Linked Lists -- an introduction

New problem: arrays are *not* dynamically sized.

So, if a program cannot predict the size of a needed array, consider a **linked list**.
Assume (for simplicity) that the item to go into the list is an `int`.

Start with an empty list.

```
listadd(1)    1
listadd(2)    2 ... 1
listadd(3)    3 ... 2 ... 1
listadd(4)    4 ... 3 ... 2 ... 1
```
“Knowing” where the next item in the list is is simple -- it is a pointer.

We need to associate each item in the list with a pointer.
Set up a struct with 2 fields:

\[
\begin{align*}
\text{int} \\
\text{pointer to a struct}
\end{align*}
\]

(often called a \text{node})

After adding all 4 ints to the example list:

\[
\begin{align*}
4 & \rightarrow 3 & \rightarrow 2 & \rightarrow 1 \\
\end{align*}
\]
struct node {
    int theint;
    struct node *next;
};
SINGLY LINKED, BUT IN THE REVERSE ORDER (ADD TO END OR BACK OF THE LIST)
struct node {
    int theint;
    struct node *next;
    struct node *previous;
};
For convenience, name this user-defined type:

```c
typedef struct node {
    int theint;
    struct node *next;
} Node;
```

Now, declarations have less (keyboard) typing:

```c
Node one, two, three;
Node *head;
```
Some code, to show pointers and such. . .

```c
one.theint = 1;
one.next = &two;
one.next->next = &three;
three.next = NULL;
head = &one;
```

```
head
```

```
one
two
three
```
int value = 1;
Node *ptr;

ptr = head;
while (ptr != NULL) {
    ptr->theint = value * 11;
    value++;
    ptr = ptr->next;
}

int value = 1;
Node *ptr;

ptr = head;
while (ptr != NULL) {
    ptr->theint = value * 11;
    value++;
    ptr = ptr.next;
}

Why is this now incorrect?
With the correct code, what happens when this code is executed?

```c
ptr = three.next;
ptr = ptr->next;
```