Dynamic memory in C

- Linked lists in C

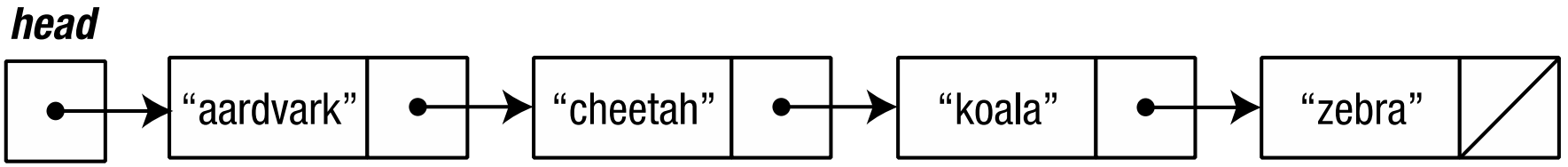
Lists

- While arrays are a convenient structure, they are not always the most suitable choice.
  - Arrays have a fixed size (albeit it may be resized with effort), yet a linked-list is exactly the size it needs to be to hold its contents.
  - Lists can be rearranged by changing a few pointers (which is cheaper than a block move like that performed by **memmove** in our dynamic-array implementation of **delname**)
  - When items are inserted or deleted from a list, the other items are not moved in memory.
  - If we store addresses to the **list elements** in some other data structure, the **list elements themselves** won't be necessarily be invalidated by changes to the **list**.

Lists

- So:
  - If the set of items we want to maintain changes frequently…
  - … especially if the number of items is unpredictable…
  - then **a list is one good way to store them.**
- Typical usage of list for problem will dictate the kind of linked-list:
  - singly
  - singly, with head & tail pointers
  - doubly
  - circular
  - skip-list

Singly-linked list (no tail pointer)



- Set of four items
  - Each item has data (in this case a string) along with a pointer to the next item.
  - Head of the list is a pointer to the first item
  - End of the list is denoted by a NULL pointer.
  - Handful of operations (add new item to front; find a specific item; add new item before or after a specific item; perhaps delete item)

List node

- We'll revisit the same problem as described earlier (that of storing <name, value> pairs)
- The one addition to the Nameval struct is a "next" field
  - This field's type is a pointer to the node type Nameval
  - This is the usual style in C of declaring types for self-referencing structures.
  - We'll see more recursive structures later…

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int     value;
    Nameval *next; /* in list */
};
```

Slight detour

- One of the tedious aspects of working with malloc is checking for success or failure
- We can accomplish this while still keeping our code clean by writing a small support function.
  - emalloc: a **wrapper function** that calls malloc; if allocation fails, it reports an error and exits the program.
  - Therefore we can use it as a memory allocator that never returns failure (but does terminate the program if there was an error).

```
void *emalloc(size_t n)
{
    void *p;

    p = malloc(n);
    if (p == NULL) {
        fprintf(stderr, "malloc of %u bytes failed", n);
        exit(1);
    }
    return p;
}
```

Constructing an item

- Before "creating a list", let us write a function that constructs an item.
  - It will allocate memory from the heap…
  - … and then assign appropriate values to fields.
  - We will make heavy us of "->" syntax
  - We assume here that some other function has allocated memory for the name

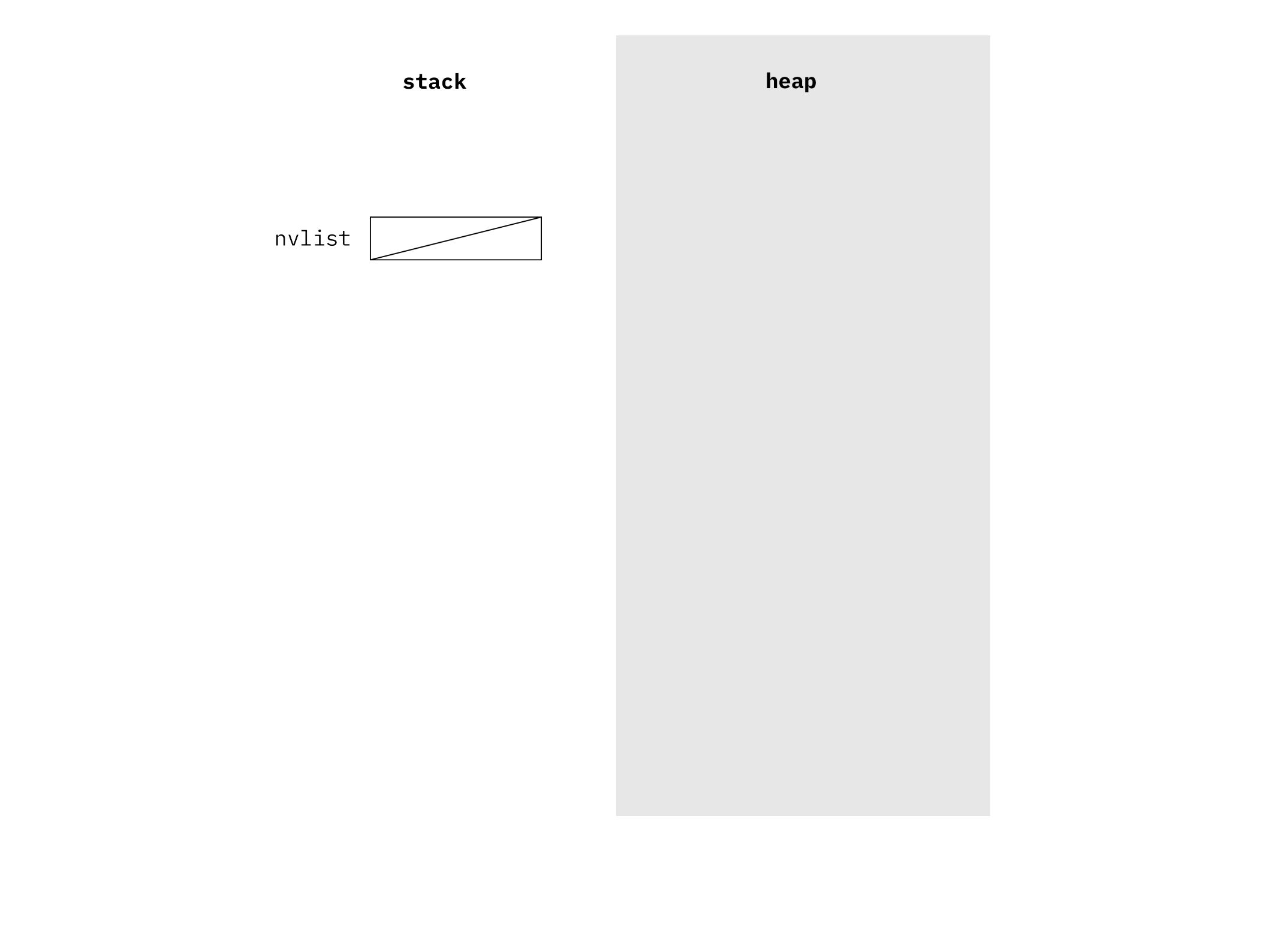
```
Nameval *newitem (char *name, int value)
{
   Nameval *newp;

   newp = (Nameval *) emalloc(sizeof(Nameval));
   newp->name  = name;  /* Is this exactly what we want??? */
   newp->value = value;
   newp->next  = NULL;
   return newp;
}
```

Adding an item to the front

- This is the simplest way to assemble a list
  - Also the fastest.
- This function (and others we'll write) all return a pointer to the first element as their function value
  - Note that this even works if the list is empty (e.g., pointing to NULL)

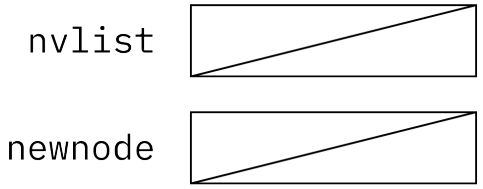
```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```

```
/* typical usage */
Nameval *nvlist = NULL;
...
Nameval *newnode = newitem(strdup("Honest Abe"), 1809);
nvlist = addfront(nvlist, newnode);
```

