



# CS 11 C track: lecture 6

---

- Last week: pointer arithmetic
- This week:
  - The `gdb` program
  - `struct`
  - `typedef`
  - linked lists



# `gdb` for debugging (1)

---

- `gdb`: the Gnu DeBugger
  - <http://courses.cms.caltech.edu/cs11/material/c/mike/misc/gdb.html>
- Use when program crashes
  - *e.g.* from a segmentation violation
- or when want to walk through execution of program line-by-line



## `gdb` for debugging (2)

---

- Before using `gdb`:
  - Must compile C code with additional flag: `-g`
  - This puts all the source code into the binary executable
- Then can execute as: `gdb myprogram`
- Brings up an interpreted environment



## gdb for debugging (3)

---

```
gdb> run
```

- Program runs...
- If all is well, program exits successfully, returning you to prompt
- If there is (e.g.) a crash, **gdb** will tell you and abort the program



## `gdb` for debugging (4)

---

- If your program needs command-line arguments, e.g. `myprogram 1 2 3`, then you should do this in `gdb`:

```
gdb> run 1 2 3
```

- This will run `myprogram` with the command-line arguments `1`, `2`, and `3`



# `gdb` – basic commands (1)

---

- Stack backtrace ("**where**")
  - Your program crashes
  - Where was the last line in the program that was executed before the crash?
  - That's what the **where** command tells you

# gdb – basic commands (2)

`gdb> where`

last call

last call in your code

#0 0x4006cb26 in free () from /lib/libc.so.6

#1 0x4006ca0d in free () from /lib/libc.so.6

#2 0x8048951 in board\_updater (array=0x8049bd0,  
ncells=2) at 1dCA2.c:148

#3 0x80486be in main (argc=3, argv=0xbffff7b4) at  
1dCA2.c:44

#4 0x40035a52 in \_\_libc\_start\_main () from  
/lib/libc.so.6

stack backtrace



## `gdb` – basic commands (3)

---

- Look for topmost location in stack backtrace that corresponds to your code
- Watch out for
  - freeing memory you didn't allocate
  - accessing arrays beyond their maximum elements
  - dereferencing pointers that don't point to part of a `malloc()` ed block





## **gdb – basic commands (4)**

---

- **break**, **continue**, **next**, **step** commands
- **break** causes execution to stop on a given line  
`gdb> break foo.c: 100` (setting a breakpoint)
- **continue** resumes execution from that point
- **next** executes the next line, then stops
- **step** executes the next statement
  - goes into functions if necessary (**next** doesn't)



## `gdb` – basic commands (5)

---

- `print` and `display` commands
- `print` prints the value of any program expression

```
gdb> print i
```

```
$1 = 100
```

- `display` prints a particular value every time execution stops

```
gdb> display i
```



# gdb – printing arrays (1)

---

- `print` will print arrays as well

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

```
gdb> print arr
```

```
$1 = {1, 2, 3}
```

- N.B. the `$1` is just a name for the result

```
print $1
```

```
$2 = {1, 2, 3}
```



## gdb – printing arrays (2)

---

- `print` has problems with dynamically-allocated arrays

```
int *arr;
```

```
arr = (int *)malloc(3 * sizeof(int));
```

```
arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
gdb> print arr
```

```
$1 = (int *) 0x8094610
```

- Not very useful...



## gdb – printing arrays (3)

---

- Can print this array by using @ (gdb special syntax)

```
int *arr;
```

```
arr = (int *)malloc(3 * sizeof(int));
```

```
arr[0] = 1; arr[1] = 2; arr[2] = 3;
```

```
gdb> print *arr@3
```

```
$2 = {1, 2, 3}
```



# `gdb` – abbreviations

---

- Common `gdb` commands have abbreviations

`p` (same as `print`)

`c` (same as `continue`)

`n` (same as `next`)

`s` (same as `step`)

- More convenient to use when interactively debugging



# structs (1)

---

- Way to package primitive data objects into an aggregate data object
- **struct** declaration:

```
struct point {  
    int x;  
    int y;  
    double dist; /* from origin */  
}; /* MUST have semicolon! */
```



## structs (2)

---

- **struct** declaration usually done outside of function, like a function prototype
- Create/initialize **struct** like this:

```
struct point p;  
p.x = 0;    /* "dot syntax" */  
p.y = 0;  
p.dist = sqrt(p.x*p.x + p.y*p.y);
```





# structs (3)

---

- Using a **struct**:

```
void foo(void) {  
    struct point p;  
  
    p.x = 10; p.y = -3;  
  
    p.dist = sqrt(p.x*p.x + p.y*p.y);  
  
    /* do stuff with p */  
  
}
```



## structs (4)

---

- Using `malloc()` with structs:

```
struct point *make_point(void) {  
    struct point *p;  
    p = (struct point *)  
        malloc(sizeof(struct point));  
    return p;  
} /* free struct elsewhere */
```



# structs (5)

---

- Using pointers to structs :

```
void init_point(struct point *p) {  
    (*p).x = (*p).y = 0;  
    (*p).dist = 0.0;  
    /* syntactic sugar: */  
    p->x = p->y = 0;  
    p->dist = 0.0;  
}
```



## structs (6)

---

- structs can contain arrays or other structs
- Usually use pointers to structs instead of just plain structs

```
struct foo {  
    int x;  
    struct point p1;    /* Unusual */  
    struct point *p2;  /* Typical */  
};
```



## structs (7)

---

- structs can be "recursive":

```
struct node {  
    int value;  
    struct node *next;  
};
```

- but can't have `struct node next` inside declaration (why?)



# typedef (1)

---

- Typing `struct point` all the time is tedious
- Use a `typedef` (type alias):

`typedef`                      original type                      new name  
`typedef` `struct point` `Point`;

`typedef int Length;`

- Original type comes first
- New name is at the end



## typedef (2)

---

- Type component of `typedef` can also be a struct

```
typedef struct { /* no name for struct */
    int x;
    int y;
    double dist;
} Point;

Point p1, p2; /* no "struct" */
```

- N.B. This is an *anonymous* struct



# typedef (3)

---

- Recursively defined structs:

```
typedef struct _node {  
    int value;  
    struct _node *next;  
} node;
```





# typedef (4)

---

- Read this as:

**typedef**

```
    struct _node {  
        int value;  
        struct _node *next;  
    }
```

**node;**



# Linked lists

---

- **node** is the linked list struct!
- Set **next** pointer to next node in list
- If **next** is **NULL**, then at end of list
- Linked lists are just chains of **nodes**



# Creating a linked list (1)

---

```
node *list, *n, *prev;
```



# Linked list (diagram)

---

list

n

prev



## Creating a linked list (2)

---

```
n = (node *)malloc(sizeof(node));  
list = n; /* list points to first node */  
n->value = 10;  
prev = n; /* pointer to previous node */
```



# Linked list (diagram)

---

list

n

prev



# Linked list (diagram)

---

list

n →



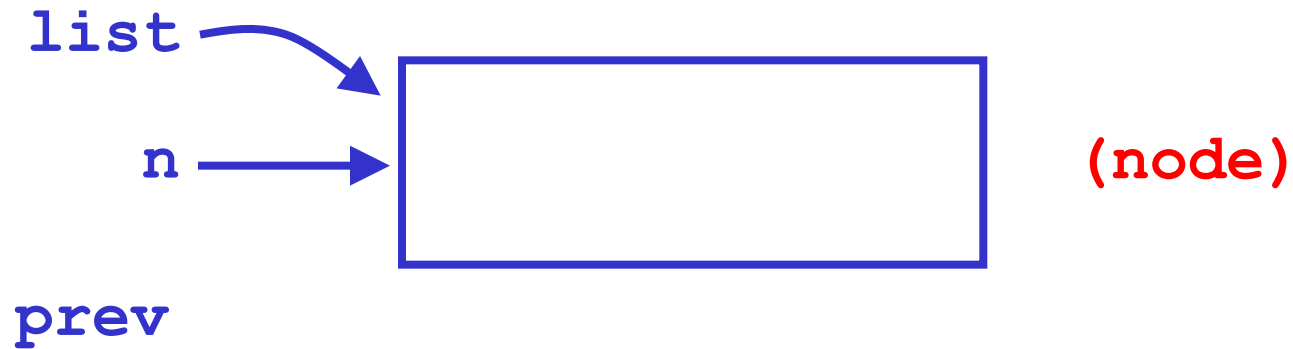
(node)

