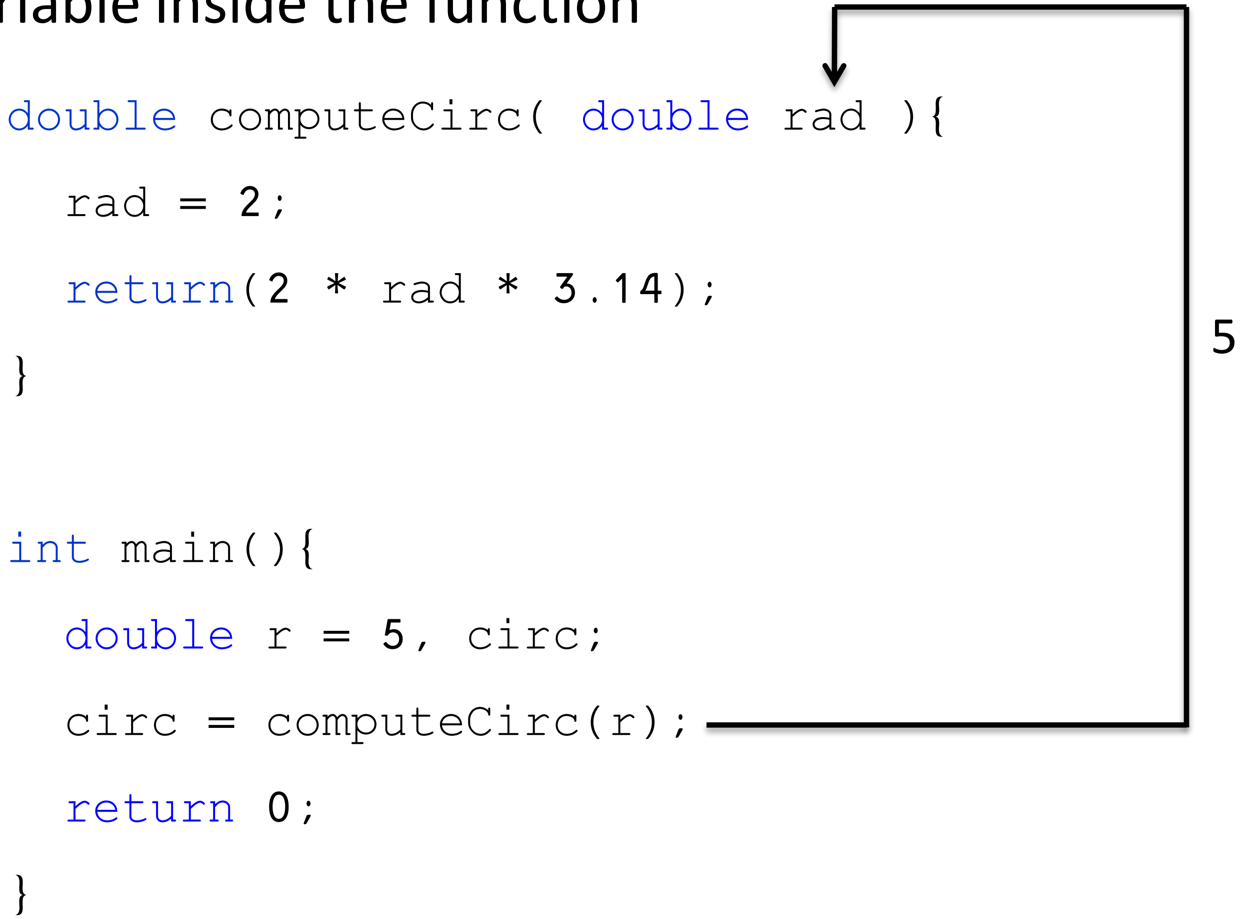
Functions
Passing arguments by value/reference

- Pass by value (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

- Pass by reference : a pointer to the variable used at invocation time is passed to the function. We can modify the variable’s value inside the function

Functions
Passing arguments by value/reference

- Pass by value (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function



```
double computeCirc( double rad ){
   rad = 2;
   return(2 * rad * 3.14);
}


int main(){
   double r = 5, circ;
   circ = computeCirc(r);
   return 0;
}
```

Functions
Passing arguments by value/reference

- Pass by value (what we have seen so far): the value of the variable used at invocation time is copied into a local variable inside the function

```
double computeCirc( double rad ){
  rad = 2;
  return(2 * rad * 3.14);
}

int main(){
  double r = 5, circ;
  circ = computeCirc(r);
  return 0;
}
```

r is not affected by anything we do inside the function

Functions
Passing arguments by value/reference

- Pass by reference : a pointer to the variable used at invocation time is passed to the function. We can modify the variable's value inside the function

```
double computeCirc( double *rad ){
   *rad = 2;
   return(2 * (*rad) * 3.14);
}

int main(){
   double r = 5, circ;
   circ = computeCirc(&r);
   return 0;
}
```

Address of r

Functions

circPointer.c

Passing arguments by value/reference

- Pass by reference : a pointer to the variable used at invocation time is passed to the function. We can modify the variable's value inside the function

```
double computeCirc( double *rad ){
  *rad = 2;
  return(2 * (*rad) * 3.14);
}

int main(){
  double r = 5, circ;
  circ = computeCirc(&r);
  return 0;
}
```

r has been modified!