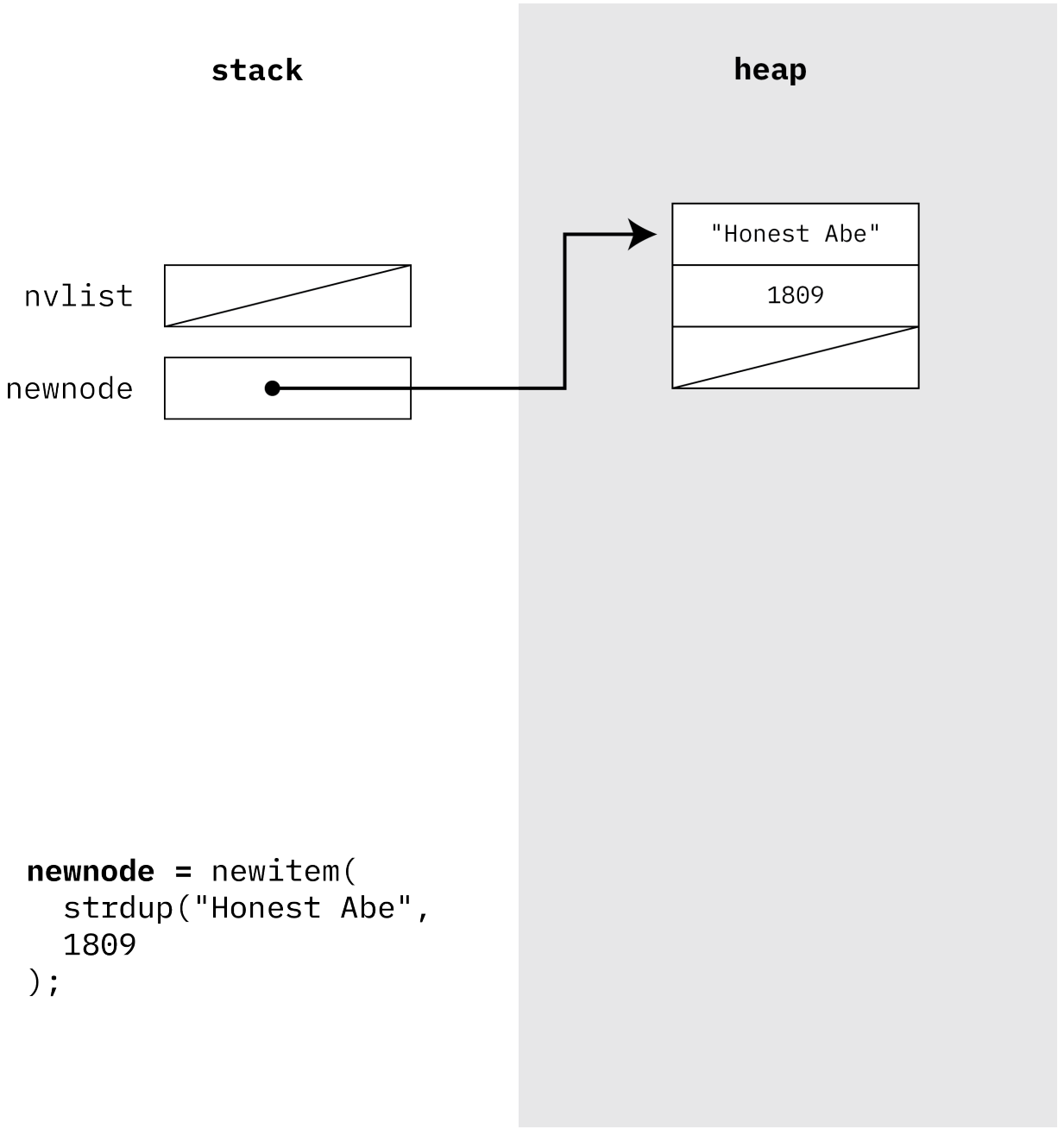

stack
heap
nvlist

stack

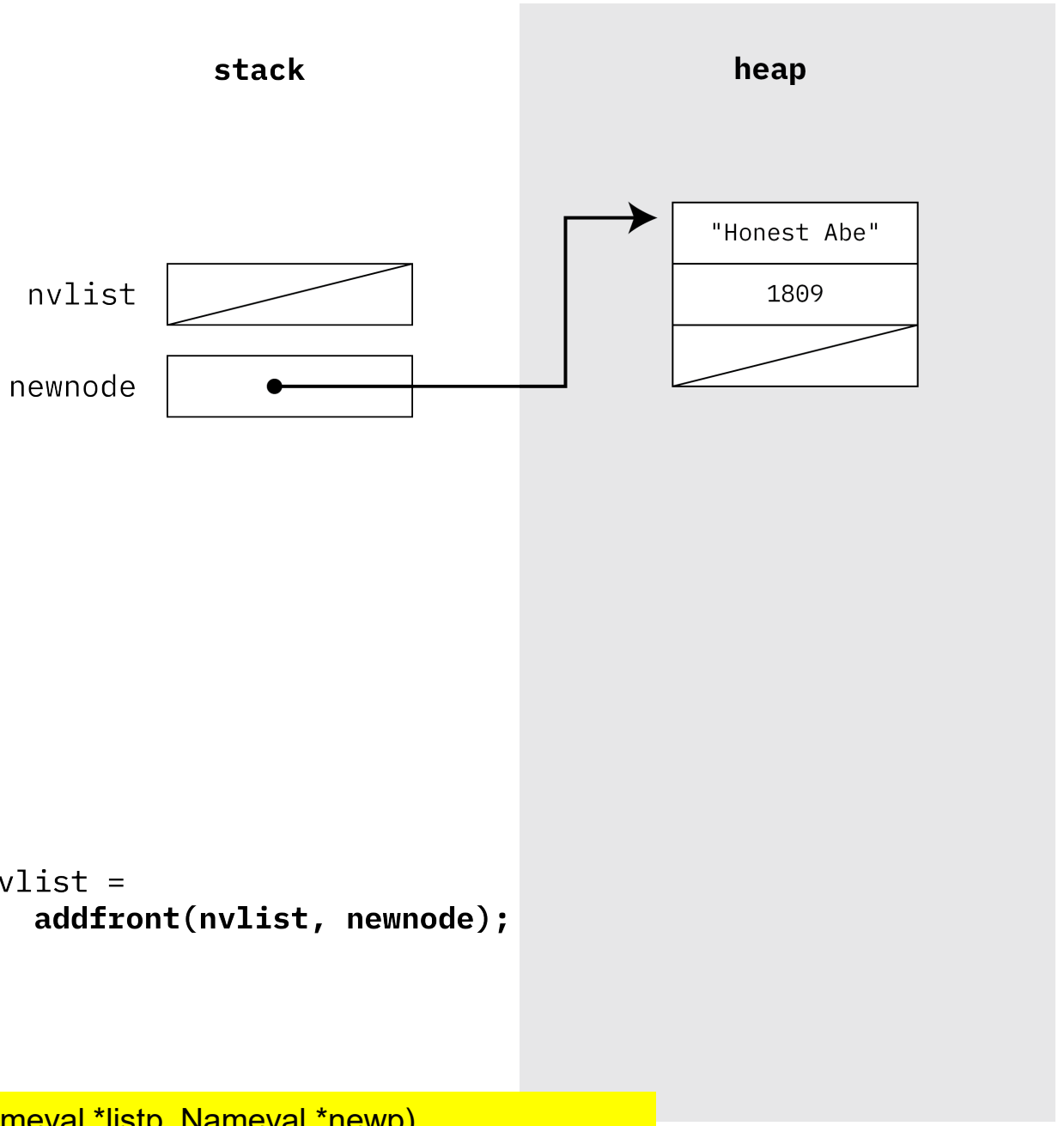
heap




```
newnode = newitem(
  strdup("Honest Abe",
  1809
);
```

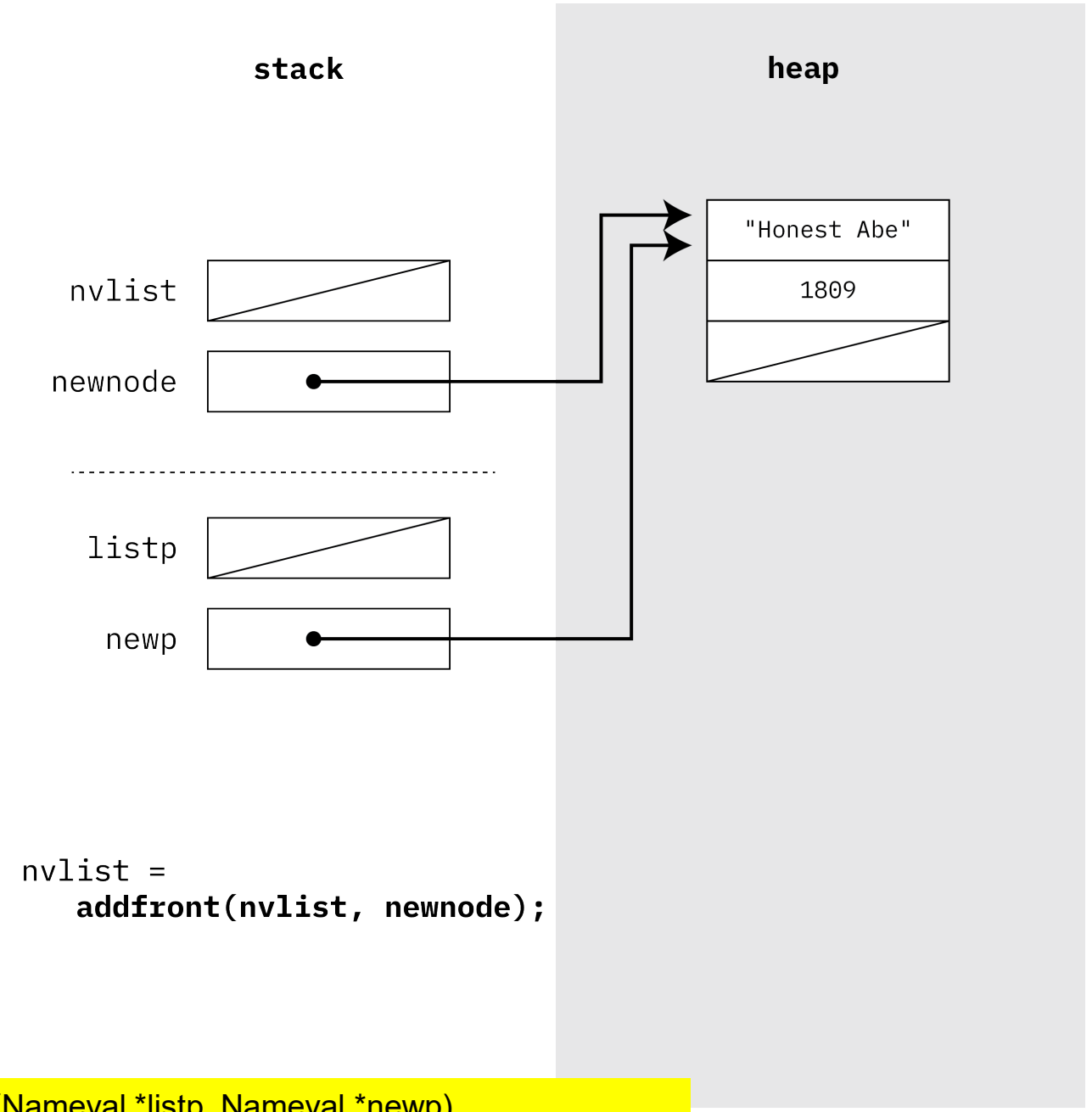



```
newnode = newitem(
  strdup("Honest Abe",
  1809
);
```



