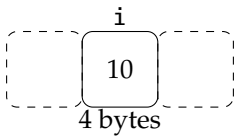
Memory Access using Pointers

1 Implementing variables

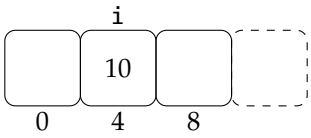
Definition: Variable

A logical name for an allocated area of memory assigned to store a value of a certain type

```
int i = 10;
```



2 Pointer variables

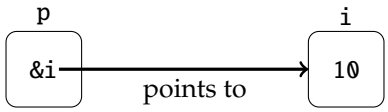


- `int i = 10;`
- value of `i` is `10`
- The memory address of `i` is `&i` and has a value of `4`
- A pointer variable stores a memory address:

```
int *p;
p = &i;
```

- now the pointer variable `p` stores the memory address of the integer variable `i`

3 Pointers




```
int i = 10;   // simple variable
int *p = &i;  // pointer variable
```

- You can read the address operator `&` as *address of*

```
printf("%d %d\n", i, *p ) ;
```

- This outputs `10 10`
- You can read the indirection operator `*` as *value of*

4 Basic pointer operations

```
int i = 5; // declare an int variable
int *p;    // declare a variable pointer to an int

p = &i;    // & "address of"
```

- Use indirection operator * to access and modify the value:

```
*p = 7;      // assign value of 7 to i

*p = *p + 1; // add 1 to value of i
```

5 The Indirection Operator: what not to do

- Applying the indirection operator to an uninitialized pointer variable causes undefined behaviour:

```
int *p;
printf("%d", *p);   /*** WRONG ***/
```

- Assigning a value to *p is particularly dangerous:

```
int *p;
*p = 1;   /*** DANGER ***/
```

6 Pointer Assignment

- C allows the use of the assignment operator to copy pointers of the same type
- Assume that the following declaration is in effect:

```
int i, j, *p, *q;
```

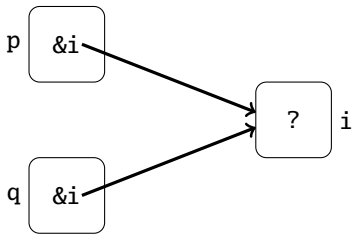
- Example of pointer assignment:

```
p = &i;
```

- Another example of pointer assignment:

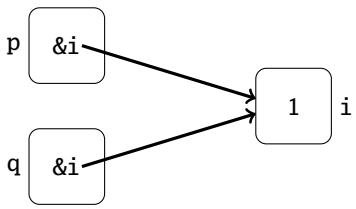
```
q = p;
```

- q now points to the same place as p:

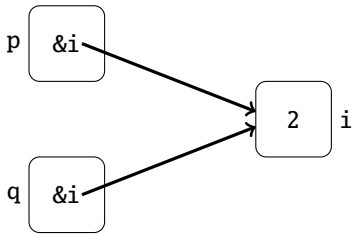


- If p and q both point to i, we can change i by assigning a new value to either *p or *q:

```
*p = 1;
```



```
*q = 2;
```



- Any number of pointer variables may point to the same object

7 Pointers as Arguments

- Previously we tried write a `swap()` function that could modify its arguments, but it didn't work
- By passing a pointer to a variable instead of the value of the variable, `swap()` can be fixed

8 Swap

- We want to write a simple function in C to swap the values of two integer variables, `x` and `y`

```
void swap(int a, int b) {
  int temp;

  temp = a;
  a = b;
  b = temp;
}
```

- Then call `swap(x,y);`
- Does this work?

8.1 A working solution

```
void swap(int *a, int *b) {
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

- Then call `swap(&x,&y);`
- Remember C uses call by value, but by using pointers you can get around this problem as the pointer still points at the original data

9 Pointers as Arguments

- Arguments in calls of `scanf()` are pointers:

```
  int i;
...
  scanf("%d", &i);
```

- Without the `&`, `scanf()` would be supplied with the value of `i`
- Although `scanf()`'s arguments must be pointers, it's not always true that every argument needs the `&` operator:

