Introduction to Pointers

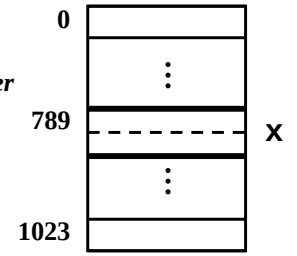
- **Pointer** - *a whole new kind of variable*
  - **scalar** variable: **single** value
  - **array** variable: **several** values (of same type)
  - **pointer** variable: **memory location** of a value
- pointers are incredibly *flexible/useful*
- unfortunately, pointers are also a bit *complicated* – takes a while to see their uses and benefits

Names and Addresses

- every variable has a location in memory. This memory location is uniquely determined by a **memory address**.
- use the **&** operator to find out the address of a variable – as already used in **`scanf`** statements
  - e.g. **`scanf("%d", &x);`**

x is the name the computer uses to refer to the value stored in the memory location (here, the integer entered by the user);

&x is 789 here, and refers to the address of the memory location



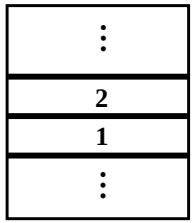
Suppose:
Memory size = 1024 bytes (obviously unrealistic!);
Integer stored in 2 bytes.

Note: address of memory location is the address of 1st byte of the location.

Names and Addresses: example

```
#include "stdio.h"
void main(void){
  int a=1, b=2;
  printf("a = %d; address of a: %u\n", a, &a);
  printf("b = %d; address of b: %u\n", b, &b);
}
```

Produces the screen output: **`a = 1; address of a: 65524`**
`b = 2; address of b: 65522`

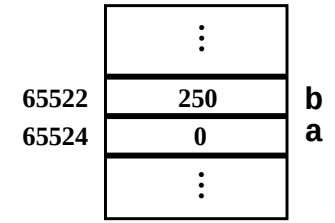


Note: locations used by the computer to store variables are system-dependent. We can find out where a variable is stored (using &), but we can't dictate where the variable should be stored, or move variables around memory. Also, here we assume 2 bytes are used to store an **`int`**

Names and Addresses: another example

```
#include "stdio.h"
void main(void){
  int a, b; /* a, b not initialised here */
  printf("a = %d; address of a: %u\n", a, &a);
  printf("b = %d; address of b: %u\n", b, &b);
}
```

Produces the screen output: **`a = 0; address of a: 65524`**
`b = 250; address of b: 65522`



Note: since we didn't initialise the variables, we get rubbish values in **a** ***and*** **b**. ***However, their addresses are the same as before.***

Pointers

- **A pointer is a datatype which stores addresses.**
  - Compare to: "an int is a datatype which stores integers"
- A particular pointer – say, **`ptr`** – is a variable which stores an address. We say that **`ptr`** "points to" the object whose address is stored in **`ptr`**
  - Every pointer has an associated datatype and is only allowed to store addresses of objects of that type.
    - therefore we have a *pointer-to-*`int`, a *pointer-to-*`float`, a *pointer-to-*`char`, etc.
  - Because a pointer is a variable:
    1. the address stored in the pointer can be changed;
    2. the pointer itself must be stored in some memory location, which must have an address. Some other pointer could therefore "point to" **`ptr`**

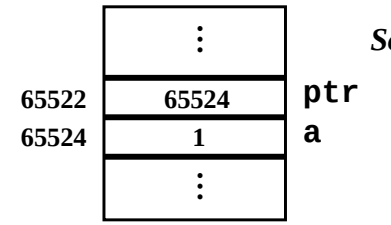
Pointers (contd.)

Declaring and initialising a pointer variable:

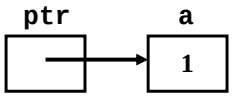
```
int a = 1;
int *ptr = &a; /* declare and initialise pointer */
```

Note: it is good practice to initialise a pointer variable when it is declared.