prev



# Linked list (diagram)

---

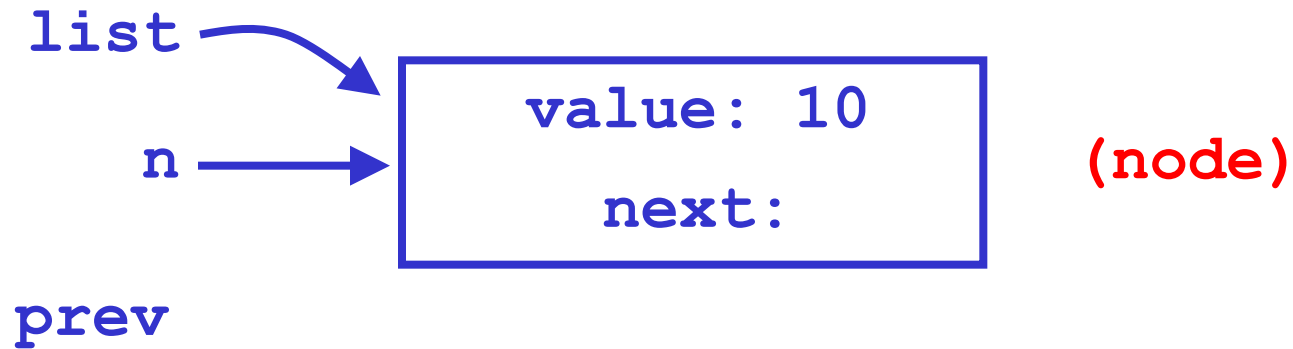






# Linked list (diagram)

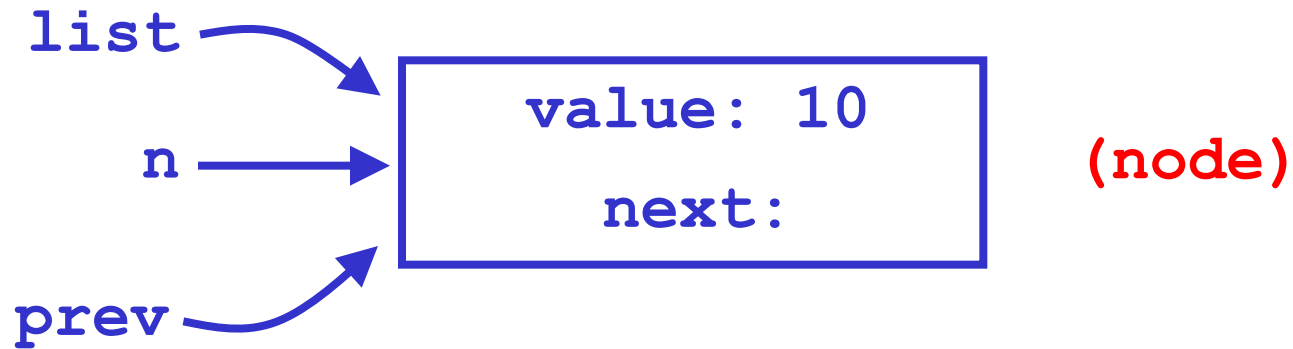
---





# Linked list (diagram)

---





## Creating a linked list (3)

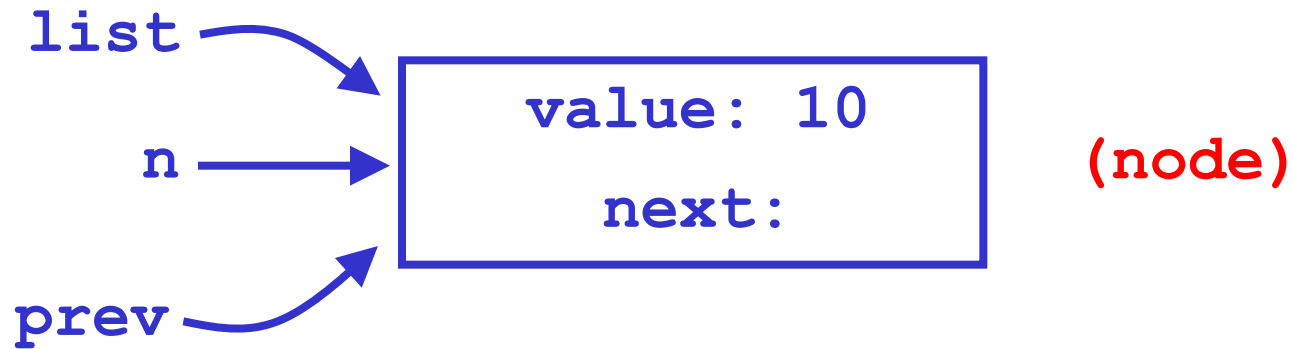
---

```
n = (node *)malloc(sizeof(node));  
prev->next = n; /* connect nodes */  
prev = n;  
n->value = 20;  
/* ... continued on next slide ... */
```



# Linked list (diagram)

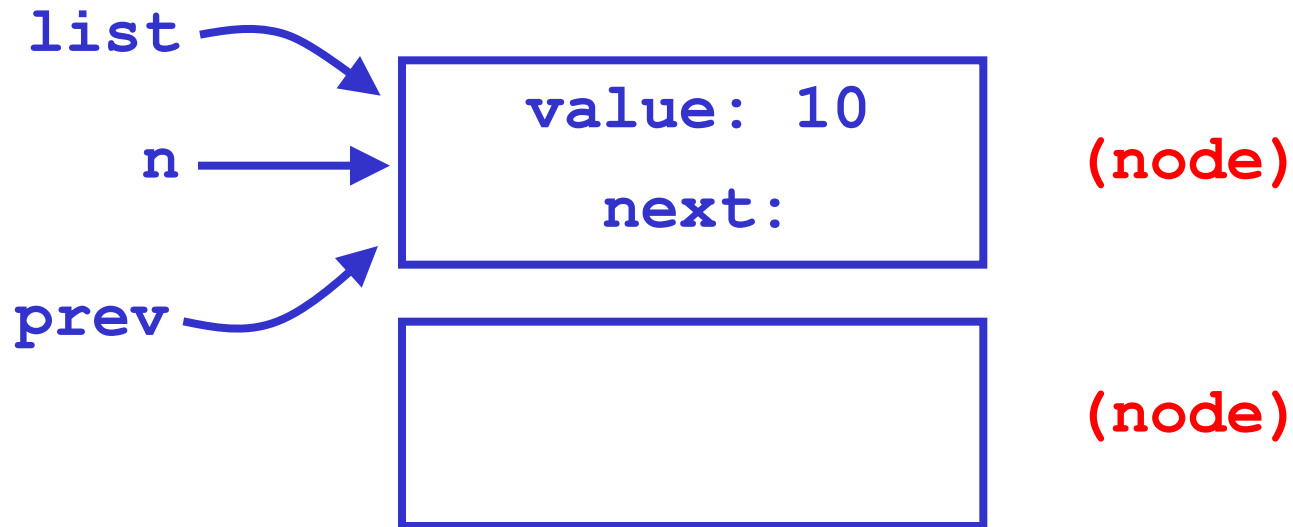
---





# Linked list (diagram)

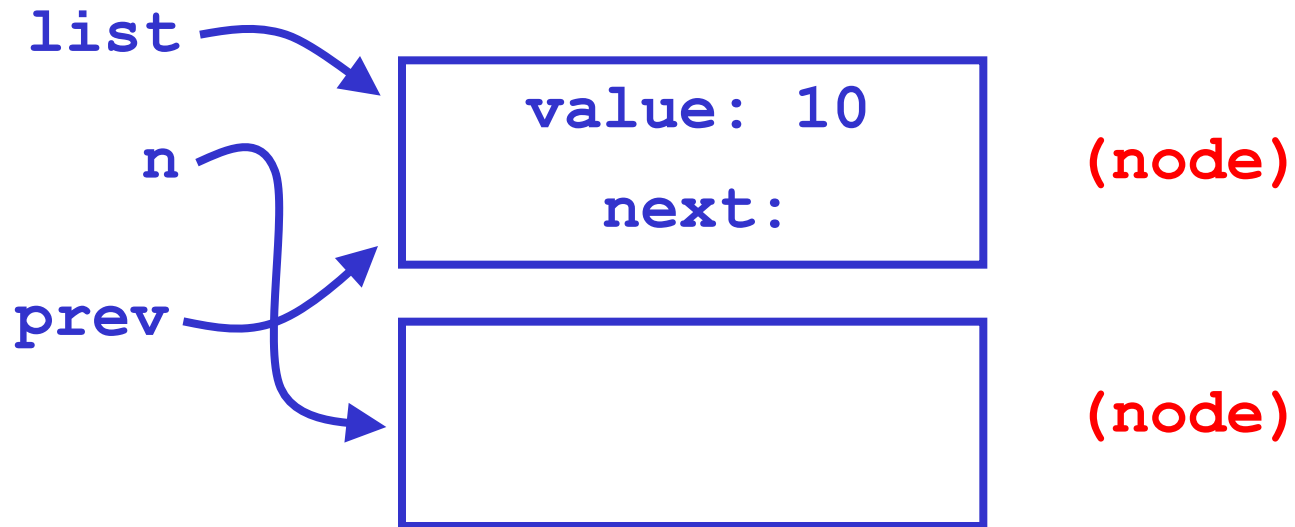
---



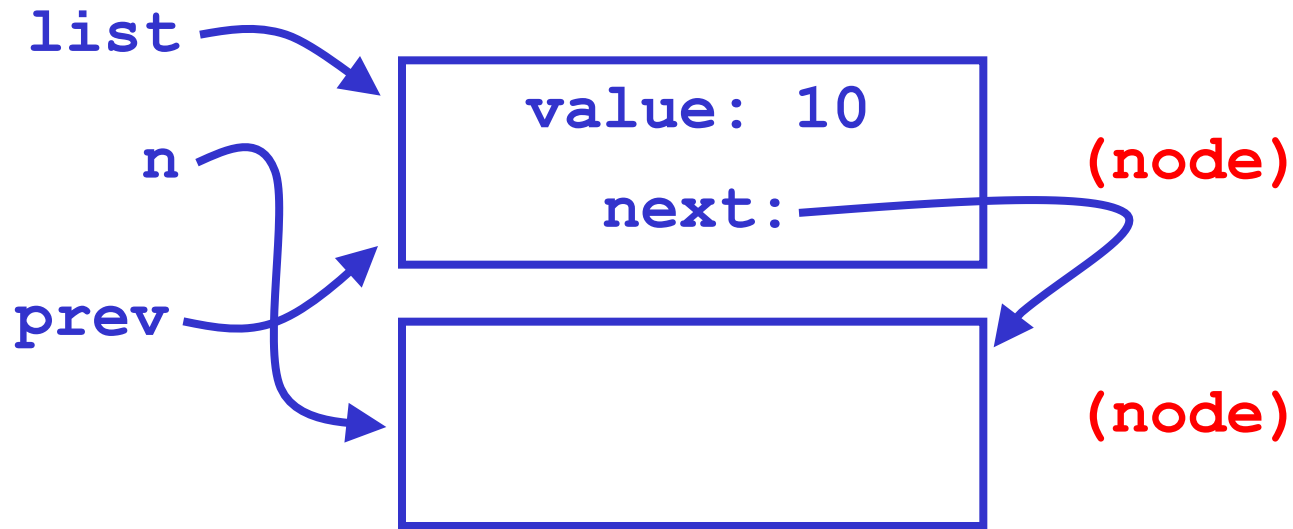


# Linked list (diagram)

---



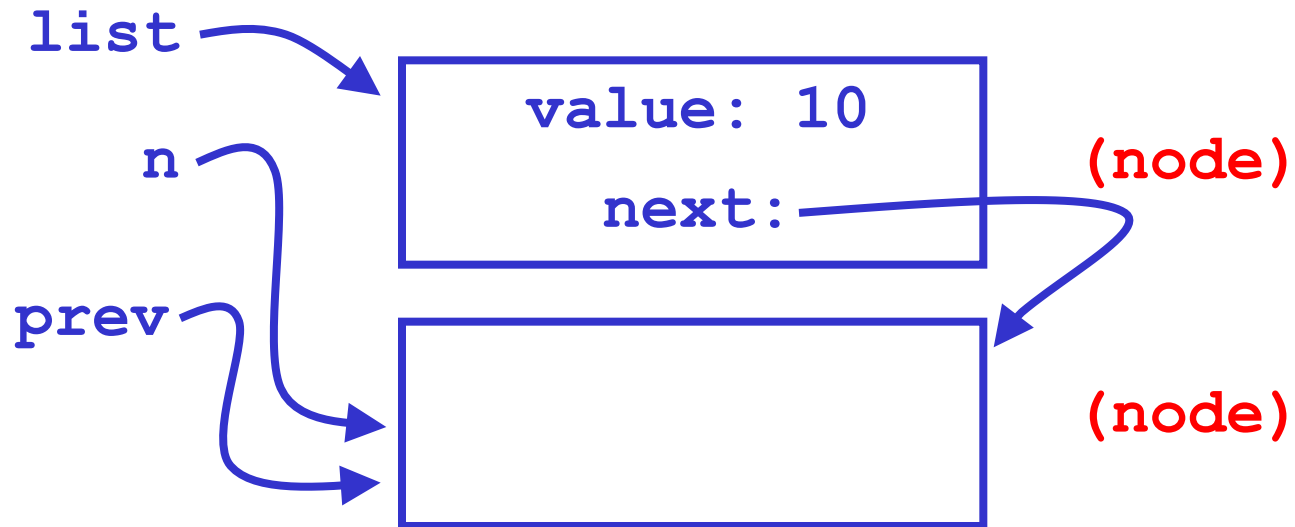
# Linked list (diagram)