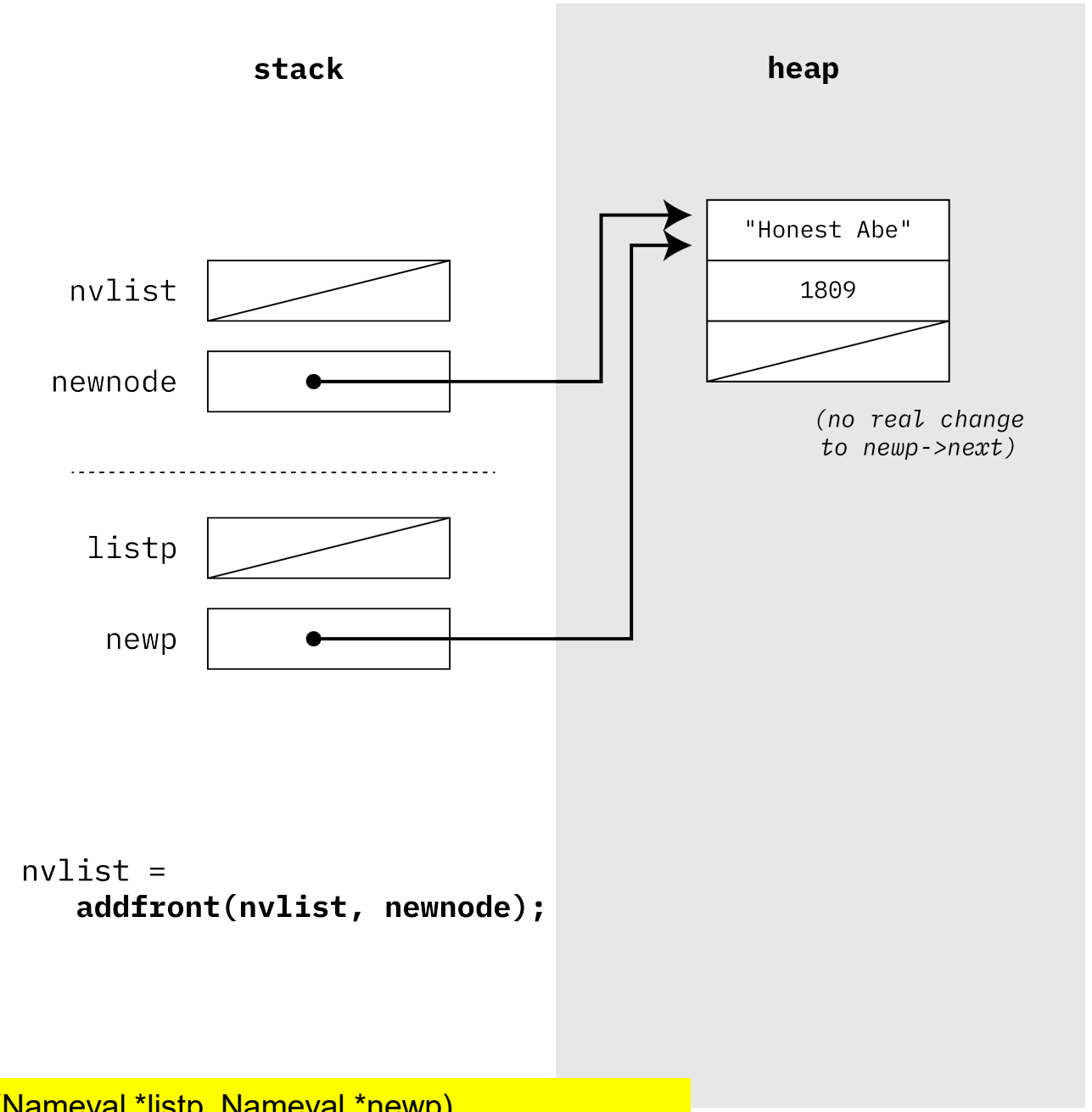

```
nvlist =
   addfront(nvlist, newnode);
```

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
   newp->next = listp;
   return newp;
}
```



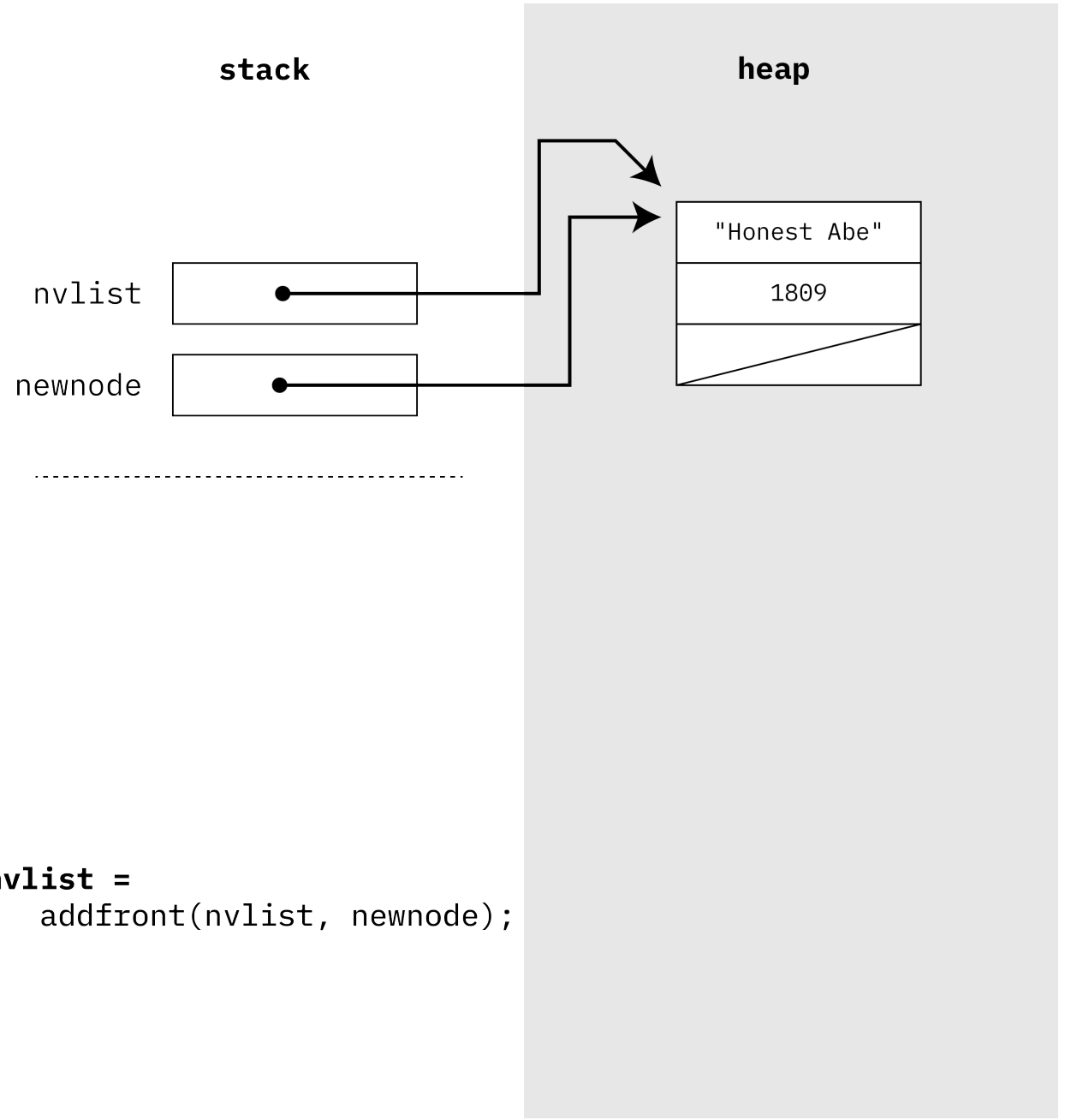

```
nvlist =
   addfront(nvlist, newnode);
```

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
   newp->next = listp;
   return newp;
}
```

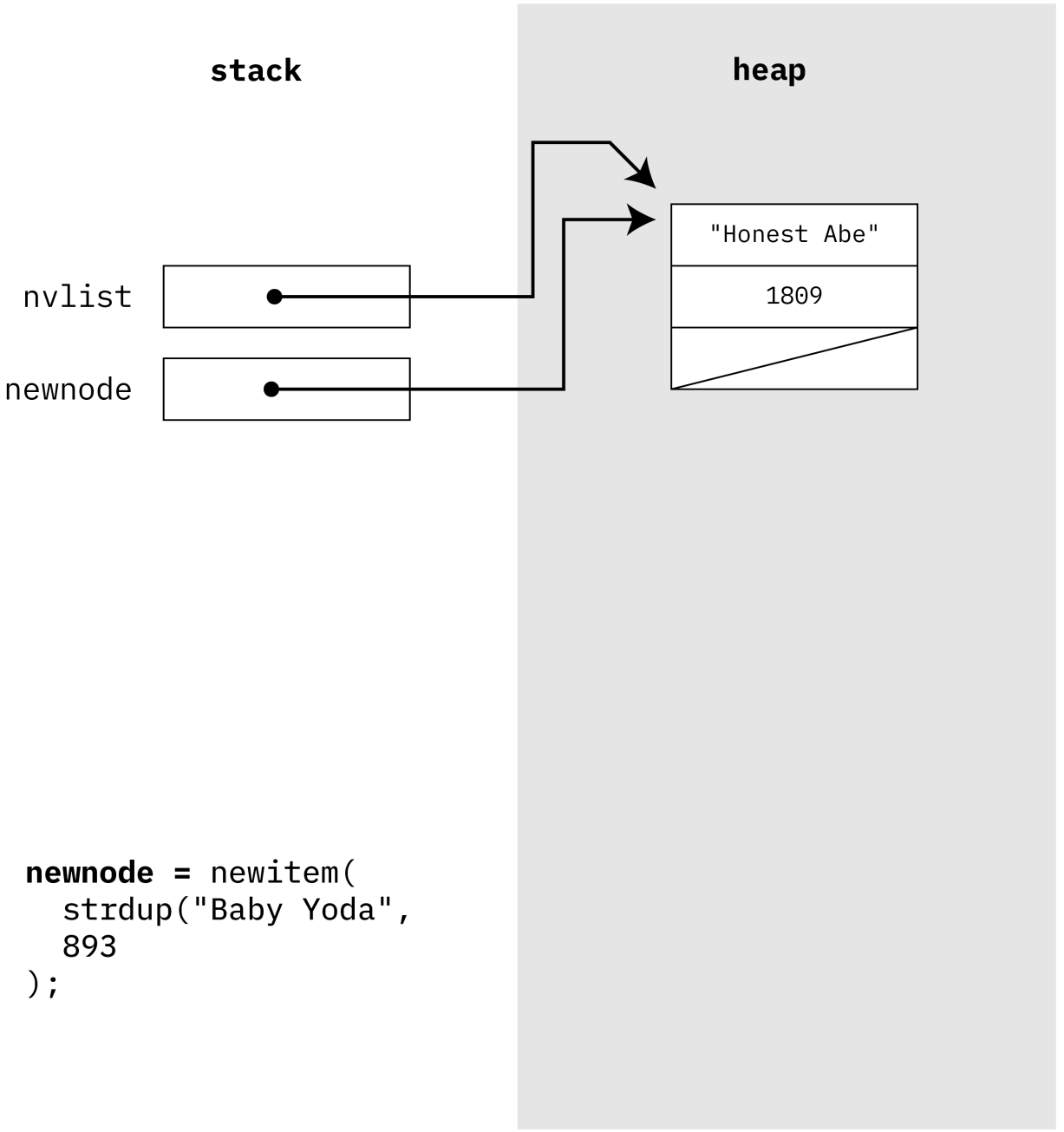



```
nvlist =
   addfront(nvlist, newnode);
```