Also, by convention, pointer names begin with p, as in `ptr`



Sometimes shown as:



Equivalently:

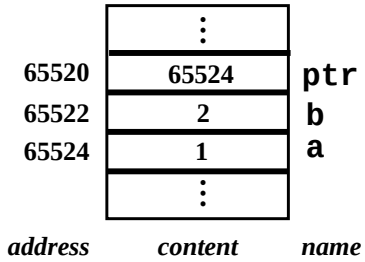
```
int a = 1;
int *ptr; /* declare pointer */
ptr = &a; /* assign address of a to pointer */
```


Pointers (contd.)

```
#include "stdio.h"
void main(void){
  int a=1, b=2, *ptr=&a;
  printf("a = %d; address of a: %u\n", a, &a);
  printf("b = %d; address of b: %u\n", b, &b);
  printf("ptr = %u; address of ptr: %u\n", ptr, &ptr);
  printf("ptr points to the value %d\n", *ptr);
}
```

Produces the screen output:

```
a = 1; address of a: 65524
b = 2; address of b: 65522
ptr = 65524; address of ptr: 65520
ptr points to the value 1
```

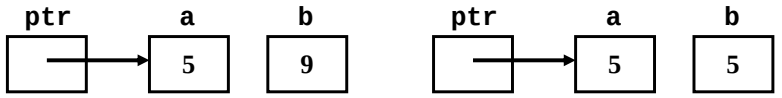


Note: except in a declaration, `*ptr` ***means "the value pointed to by*** `ptr`***". This is called dereferencing the pointer*** `ptr`

Pointers (contd.)

```
int a=5, b=9, *ptr=&a;
/* assign the value pointed to by ptr to b */
b = *ptr; /* *ptr is another name for a here */
```

Before `b=*ptr;` ***After*** `b=*ptr;`




```
int a=5, b=9, *ptr=&a;
/* assign value of b to whatever ptr points to */
*ptr = b; /* *ptr is another name for a here */
```

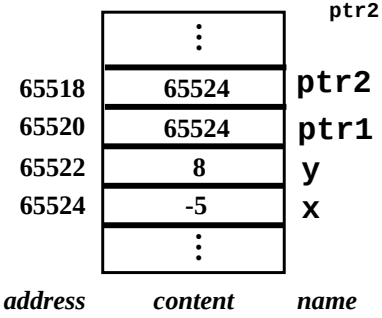
Before `*ptr=b;` ***After*** `*ptr=b;`

Pointers (contd.)

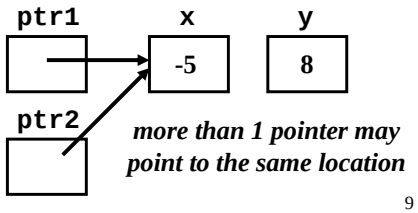
```
int x=-5, y=8, *ptr1, *ptr2;
ptr1=&x; /* ptr1 now points to x */
ptr2=ptr1; /* ptr2 now also points to x */
printf("x = %d; address of x: %u\n", x, &x);
printf("y = %d; address of y: %u\n", y, &y);
printf("ptr1 = %u; address of ptr1: %u\n", ptr1, &ptr1);
printf("ptr2 = %u; address of ptr2: %u\n", ptr2, &ptr2);
```

Produces the screen output:

```
x = -5; address of x: 65524
y = 8; address of y: 65522
ptr1 = 65524; address of ptr1: 65520
ptr2 = 65524; address of ptr2: 65518
```



Equivalently:



more than 1 pointer may point to the same location

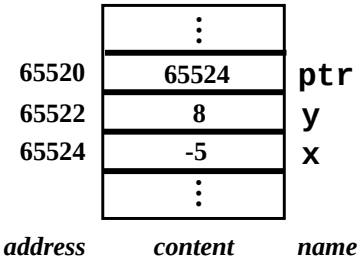
Pointers (contd.)

```
int x=-5, y=8, *ptr;
ptr=&x; /* ptr now points to x */
printf("x = %d; address of x: %u\n", x, &x);
printf("y = %d; address of y: %u\n", y, &y);
printf("ptr = %u; ptr points to the value %d\n", ptr, *ptr);
ptr=&y; /* ptr now points to y */
printf("now ptr = %u; ptr points to the value %d\n", ptr, *ptr);
```