# Linked list (diagram)

---

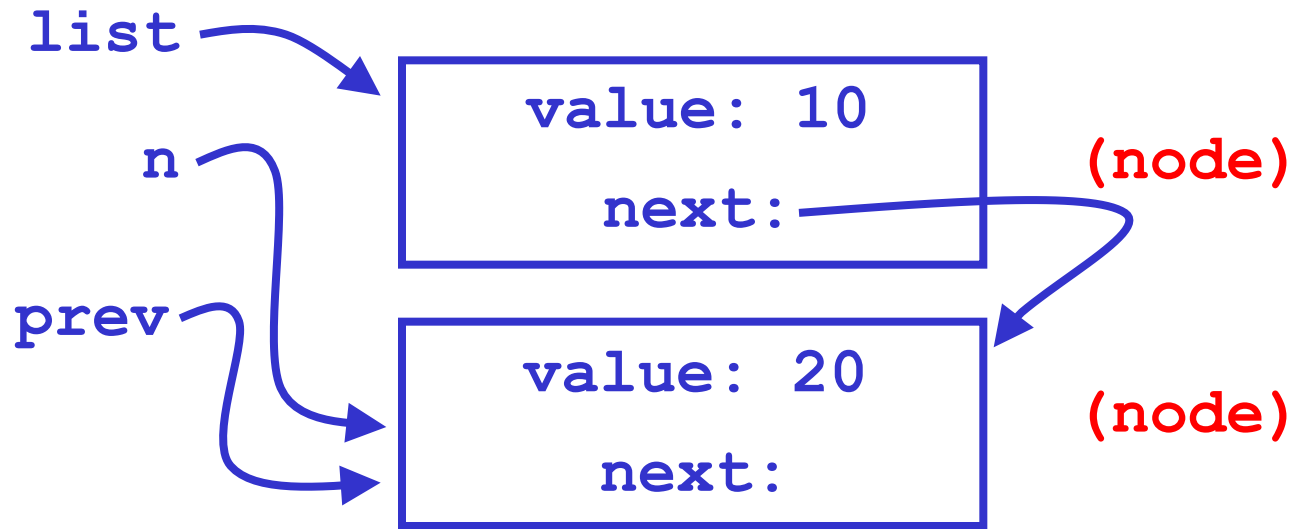






# Linked list (diagram)

---





## Creating a linked list (4)

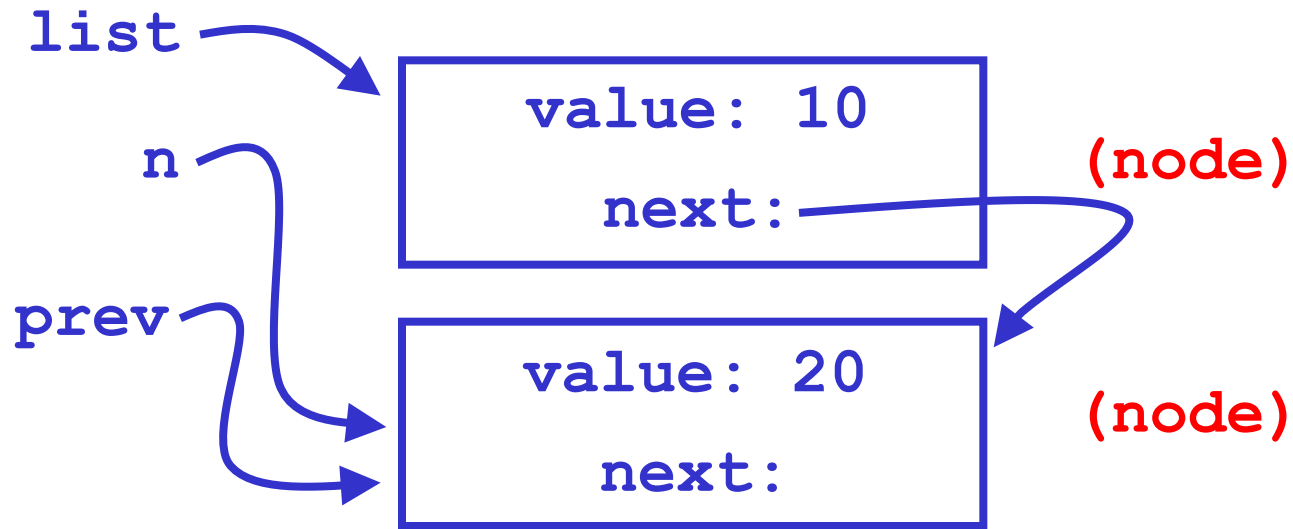
---

```
/* Continued... */  
n = (node *) malloc(sizeof(node));  
prev->next = n;  
prev = n;  
n->value = 30;  
n->next = NULL; /* End of list marker. */
```