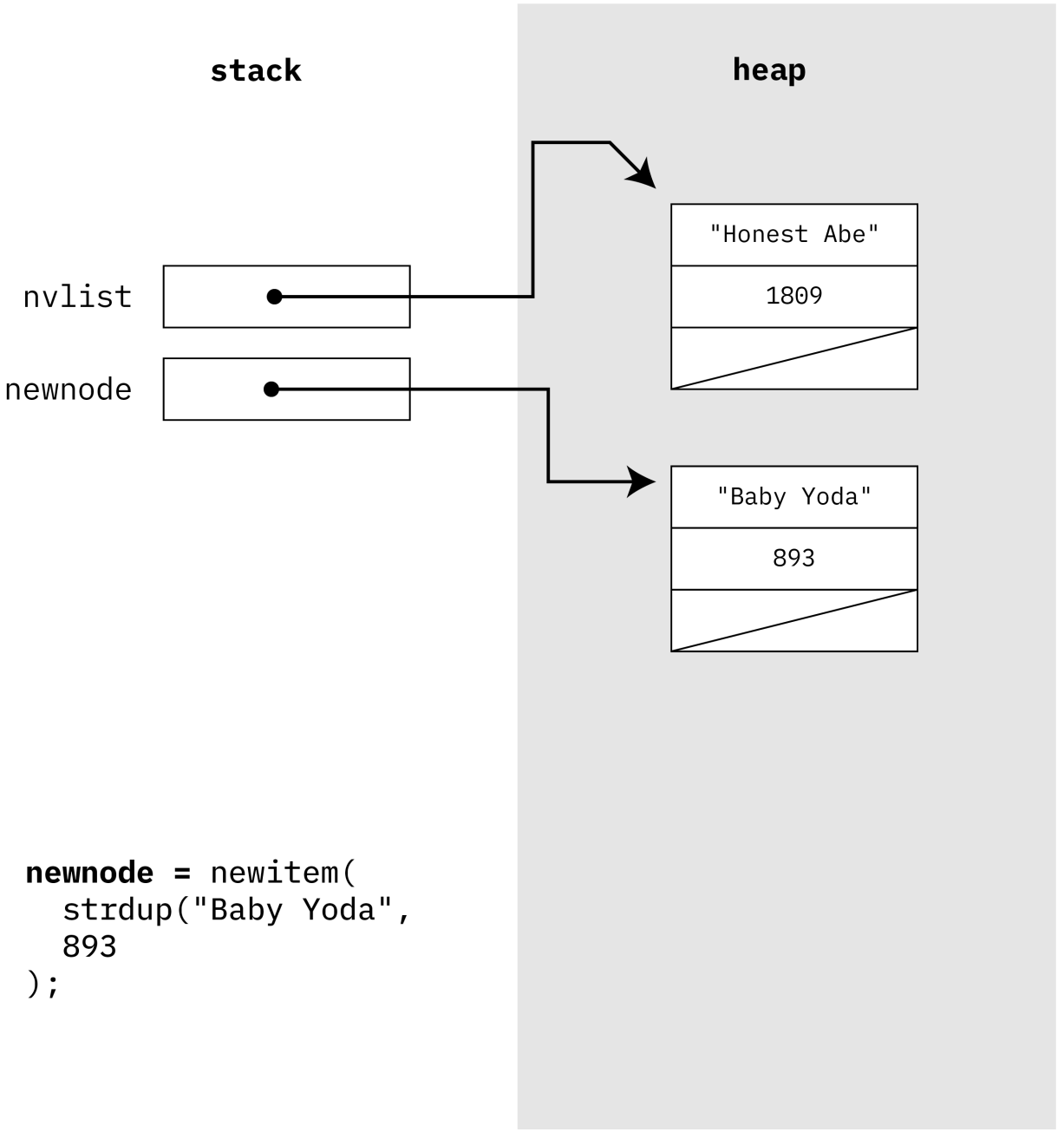
```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
   newp->next = listp;
   return newp;
}
```



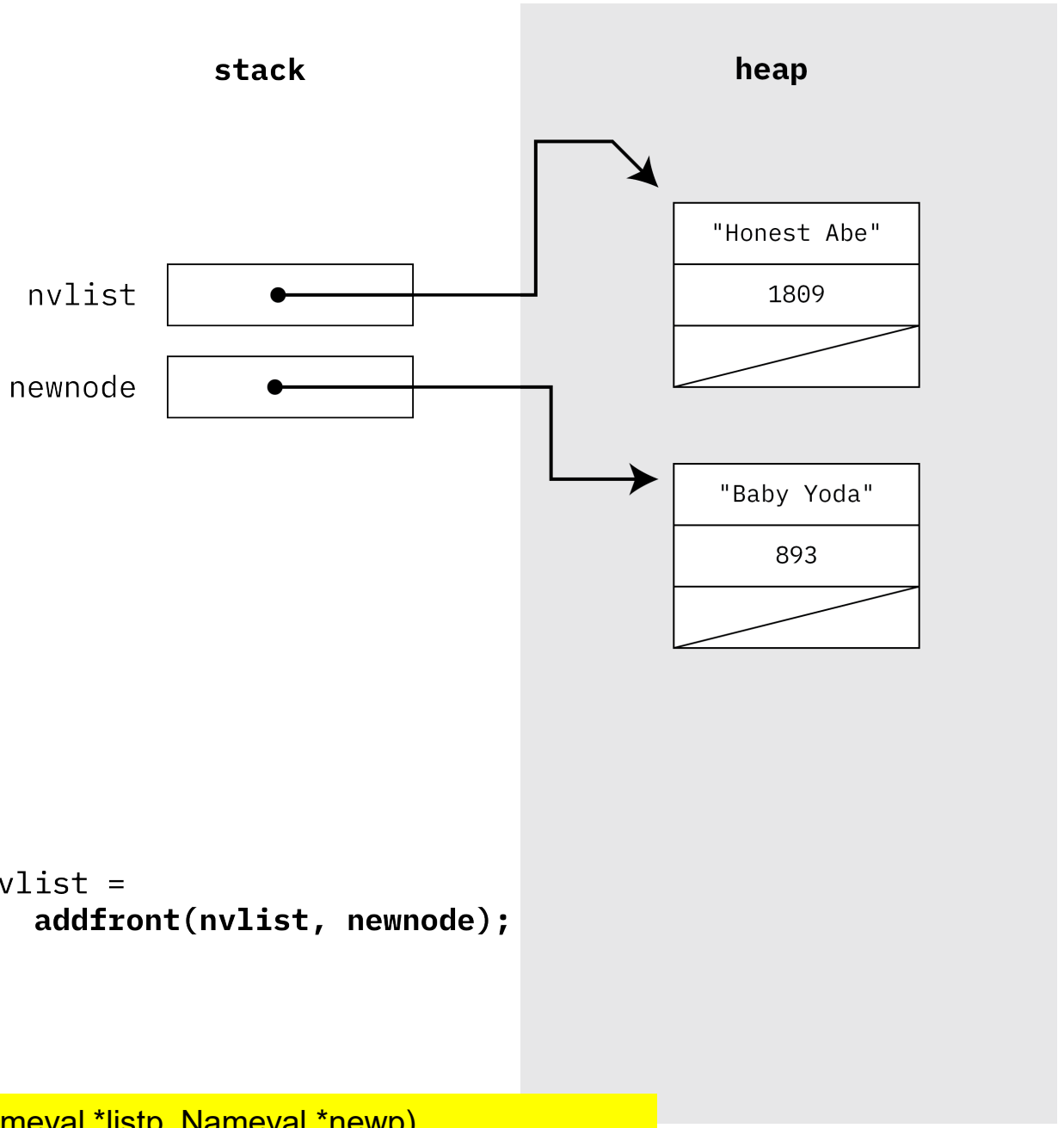
nvlist =
   addfront(nvlist, newnode);




```
newnode = newitem(
  strdup("Baby Yoda",
  893
);
```