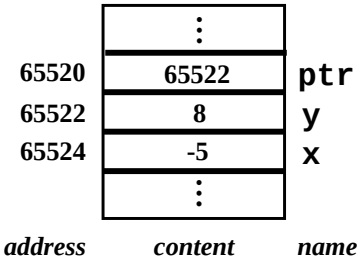
Produces the screen output:

```
x = -5; address of x: 65524
y = 8; address of y: 65522
ptr = 65524; ptr points to the value -5
now ptr = 65522; ptr points to the value 8
```

Initially:



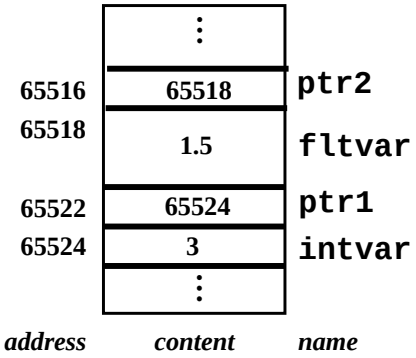
After* `ptr` *has been reassigned:



Pointers: example

```
int intvar=3, *ptr1=&intvar;
float fltvar=1.5, *ptr2=&fltvar;
```

Expression	holds	evaluates to
`intvar`	int	3
`&intvar`	address	65524
`ptr1`	address	65524
`&ptr1`	address	65522
`*ptr1`	int	3
`fltvar`	float	1.5
`&fltvar`	address	65518
`ptr2`	address	65518
`&ptr2`	address	65516
`*ptr2`	float	1.5



Note: here, pointer variables are stored in 2 bytes, regardless of the type of object they point to. Variables of type `int` are also stored in 2 bytes here; variables of type `float` are stored in 4 bytes here.

Pointers and arrays

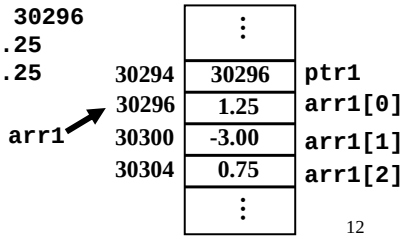
- In C there is a close relationship between pointers and arrays: the **name** of an array is a **pointer to the first element of the array**

```
float arr1[3] = {1.25, -3, 0.75};
float *ptr1 = &(arr1[0]);
printf("address of arr1's 1st element is %u\n", ptr1);
printf("address of arr1's 1st element is %u\n", arr1);
printf("value of arr1's 1st element is %.2f\n", *ptr1);
printf("value of arr1's 1st element is %.2f\n", *arr1);
```

Produces the screen output:

```
address of arr1's 1st element is 30296
address of arr1's 1st element is 30296
value of arr1's 1st element is 1.25
value of arr1's 1st element is 1.25
```

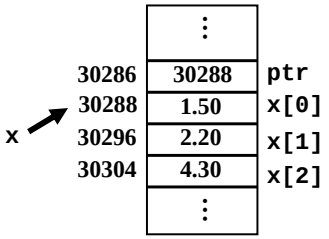
***An array name (e.g.* `arr1`*) is treated by the C compiler as a pointer to the first element of the array,* `arr1[0]`*. Therefore* `arr1 == &(arr1[0])` *and* `*arr1 == arr1[0]`**



Pointers and arrays: example

```
double x[3] = {1.5, 2.2, 4.3};
double *ptr = &x[0];
```

Expression	holds	evaluates to
x[0]	double	1.50
x[1]	double	2.20
x[2]	double	4.30
x	address	30288
&x[0]	address	30288
&x[1]	address	30296
&x[2]	address	30304
ptr	address	30288
&ptr	address	30286
*ptr	double	1.50
ptr+1	address	30296
*(ptr+1)	double	2.20
ptr+2	address	30304
*(ptr+2)	double	4.30



Note: since each `double` *is stored in 8 bytes here, and since* `ptr` *is a pointer to a* `double`*, the C compiler interprets* `ptr+1` *as "the address of the next* `double` *after the address in* `ptr`*", and* `ptr+2` *as "the address of the* `double` *2 locations away from the address in* `ptr`*", etc.*