# Linked list (diagram)

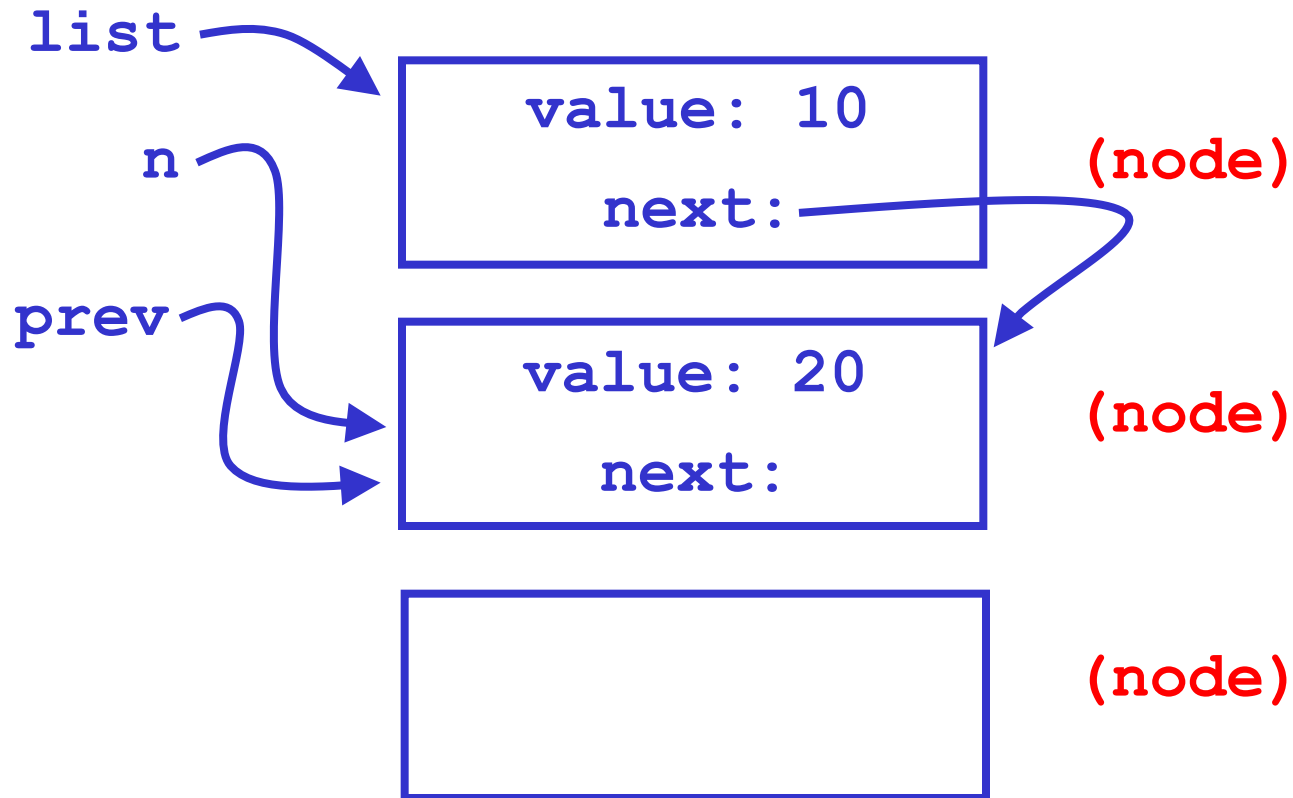
---





# Linked list (diagram)

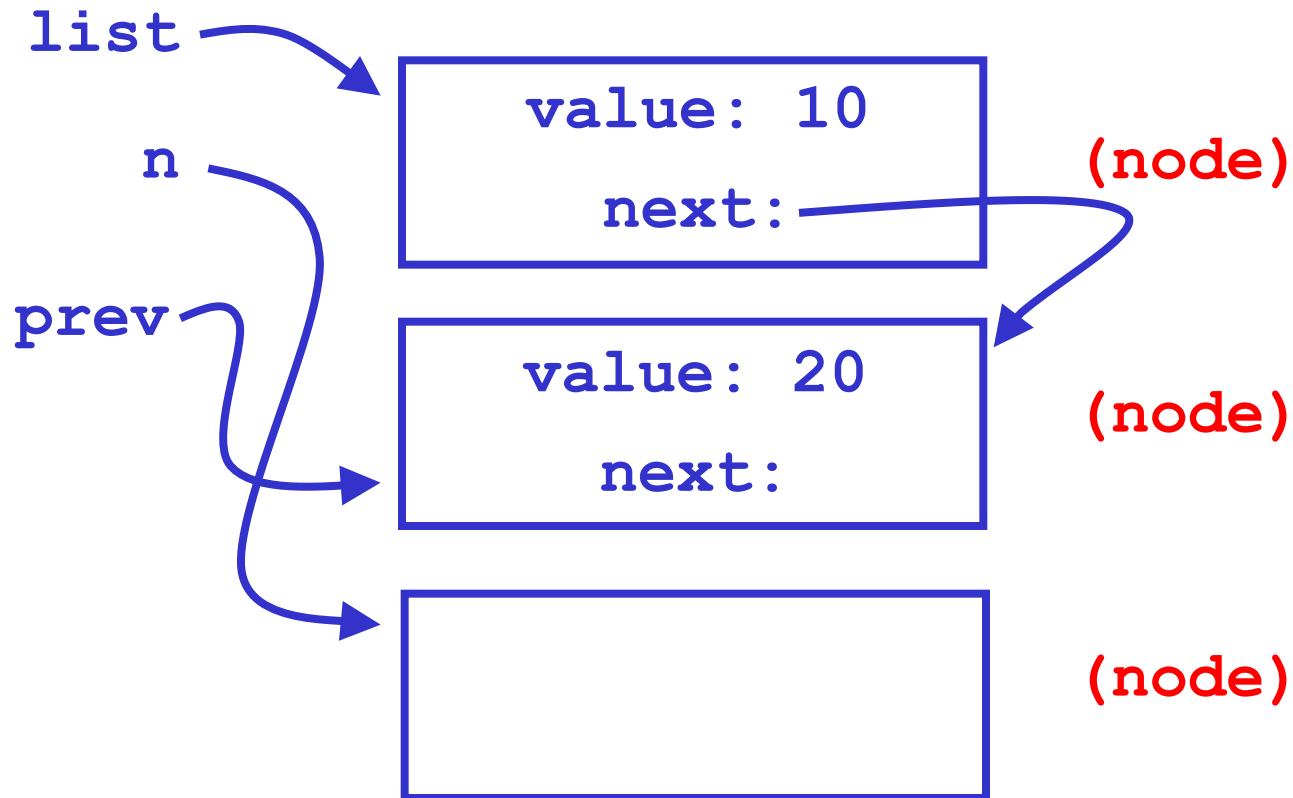
---





# Linked list (diagram)

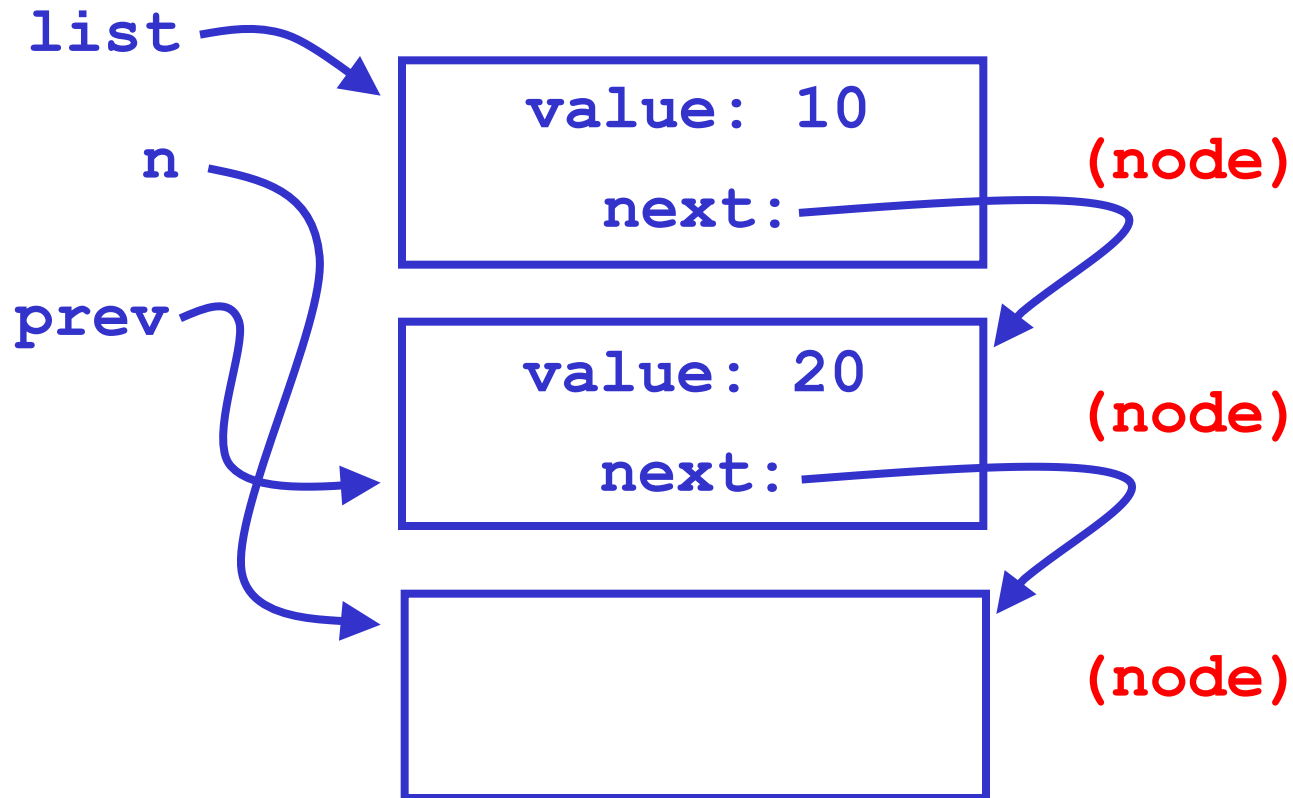
---





# Linked list (diagram)

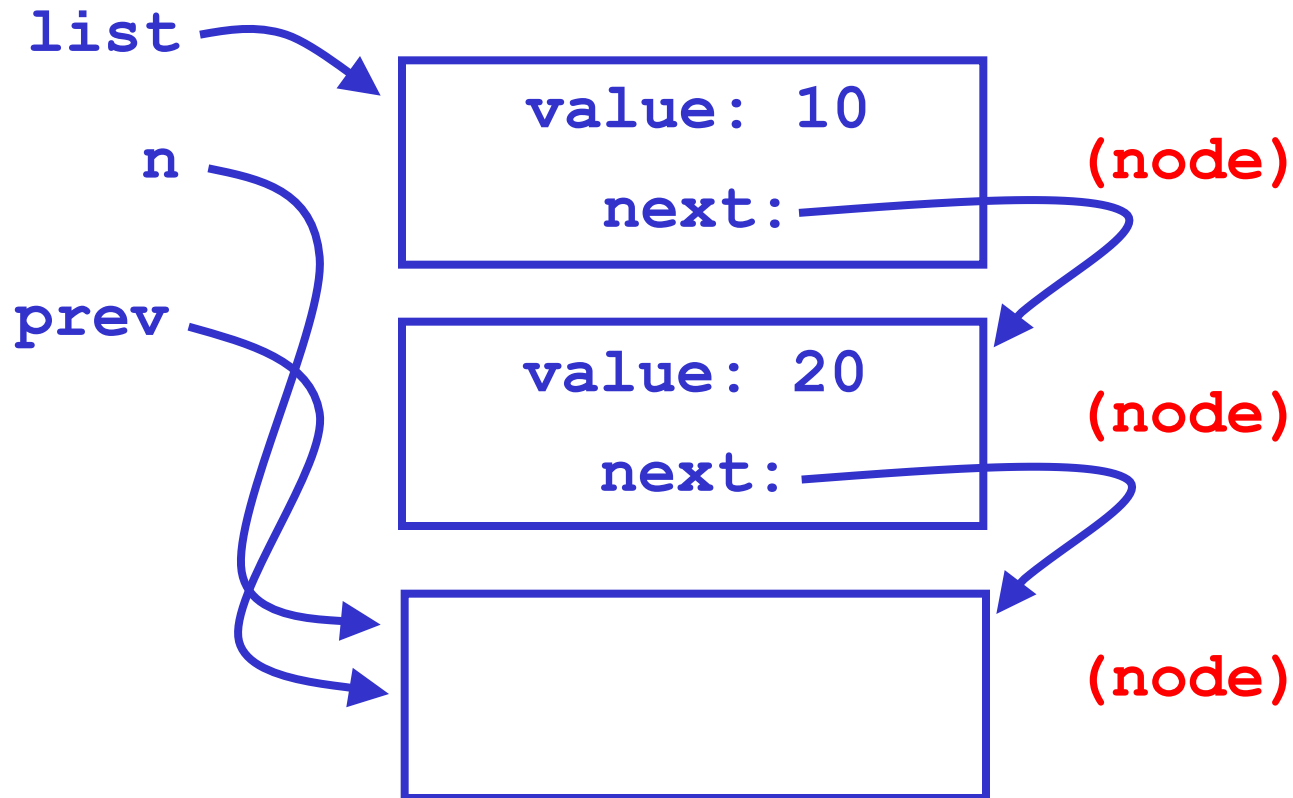
---





# Linked list (diagram)

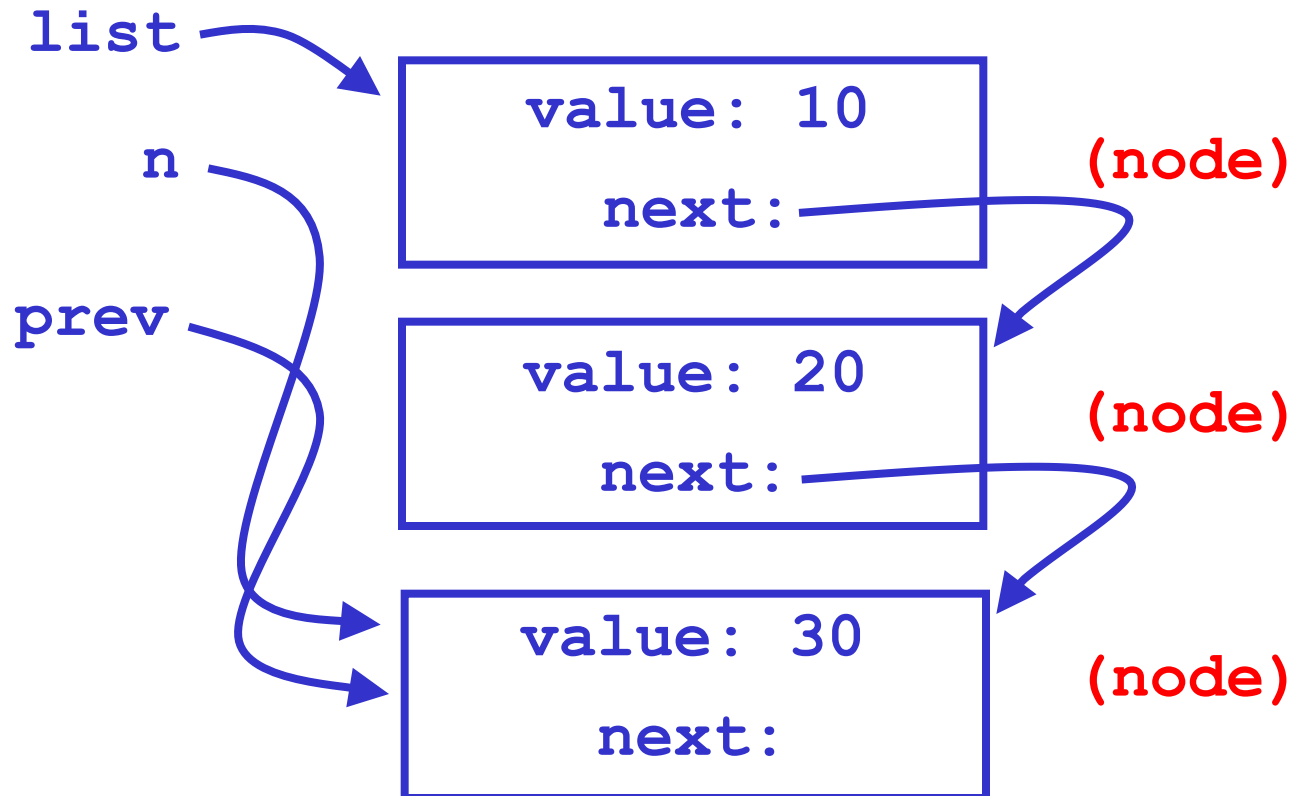
---





# Linked list (diagram)

---

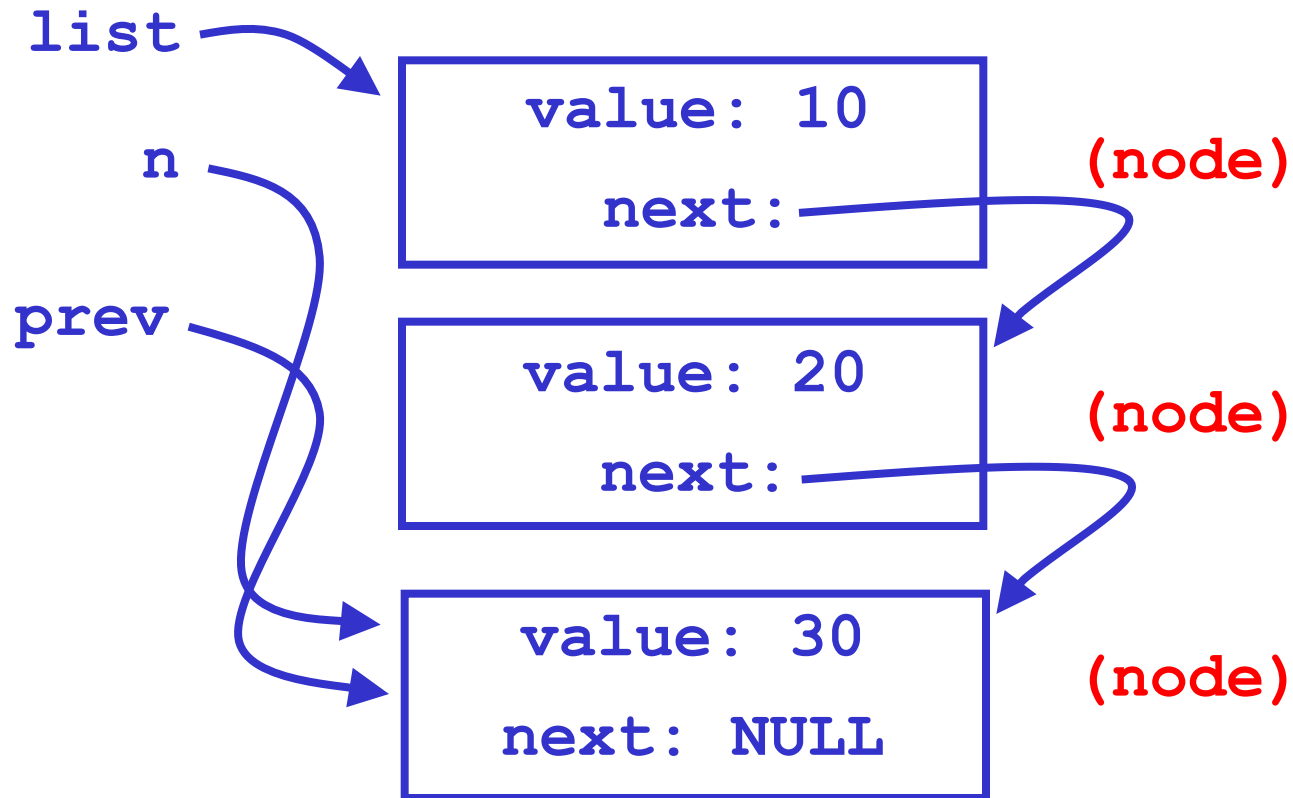






# Linked list (diagram)

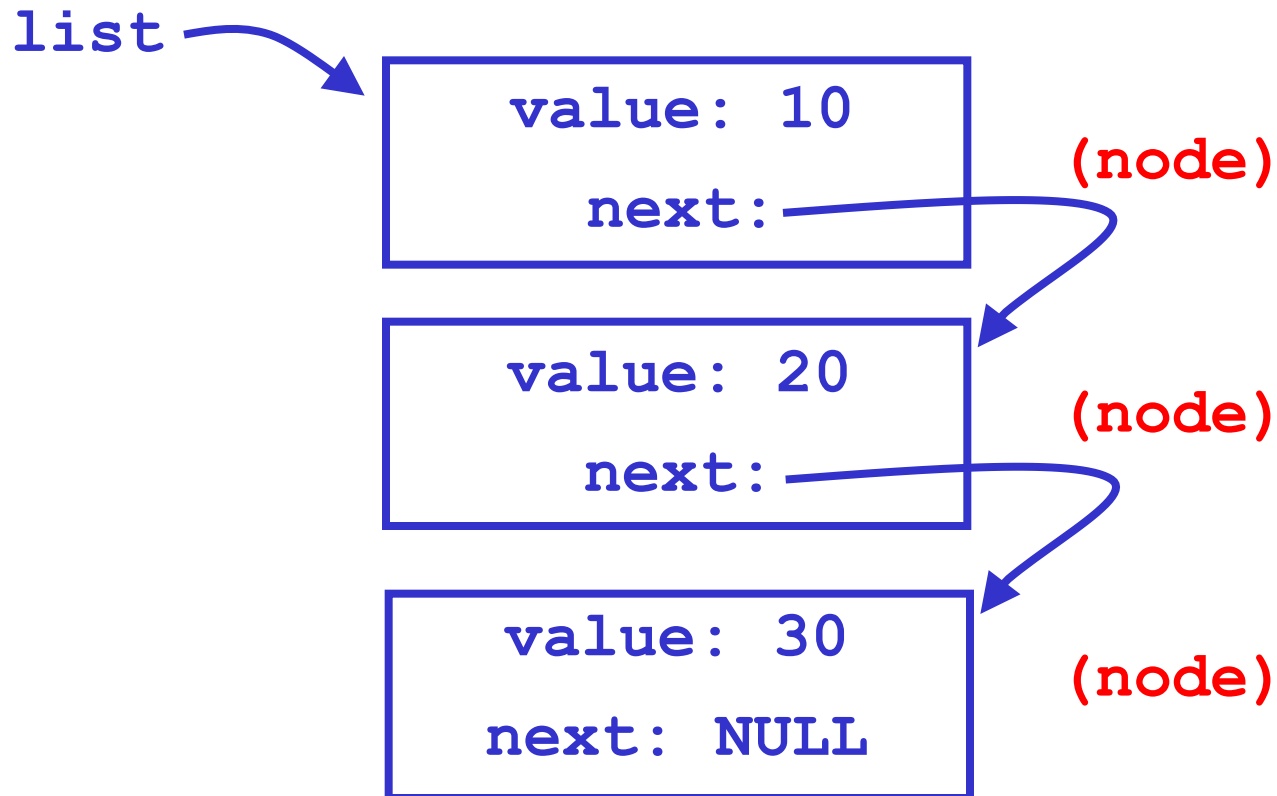
---





# Linked list (final diagram)

---





## Creating a linked list (5)

---

- Can also create linked lists from the end back to the front
- Actually easier to do it that way when possible
  - example: lab 6 command-line arguments
- End-of-list is represented as NULL pointer
- add nodes to previous list (or to NULL)



# Creating a linked list (6)

---

```
list = NULL;      /* Empty list. */  
node *n = (node *) malloc(sizeof(node));  
n->value = 30;  
n->next = list;  
list = n;        /* now 1-node list */
```