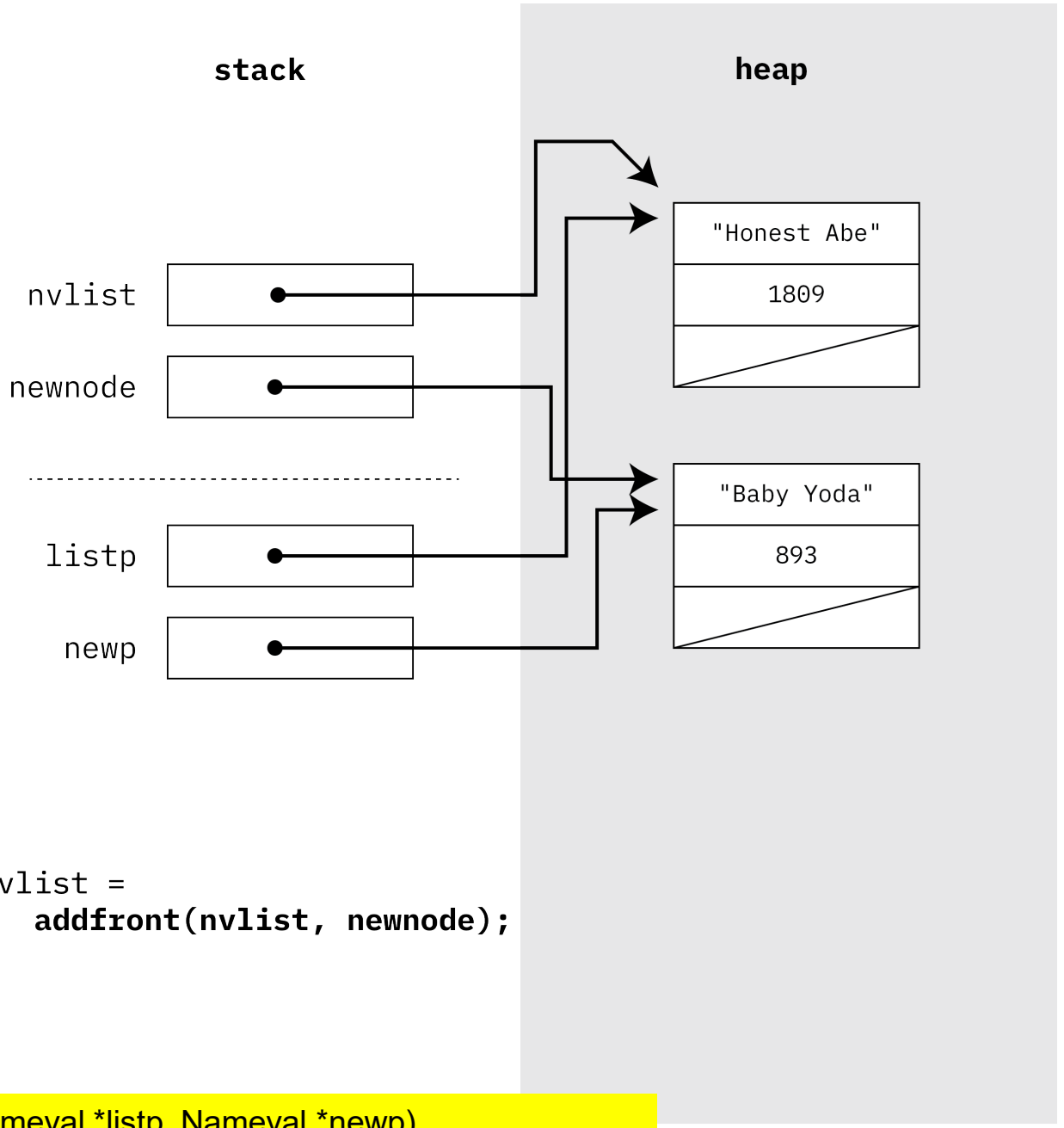



```
newnode = newitem(
  strdup("Baby Yoda",
  893
);
```



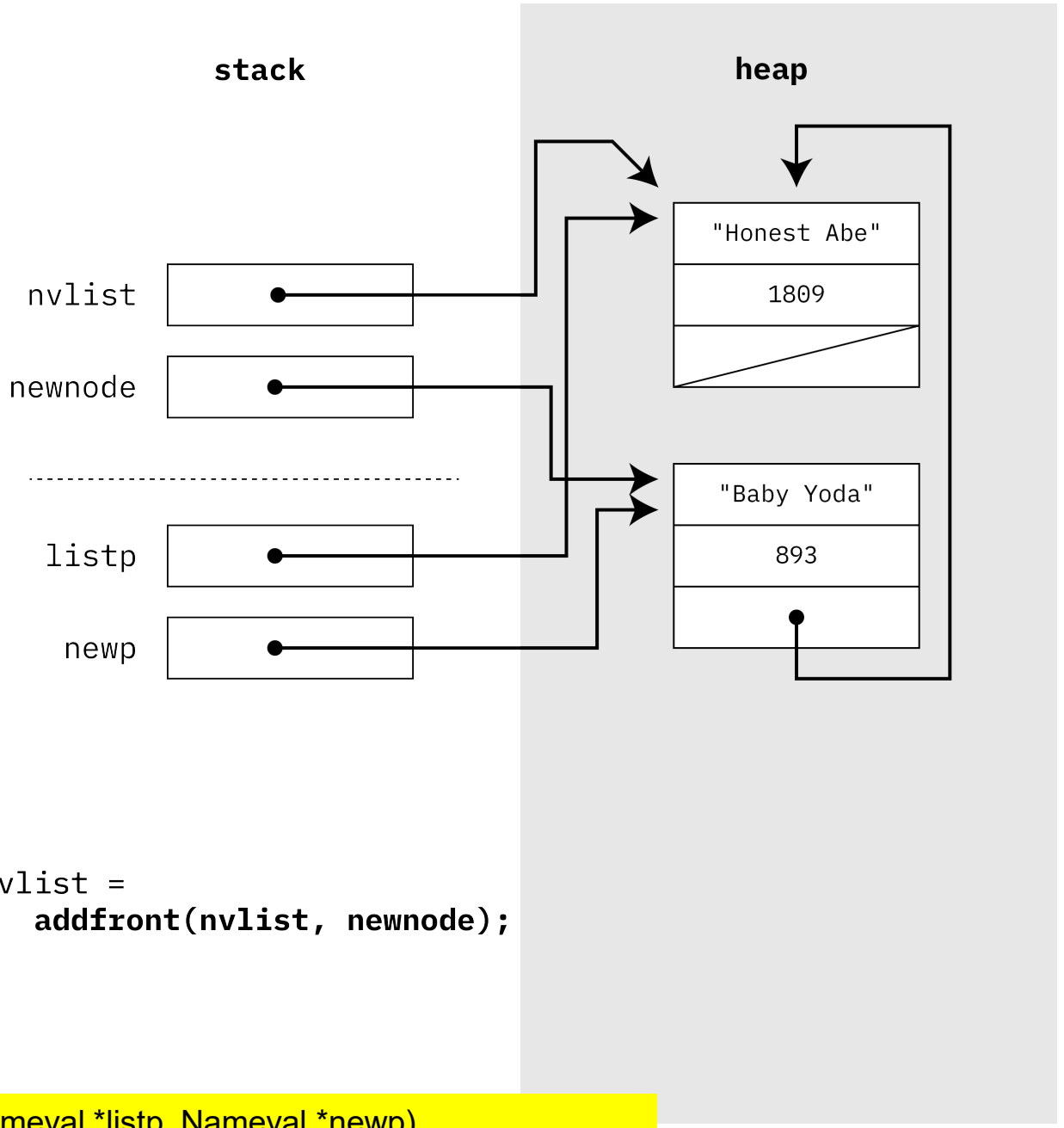

```
nvlist =
   addfront(nvlist, newnode);
```

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
   newp->next = listp;
   return newp;
}
```



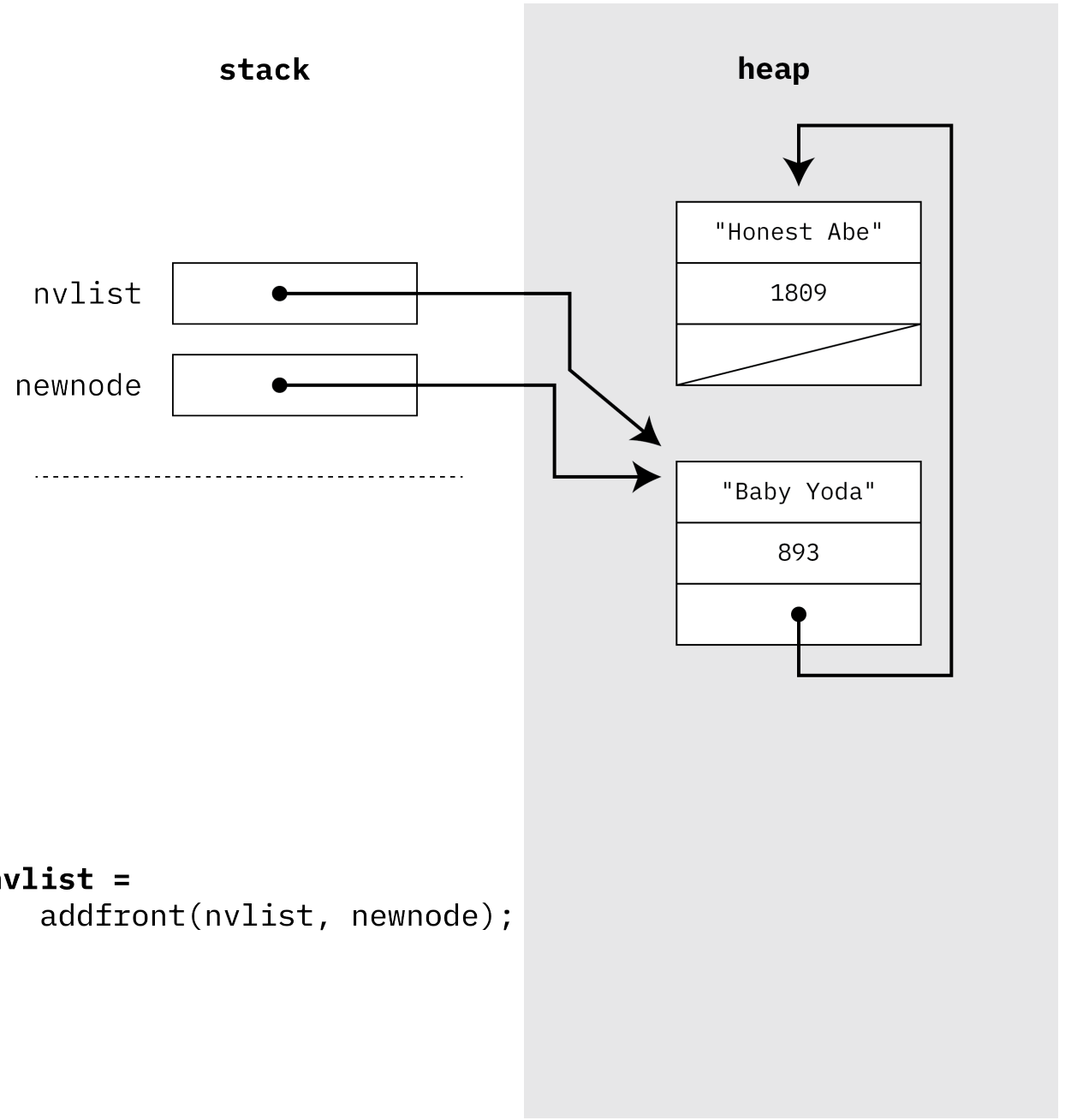

```
nvlist =
   addfront(nvlist, newnode);
```

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
   newp->next = listp;
   return newp;
}
```