Pointer arithmetic

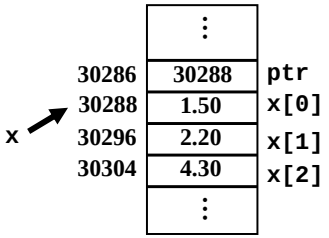
- From the preceding example, you can see that the C compiler interprets expressions of the form **`pointer+n`** in a different way to the usual arithmetic addition
  - First the C compiler determines, based on what datatype **`pointer`** is pointing to, what "address unit" to use in evaluating the expression **`pointer+n`**
    - If **`pointer`** is a ***pointer-to-double*** and doubles are stored in 8 bytes, the address unit is 8
    - If **`pointer`** is a ***pointer-to-int*** and ints are stored in 2 bytes, the address unit is 2
  - Then the address unit is multiplied by **`n`** to figure out what the **offset** is to the address currently stored in **`pointer`**
  - This offset is added to the address in **`pointer`** to yield the address referred to by **`pointer+n`**
- Main use of all this: another way to access array elements…

Pointers and arrays (contd.)

- Pointer notation to access array elements is often preferred

```
double x[3] = {1.5, 2.2, 4.3};
double *ptr = &x[0];
```

Expression	equivalent to	evaluates to
*ptr	x[0]	1.50
*(ptr+1)	x[1]	2.20
*(ptr+2)	x[2]	4.30
x	&x[0]	30288
x+1	&x[1]	30296
x+2	&x[2]	30304
*x	x[0]	1.50
*(x+1)	x[1]	2.20
*(x+2)	x[2]	4.30



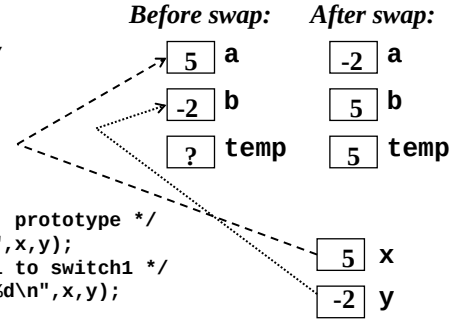
Key difference: a pointer is a <u>variable</u> and so can be reassigned; an array name "pointer" is <u>fixed</u> at compile time and can't be reassigned.

Pointers and functions

- Why are pointers useful when used with functions?
  - A: because they provide a way for a called function to access the calling function's actual parameters. For example, what if we wanted the called function to "return" more than 1 value?

<u>Incorrect swap program:</u> **`switch1()`** ***cannot change values of*** **`x`** ***and*** **`y`** ***in*** **`main()`**

```
#include <stdio.h>
void switch1(int a, int b){
  int temp; /* local variable */
  temp = a;
  a = b;
  b = temp;
} /* implicit return */
void main(void){
  int x=5, y=-2;
  void switch1(int, int); /* fn. prototype */
  printf("x is %d and y is %d\n",x,y);
  switch1(x,y); /* function call to switch1 */
  printf("now x is %d and y is %d\n",x,y);
}
```



<u>Produces the screen output:</u>

```
x is 5 and y is -2
now x is 5 and y is -2
```

Pointers and functions (contd.)

- Provide called function with **addresses** of actual parameter variables in the calling function, if you want called function to modify actual parameters:

Correct swap program:

```
#include <stdio.h>
void switch2(int *a, int *b){
  int temp;
  temp = *a; /* *a is x from main() */
  *a = *b; /* *b is y from main() */
  *b = temp;
}
void main(void){
  int x=5, y=-2;
  void switch2(int *, int *);
  printf("x is %d and y is %d\n",x,y);
  switch2(&x,&y);
  printf("now x is %d and y is %d\n",x,y);
}
```

Before swap:		*After swap:*	
&x	a	&x	a
&y	b	&y	b
?	temp	5	temp

Before swap:		*After swap:*	
5	x	-2	x
-2	y	5	y

Produces the screen output:
```
x is 5 and y is -2
now x is -2 and y is 5
```


Pointers and functions (contd.)