# Linked list (diagram)

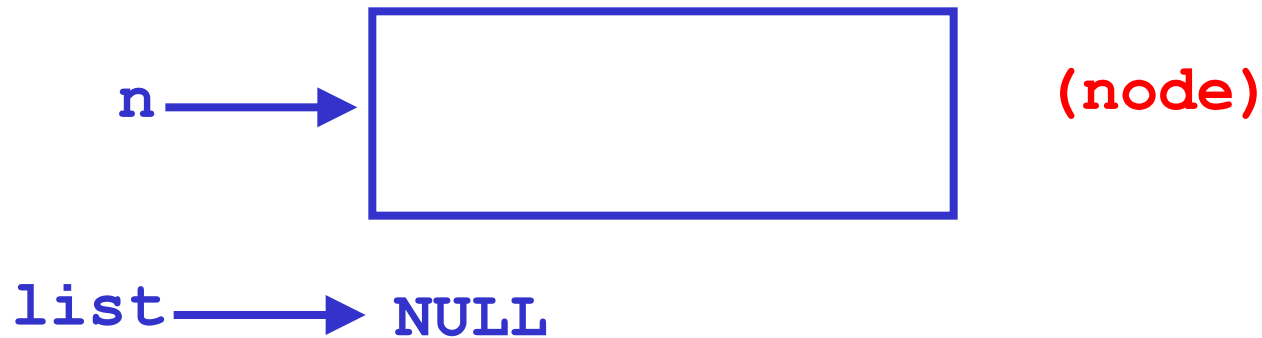
---

`list` → `NULL`



# Linked list (diagram)

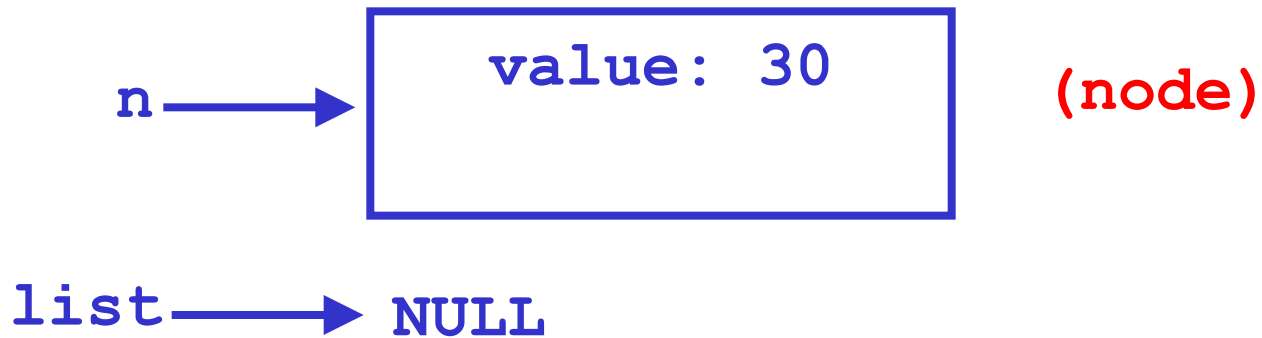
---





# Linked list (diagram)

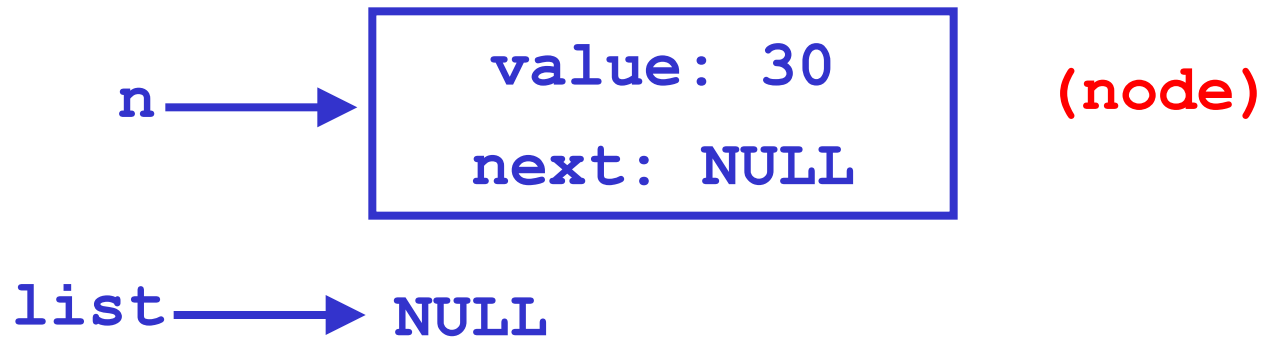
---





# Linked list (diagram)

---

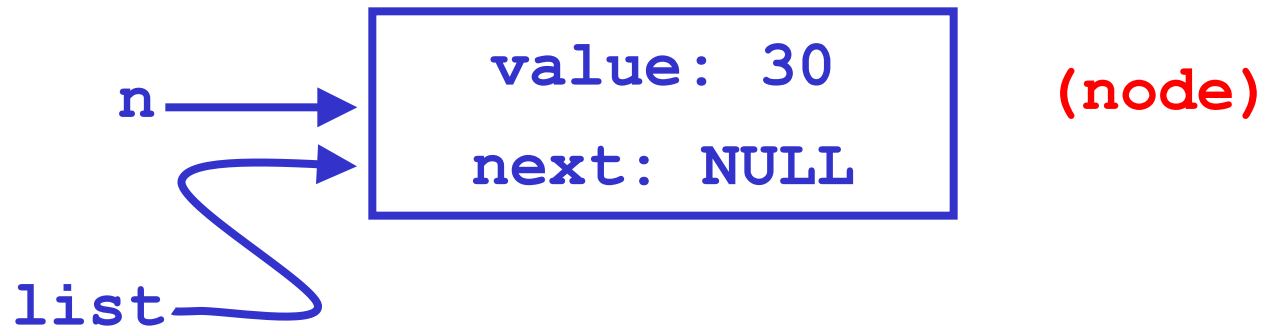






# Linked list (diagram)

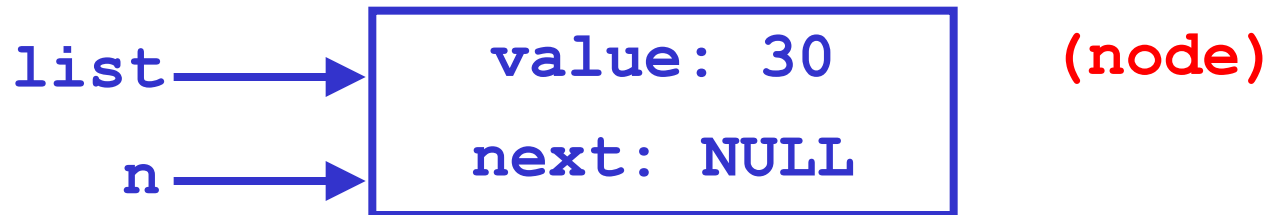
---





# Linked list (diagram)

---





# Creating a linked list (7)

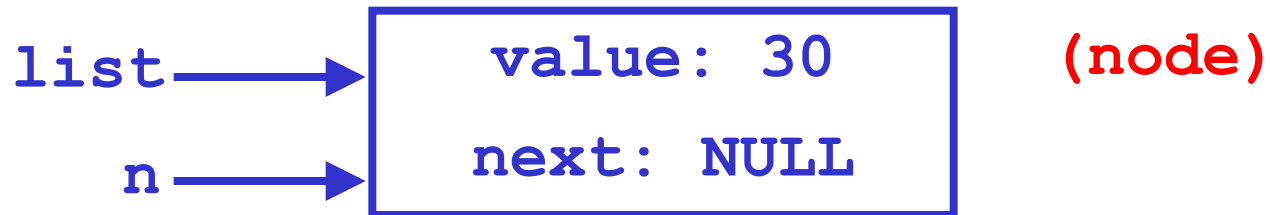
---

```
node *n = (node *) malloc(sizeof(node));  
n->value = 20;  
n->next = list;  
list = n;      /* now 2-node list */
```