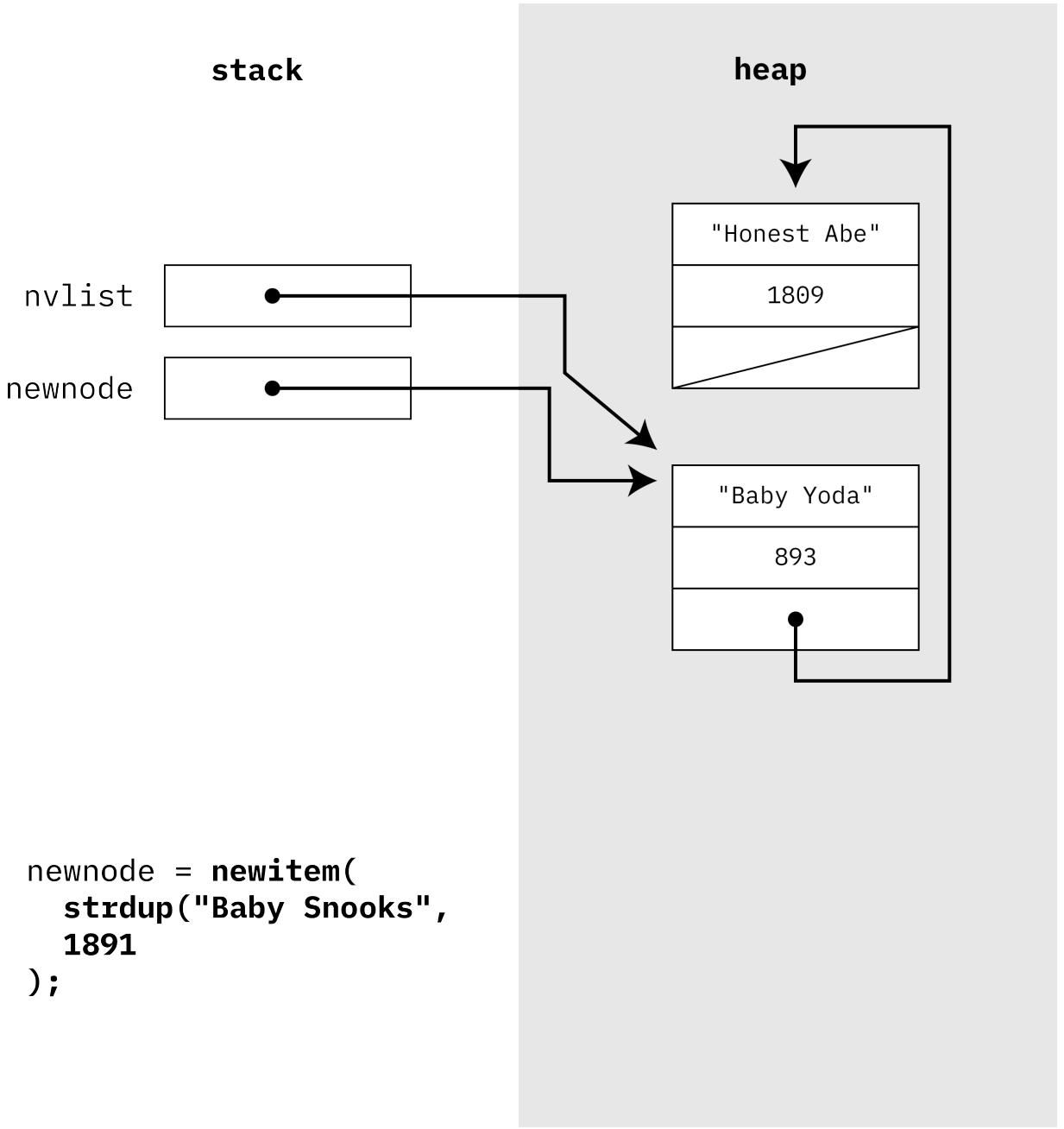



```
nvlist =
   addfront(nvlist, newnode);
```

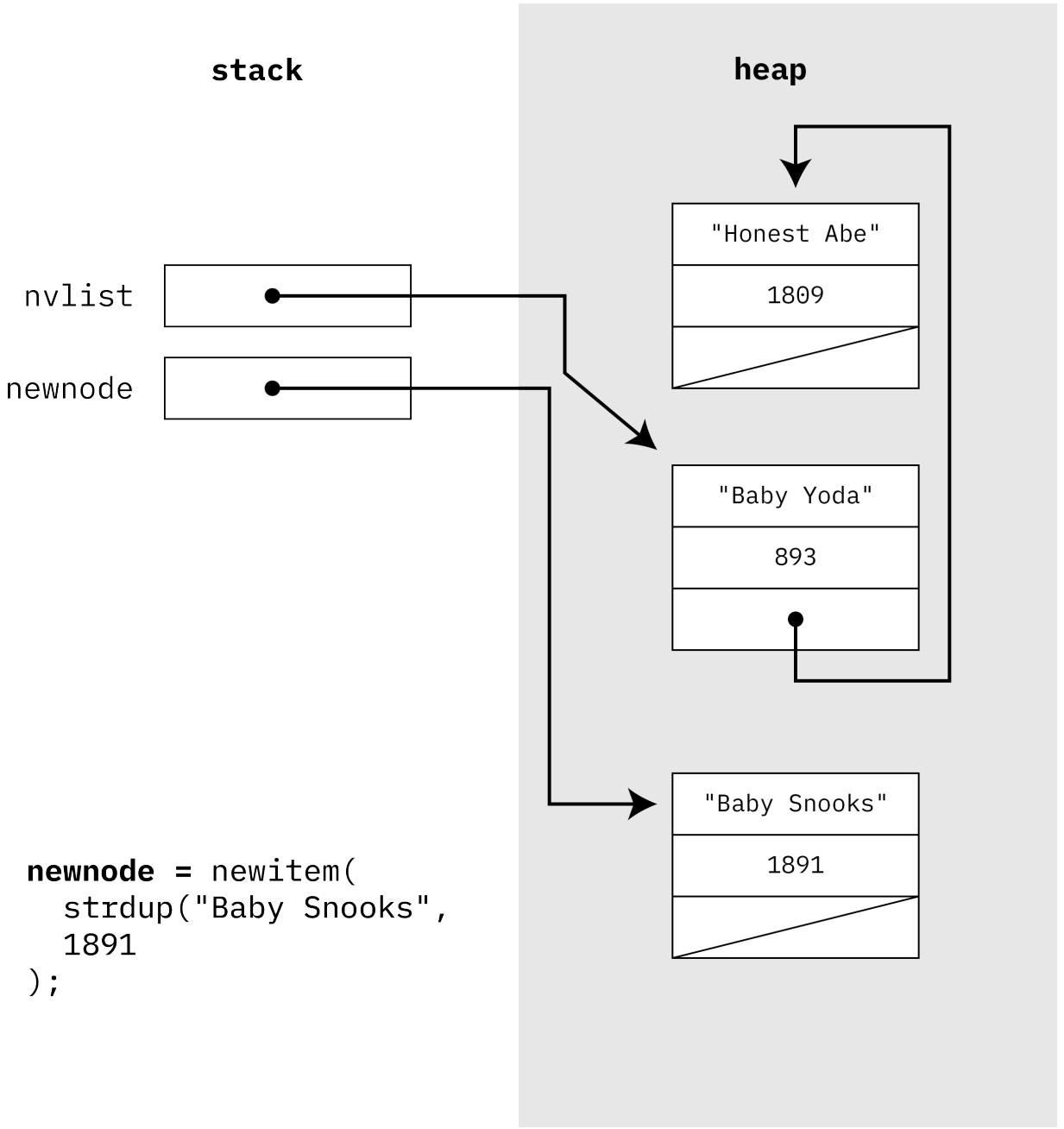
```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
   newp->next = listp;
   return newp;
}
```



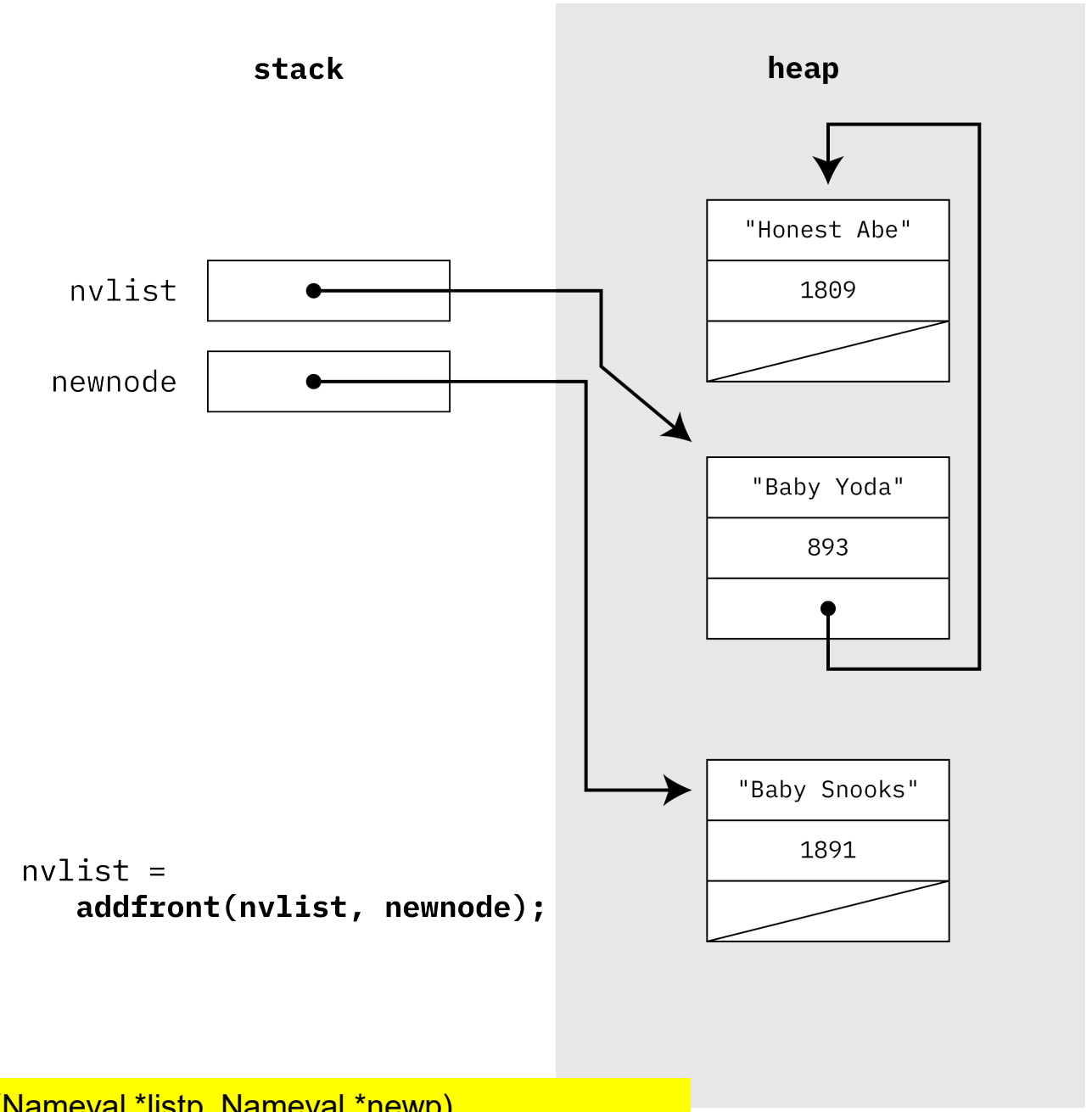
nvlist =
```
   addfront(nvlist, newnode);
```