```
  int i, *p;
...
  p = &i;
  scanf("%d", p);
```

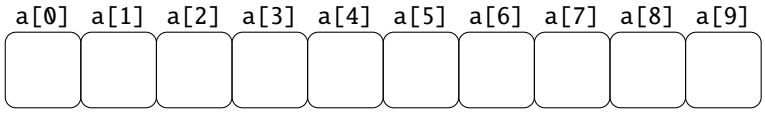
- Using the & operator in the call would be wrong:

```
scanf("%d", &p);   /*** WRONG, would be the address of the pointer,
 rather than the address of the thing p points at ***/
```

10 Arrays in C

```
int a[10];
```

- declares a fixed size array holding ten `int` values



- `a[i]` is the `i`th element of the array
- `sizeof(a) = 10 * sizeof(int) = 40` bytes
- The array is stored in memory as a single contiguous block that is 40 bytes (10 `int`s) in size
- Note that `sizeof(a) / sizeof(a[0])=10`
  - This is a common way of checking the number of elements in an array.
  - We can't pass an array to a function, but we can pass a pointer to it. The line above will not work correctly on a pointer, so we will need to pass the length of the array too.

11 Strings

- Are represented as an array of characters

```
char a[] = "Hello worlds";
char b[13];
b = a; // Not allowed
char *c;
c = a;
```

- will set pointer `c` to same address as `a`
- assignment of an array to array is not supported in C
- unlike `struct` as we saw last lecture
- `strcpy(b,a);` first argument is the destination, ordered like assignment above
  - need to `#include<string.h>`

```
char a[] = "Hello";
strlen(a) = 5; // number of characters
sizeof(a) = 6; // includes the null character at the end
```

- Strings are null terminated – important when allocating space to store them

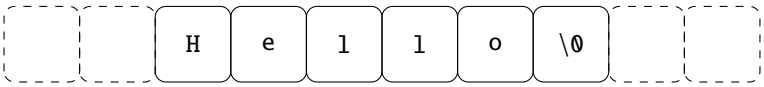
```
printf("%s %c\n", a, a[0]);
```

- Output is:

```
Hello H
```

- If you use the variable a on its own, it represents the memory address of the start of the string

12 Pointers, strings and arrays




```c
char a[] = "Hello"; // sizeof = 6
char *a = "Hello"; // sizeof = 8  as pointers take 8 bytes
```

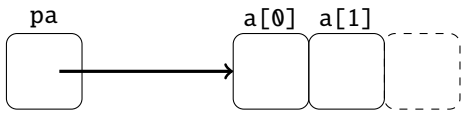
- These are equivalent declarations, and create the identical bytes in memory, as shown above.
- *Warning:* using `sizeof(a)` will give `6` in the first case (the size of the array) and `8` in the second case (the size of the pointer).
- In the second case, the string `"Hello"` is constant and cannot be modified.
- Pointers and arrays are often used interchangeably

13 Pointer arithmetic

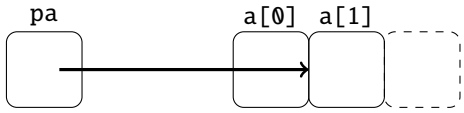
- Pointer arithmetic accounts for the base type of the items:

```c
int a[10];
int *pa;

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




```c
pa = &a[1];
pa = (a+1);
```



- The two pairs of statements above are equivalent using array or pointer notation: `+1` translates to `+4` bytes (1 `int`)

14 Strange but true

- In C if I write `a[x]` this works by adding `x` to `a` to find the pointer
- Hence `a[x]` is the same as `*(a+x)`
- This seems fine if I write `a[2]`
- But what if I write `2[a]`?
- It compiles and works!
- See `array.c`

15 What about?

- `a[-4]` ?
- Interpreted as `*(a + -4)`
- Is the following valid?

```c
int *p;
int i = 5;
int j = 20;
p = &i;
printf("%d %d\n", p[0], p[1]);
```

- What will the output be?

16 Peeking at memory

- Can look at bits of memory
- See `peek.c`
- Can find adjacent local variables and parameters
- Easy to make mistakes
- Cannot tell what data is by looking at it

17 Breaking things

- We can use random numbers to write random values in random places
- See `break.c`
- This can upset the system
- Segmentation fault occurs: hardware tells OS a memory access is not allowed
- Sometimes it goes on for a shockingly long time
- Sometimes the last number is very strange: why?