# Linked list (diagram)

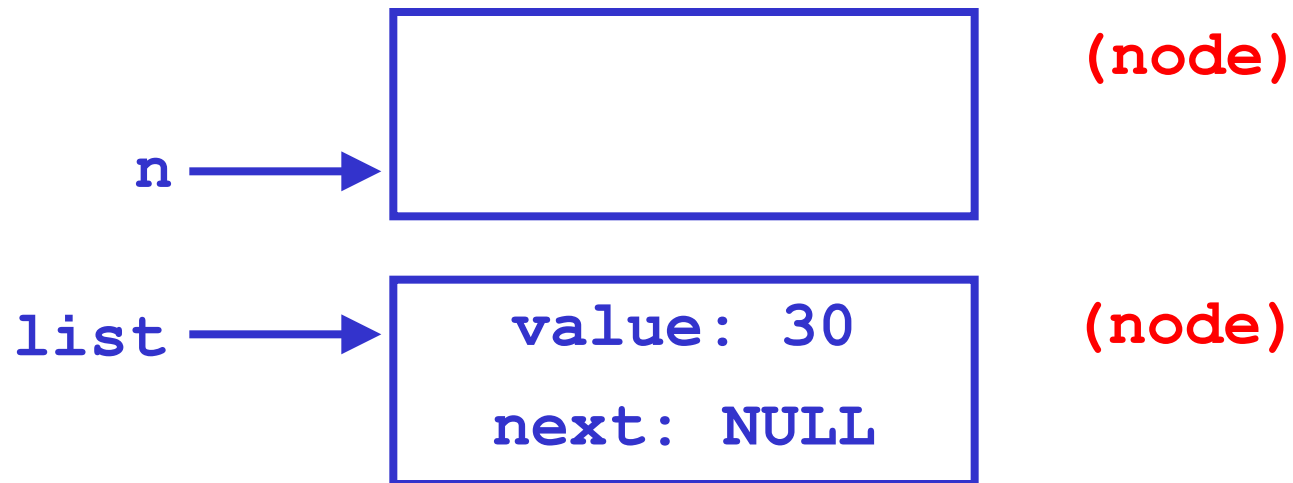
---





# Linked list (diagram)

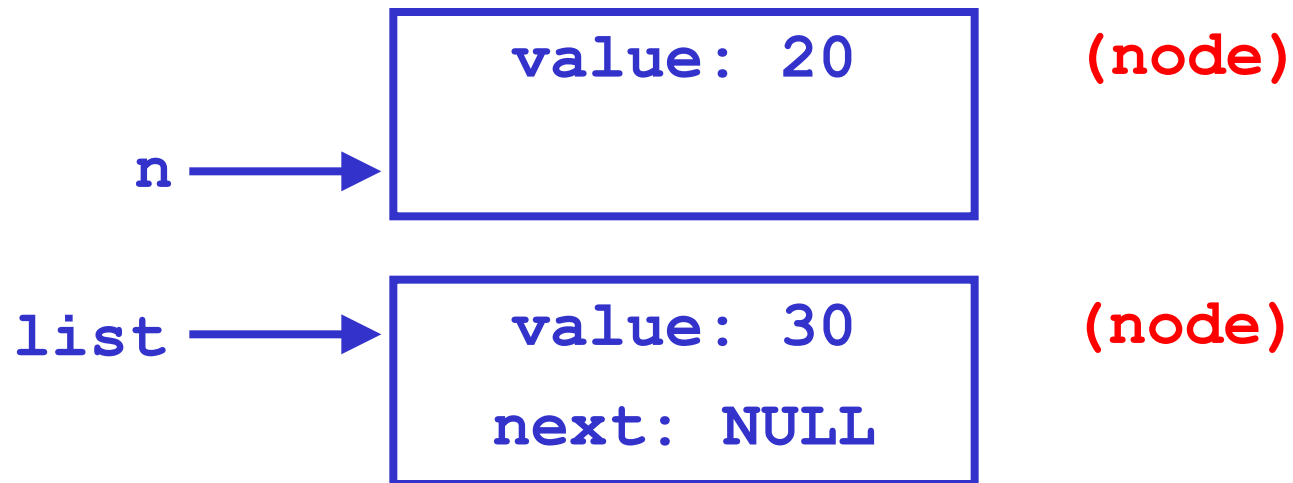
---





# Linked list (diagram)

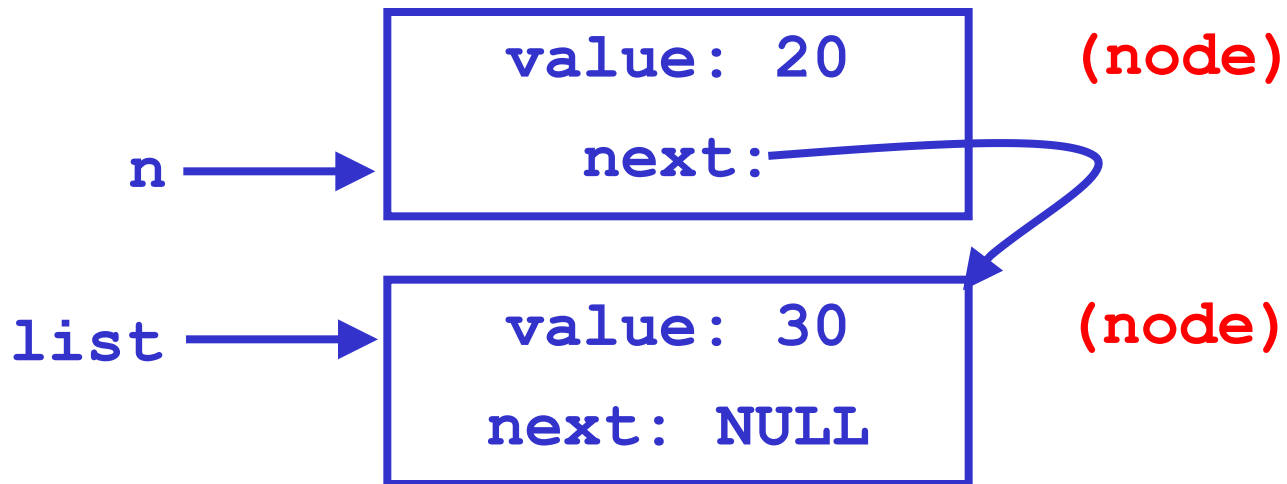
---





# Linked list (diagram)

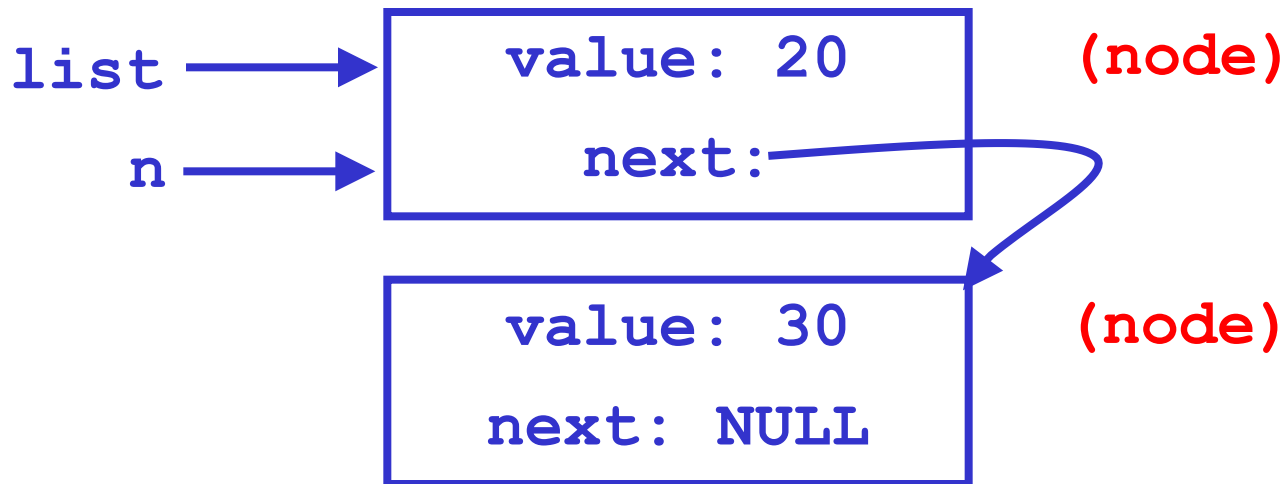
---





# Linked list (diagram)

---







# Creating a linked list (8)

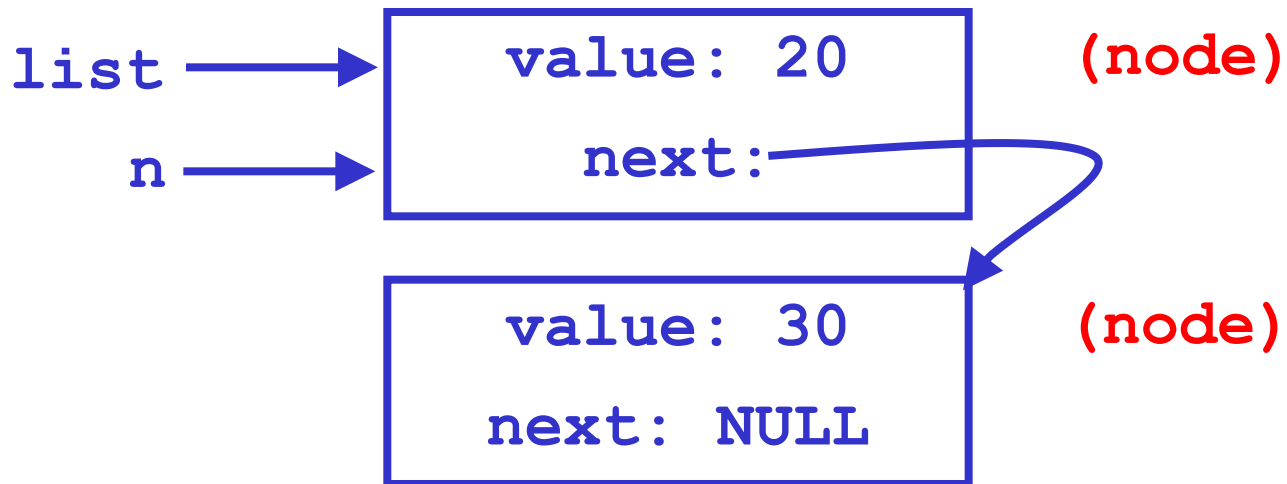
---

```
node *n = (node *) malloc(sizeof(node));  
n->value = 10;  
n->next = list;  
list = n;      /* now 3-node list */
```