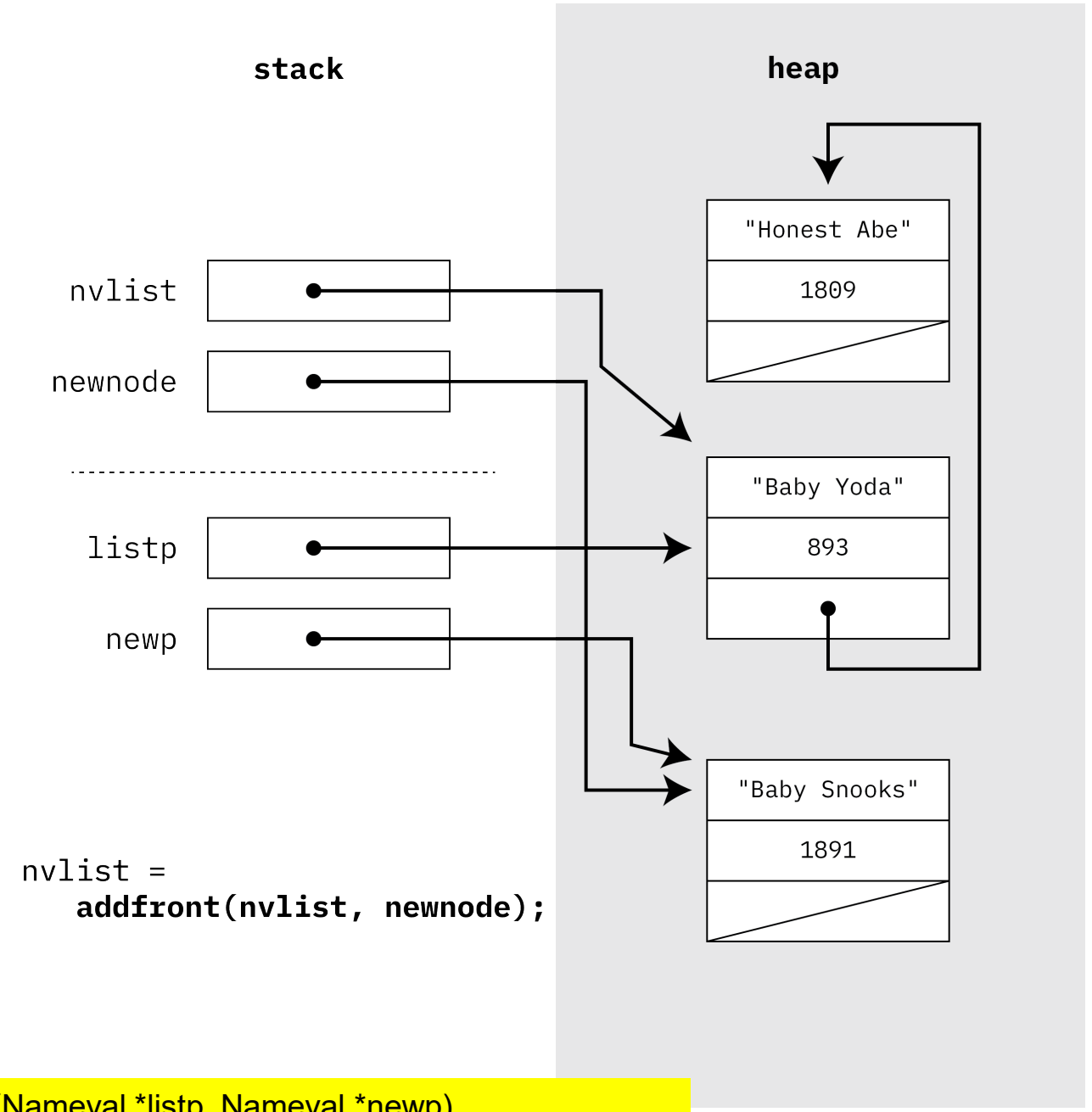

```
newnode = newitem(
  strdup("Baby Snooks",
  1891
);
```



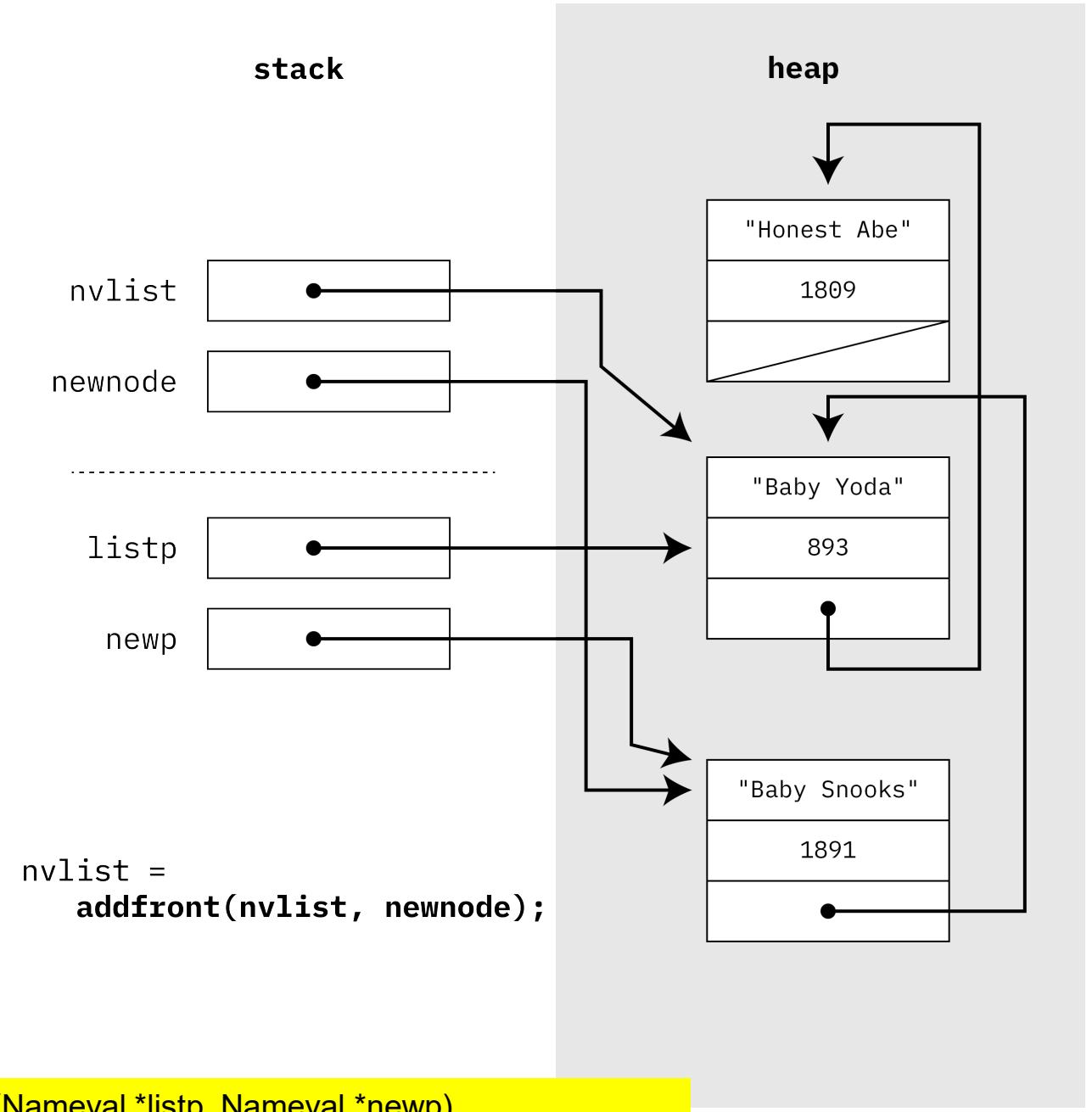

```
newnode = newitem(
  strdup("Baby Snooks",
  1891
);
```