- Can also get the same effect by using pointers in the function call:

Another correct swap program:

```
#include <stdio.h>
void switch2(int *a, int *b){
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
void main(void){
  int x=5, y=-2, *ptr1=&x, *ptr2=&y;
  void switch2(int *, int *);
  printf("x is %d and y is %d\n",x,y);
  switch2(ptr1,ptr2); /* no need for & here */
  printf("now x is %d and y is %d\n",x,y);
}
```

Note: here, pointers `ptr1` *in* `main()` *and* `a` *in* `switch2()` *both point to* `x` *in* `main()`*; pointers* `ptr2` *in* `main()` *and* `b` *in* `switch2()` *both point to* `y` *in* `main()`

Produces the screen output:
```
x is 5 and y is -2
now x is -2 and y is 5
```


Pointers, functions and arrays

- Use `switch2()` to swap first and last elements in array in `main()`

```
#include <stdio.h>
void switch2(int *a, int *b){
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
void main(void){
int intarray[] = {5,7,9};
int *ptr = &intarray[0];
void switch2(int *, int *);
printf("1st element is %d, ", intarray[0]);
printf("last element is %d\n", intarray[2]);
switch2(&intarray[0],&intarray[2]);
printf("now 1st element is %d, ", intarray[0]);
printf("last element is %d\n", intarray[2]);
}
```

Note: instead of `switch2(&intarray[0],&intarray[2]);`
could use `switch2(intarray,intarray+2);`
or `switch2(ptr,ptr+2);`

Produces the screen output:
```
1st element is 5, last element is 9
now 1st element is 9, last element is 5
```


Pointers, functions and arrays (contd.)

- Recall: arrays are passed by address. For example:

```
#include <stdio.h>
void zap(int array1[], int size) { /* knows address of charges[] */
  int i;
  for (i = 0; i < size; i++) {
     array1[i] = 0; /* sets all elements to 0 */
  }
}
void main(void) {
  int i, charges[5]={7,8,7,6,7};
  printf("elements of charges[] are ");
  for (i=0;i<5;i++){printf("%d ", charges[i]);}
  zap(charges, 5); /* pass address of 1st element of charges[] */
  printf("\nnow elements of charges[] are ");
  for (i=0;i<5;i++){printf("%d ", charges[i]);}
}
```

`array1[]` in `zap()` is another name for `charges[]` in `main()`

Produces the screen output:
```
elements of charges[] are 7 8 7 6 7
now elements of charges[] are 0 0 0 0 0
```

Pointers: example

What is the screen output of this program?

```
#include <stdio.h>
void main(void){
double a = 0.5;
double *ptr = &a;
a = a + 1.0;
*ptr = a + 3.0;
*ptr = (*ptr)*10.0;
printf("final value of a is %.2f\n", a);
}
```


Pointers: another example

What is the screen output of this program?

```
#include <stdio.h>
int func(int x, int *y){
  x = 6;
  *y = *y + 2;
  return x;
}
void main(void){
  int a=5, b=7, c;
  int func(int, int *);
  c = func(a, &b);
  printf("a is %d, b is %d, c is %d\n",a,b,c);
}
```


A pointer to a pointer

```
#include <stdio.h>
void main(void){
int a=7;
int *pInt = &a;
int **pInt2 = &pInt;
/* pInt2 is a pointer to a pointer-to-int */
printf("a is %d, address of a is %u\n", a,&a);
printf("pInt is %u, address of pInt is %u\n", pInt,&pInt);
printf("pInt points to %d\n", *pInt);
printf("pInt2 is %u, address of pInt2 is %u\n", pInt2,&pInt2);
printf("pInt2 points to %u\n", *pInt2);
printf("pInt2 doubly points to %d\n", **pInt2);
}
```

Produces the screen output:

```
a is 7, address of a is 30308
pInt is 30308, address of pInt is 30304
pInt points to 7
pInt2 is 30304, address of pInt2 is 30300
pInt2 points to 30308
pInt2 doubly points to 7
```