# Linked list (diagram)

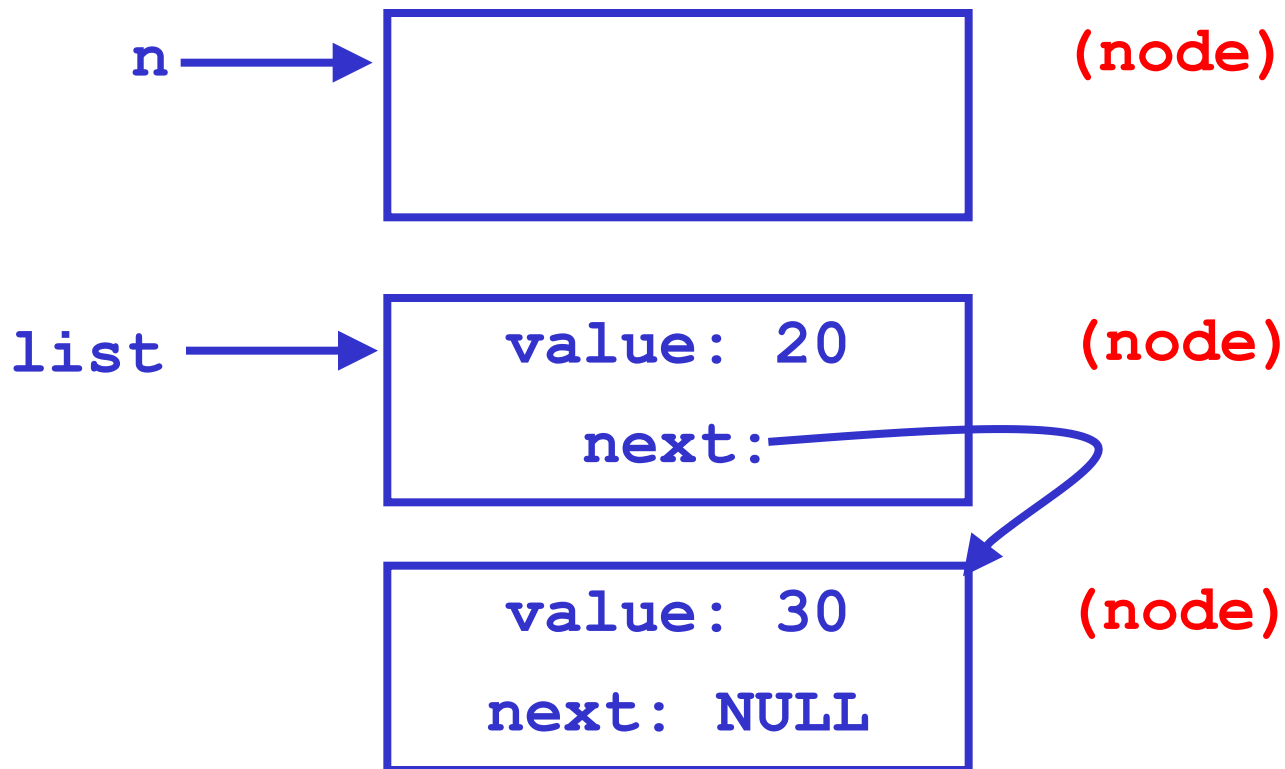
---





# Linked list (diagram)

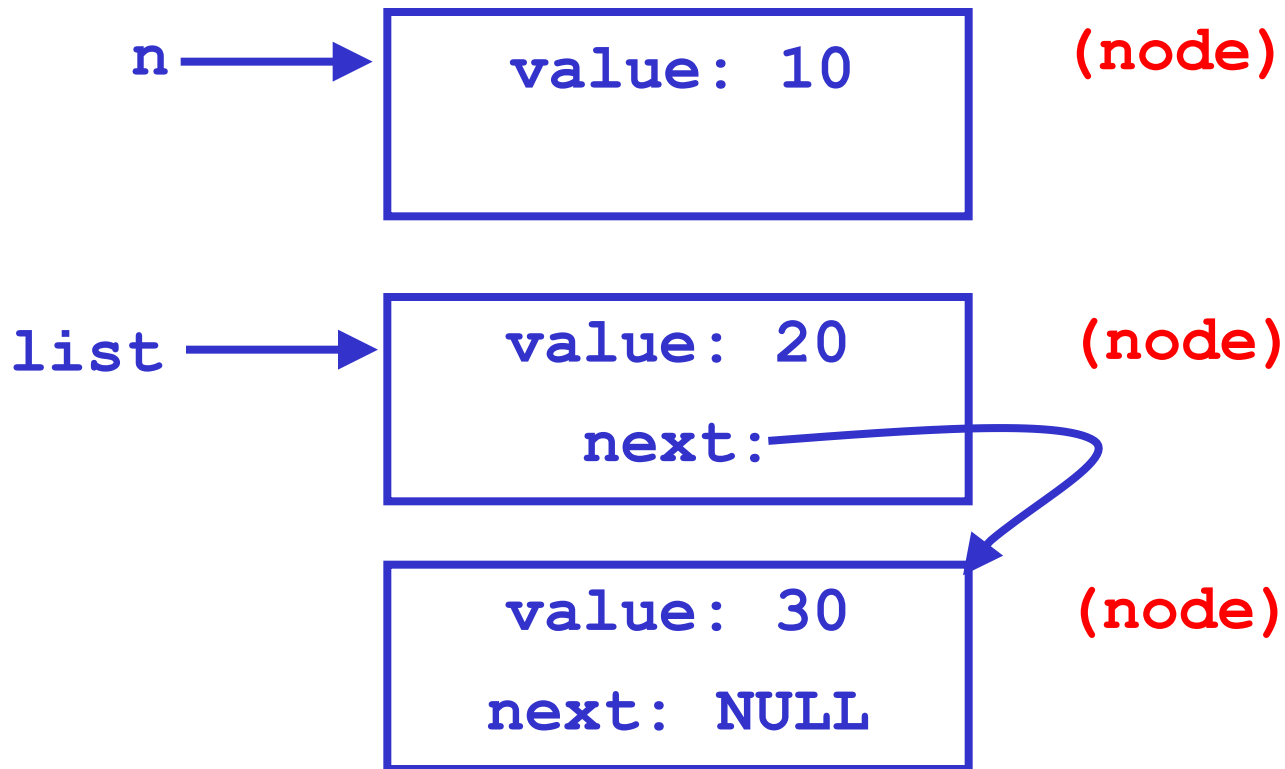
---





# Linked list (diagram)

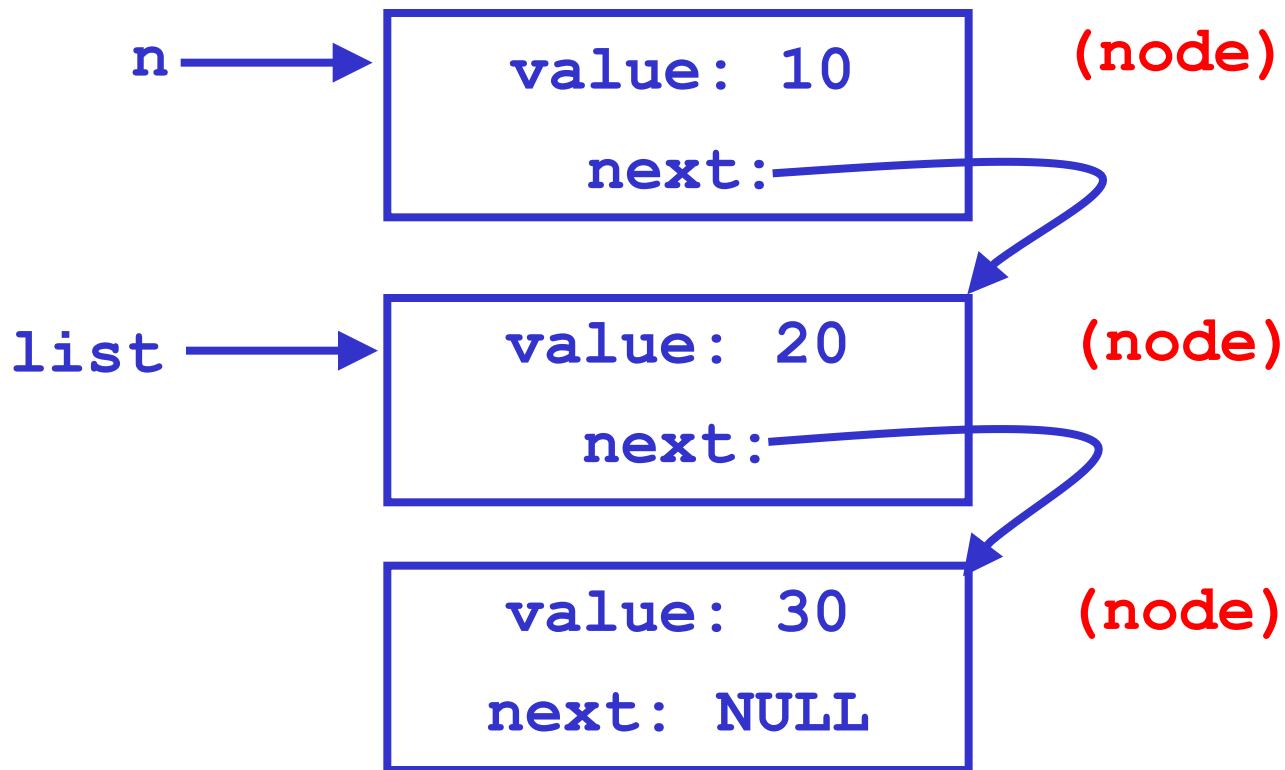
---





# Linked list (diagram)

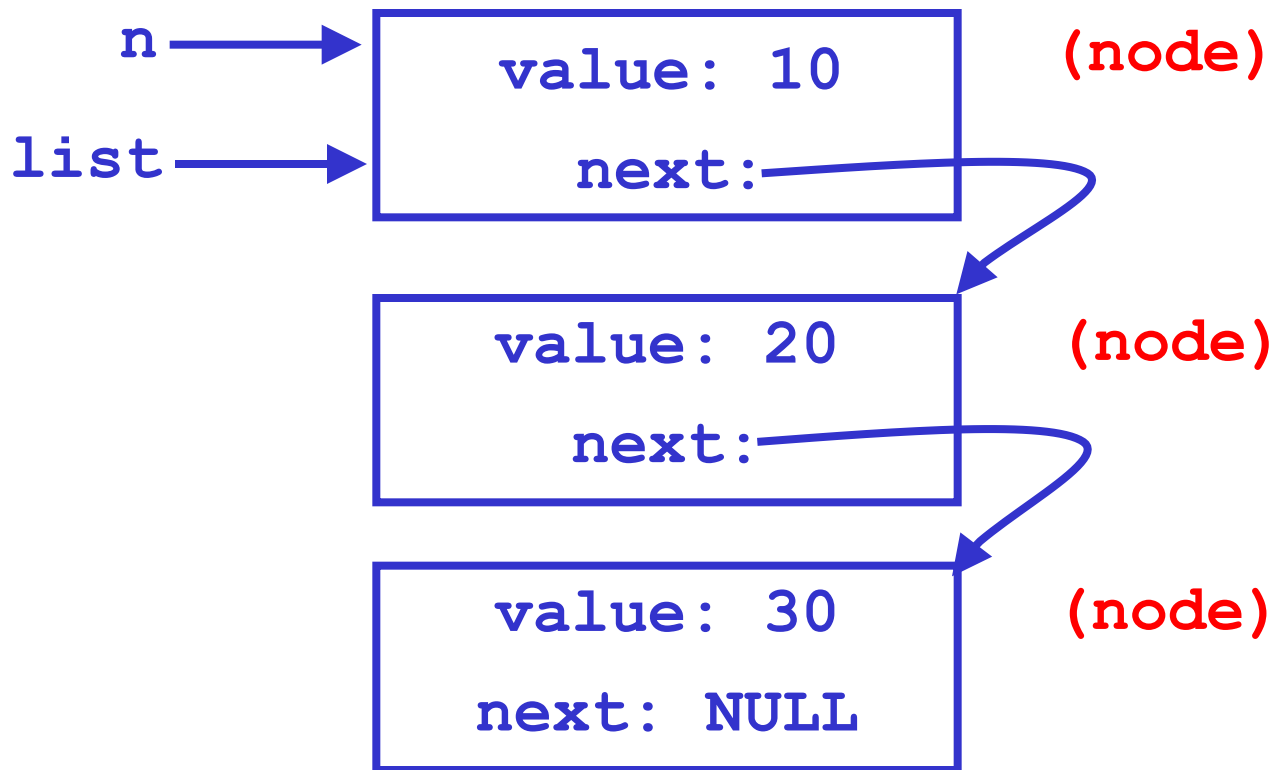
---





# Linked list (diagram)

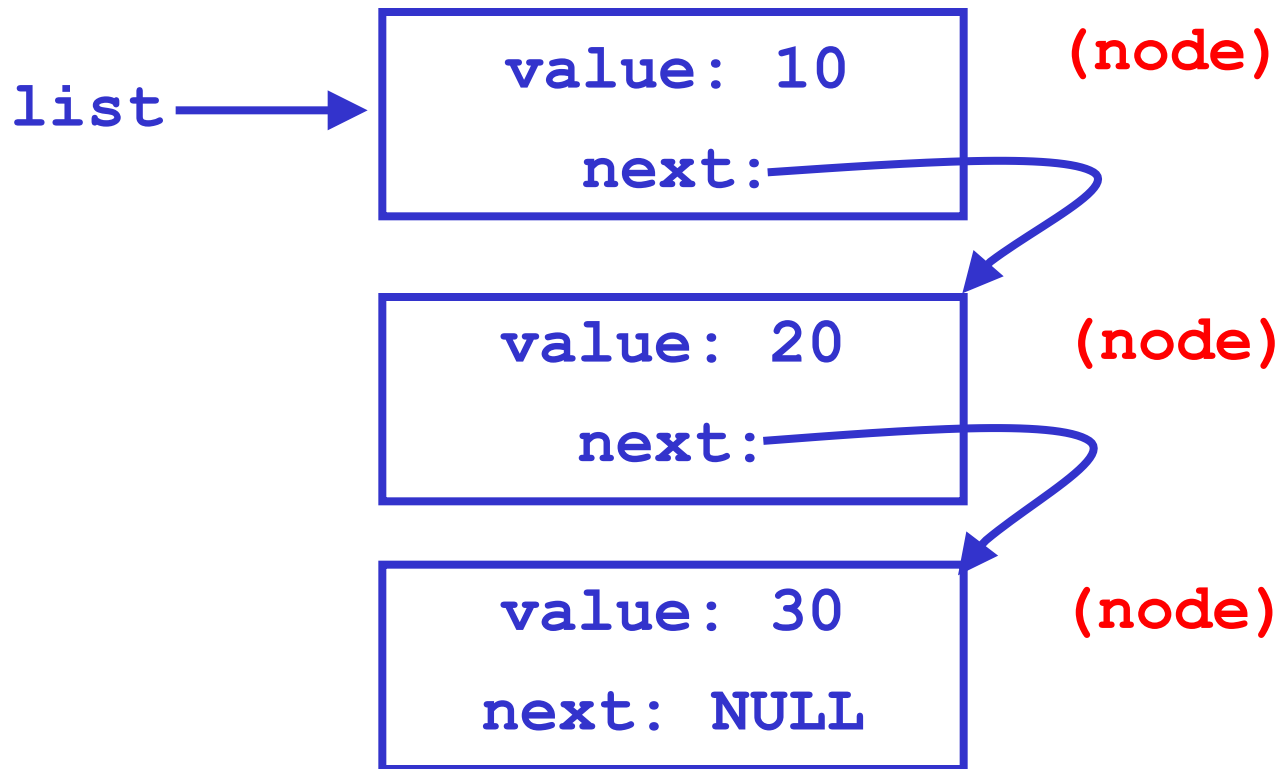
---





# Linked list (final diagram)

---





# Checking malloc()

---

- Previous code simplified to fit on slide
- Actually should check every `malloc` call for failure

```
n = (node *)malloc(sizeof(node));  
if (n == NULL)  
{  
    fprintf(stderr,  
            "Error: out of memory.\n");  
    exit(1);  
}
```





# Iterating through a linked list

---

- Standard idiom for going through linked lists:

```
node *n;
```

```
/* Set all node values to zero. */
```

```
for (n = list; n != NULL; n = n->next) {
```

```
    n->value = 0;
```

```
}
```

- You should be able to figure out how this works



# Lab 6

---

- This week's lab:
  - New sorting algorithm: "quicksort"
  - More efficient than ME sort, bubblesort
  - Use on linked lists, not arrays
  - Memory management will be a challenge!



# Next time

---

- Hash tables
- More "fun" with pointers ;-)