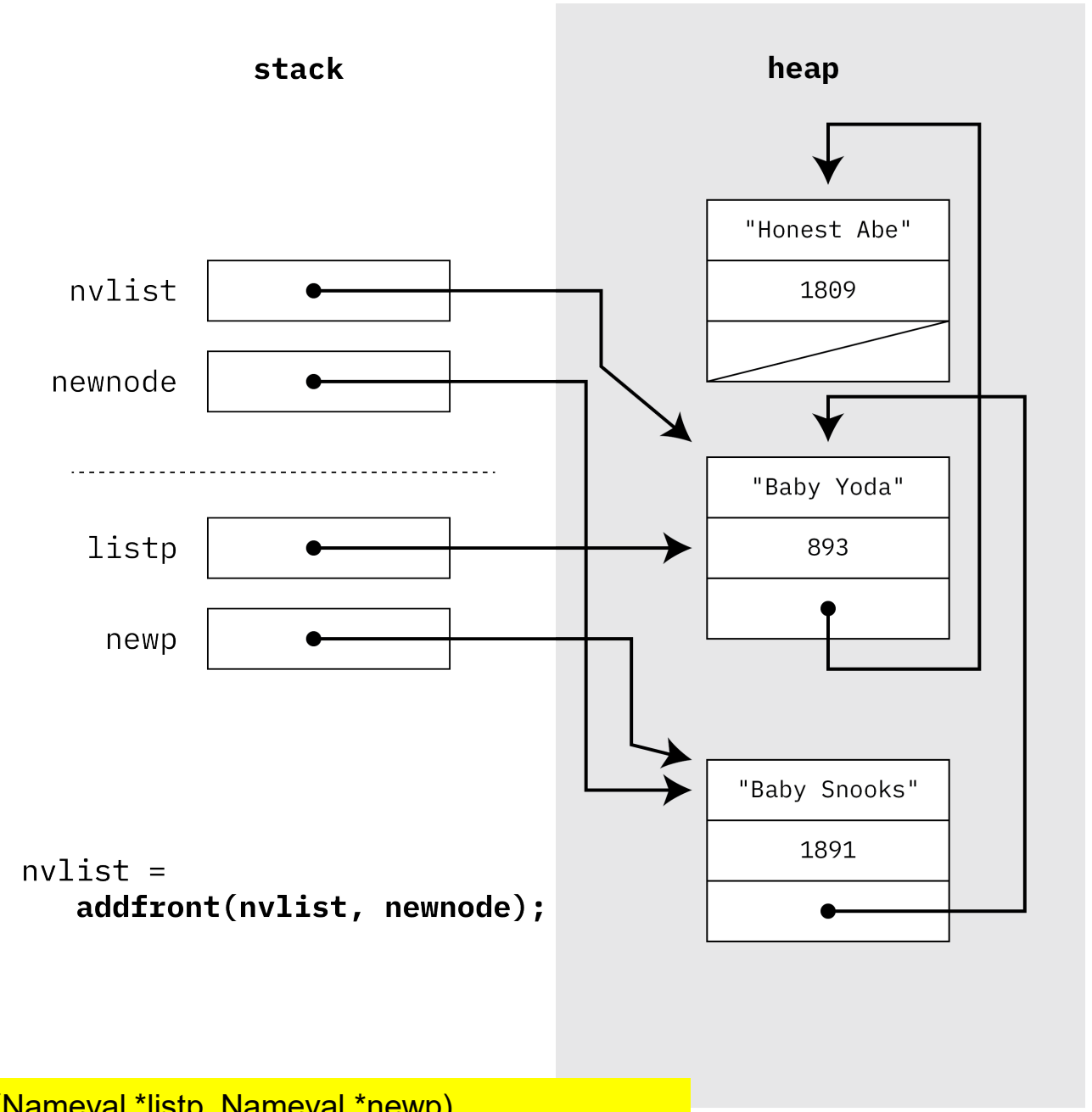

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```



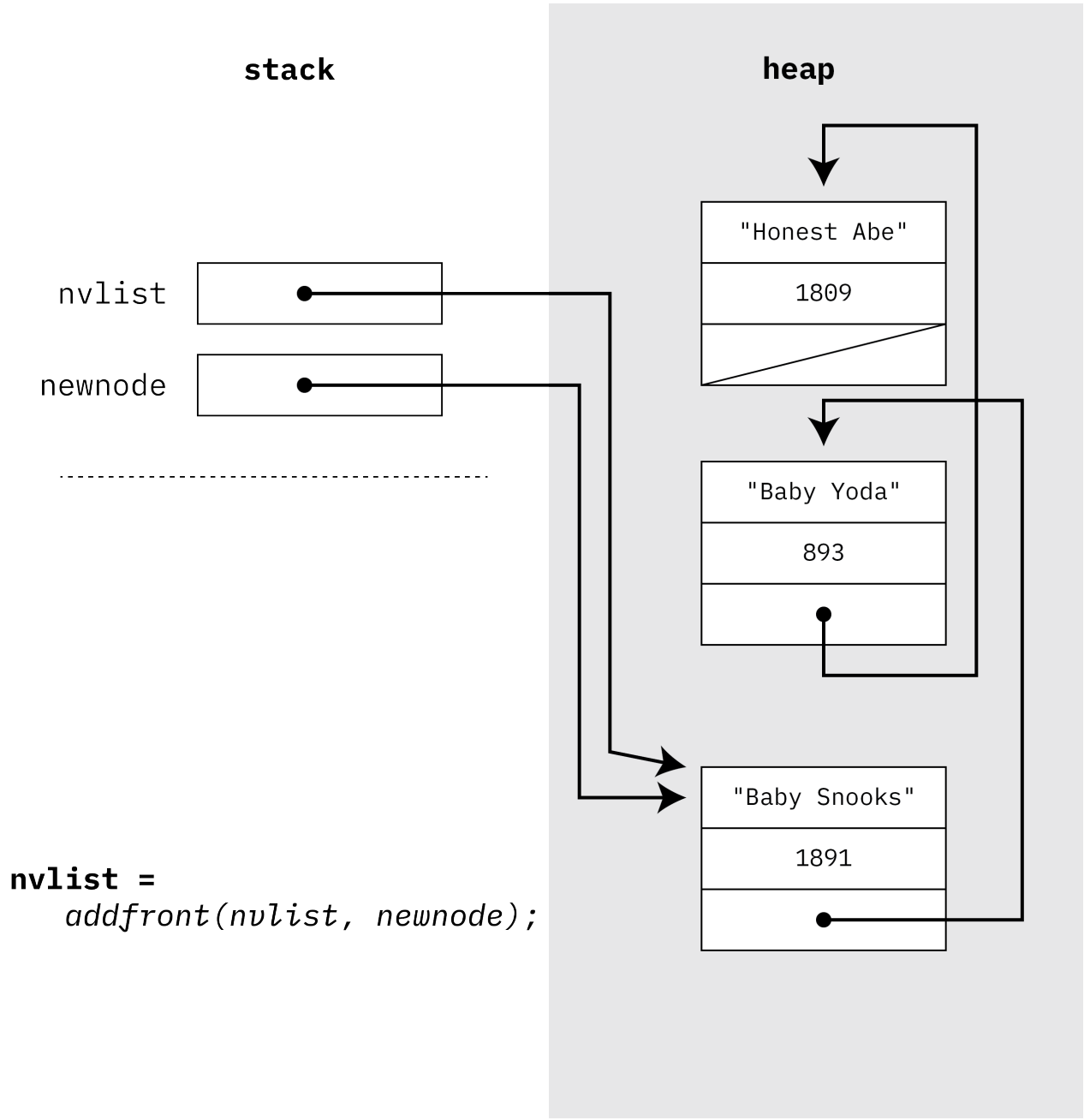

```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```




```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```




```
Nameval *addfront(Nameval *listp, Nameval *newp)
{
    newp->next = listp;
    return newp;
}
```



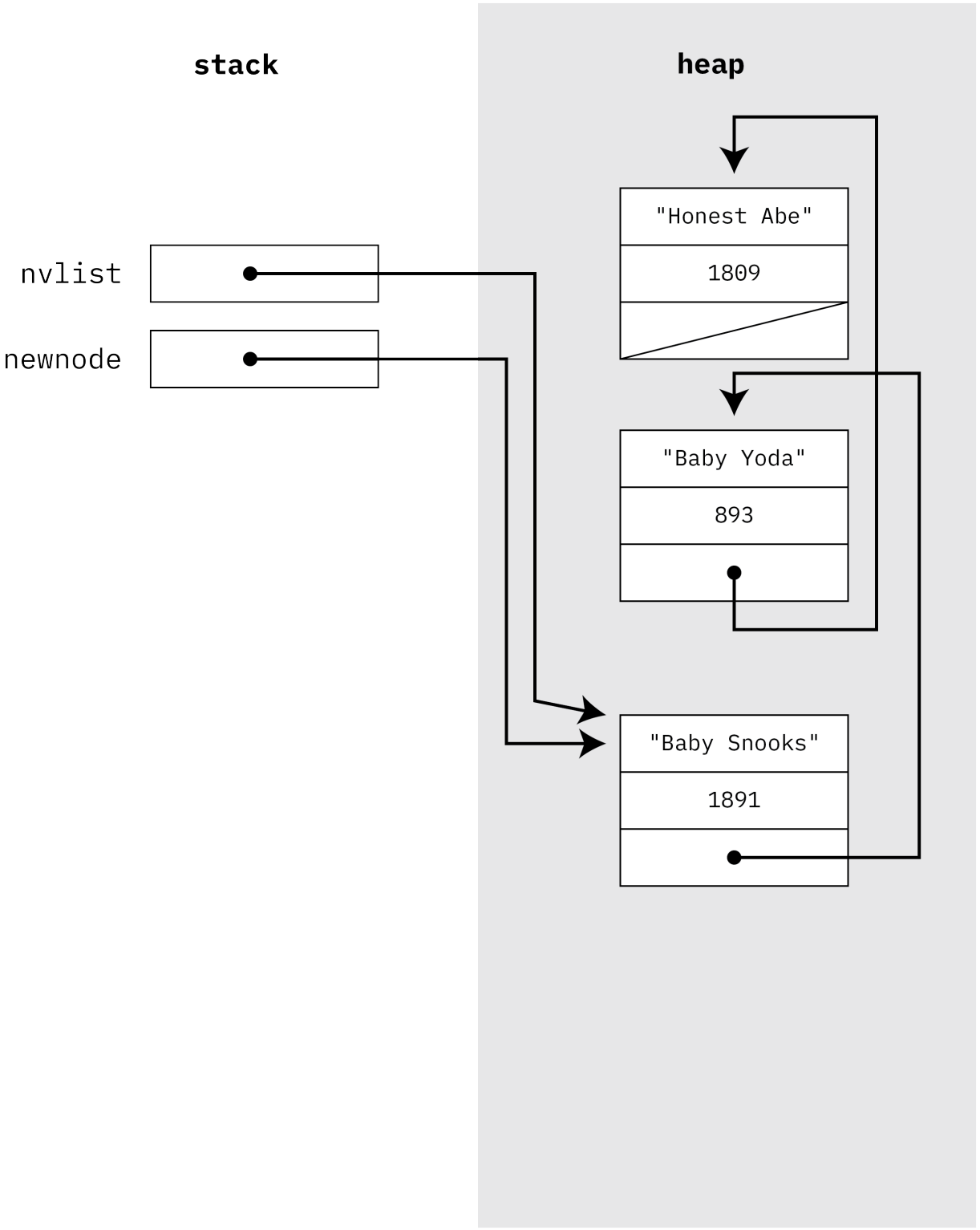
nvlist =
addfront(nvlist, newnode);

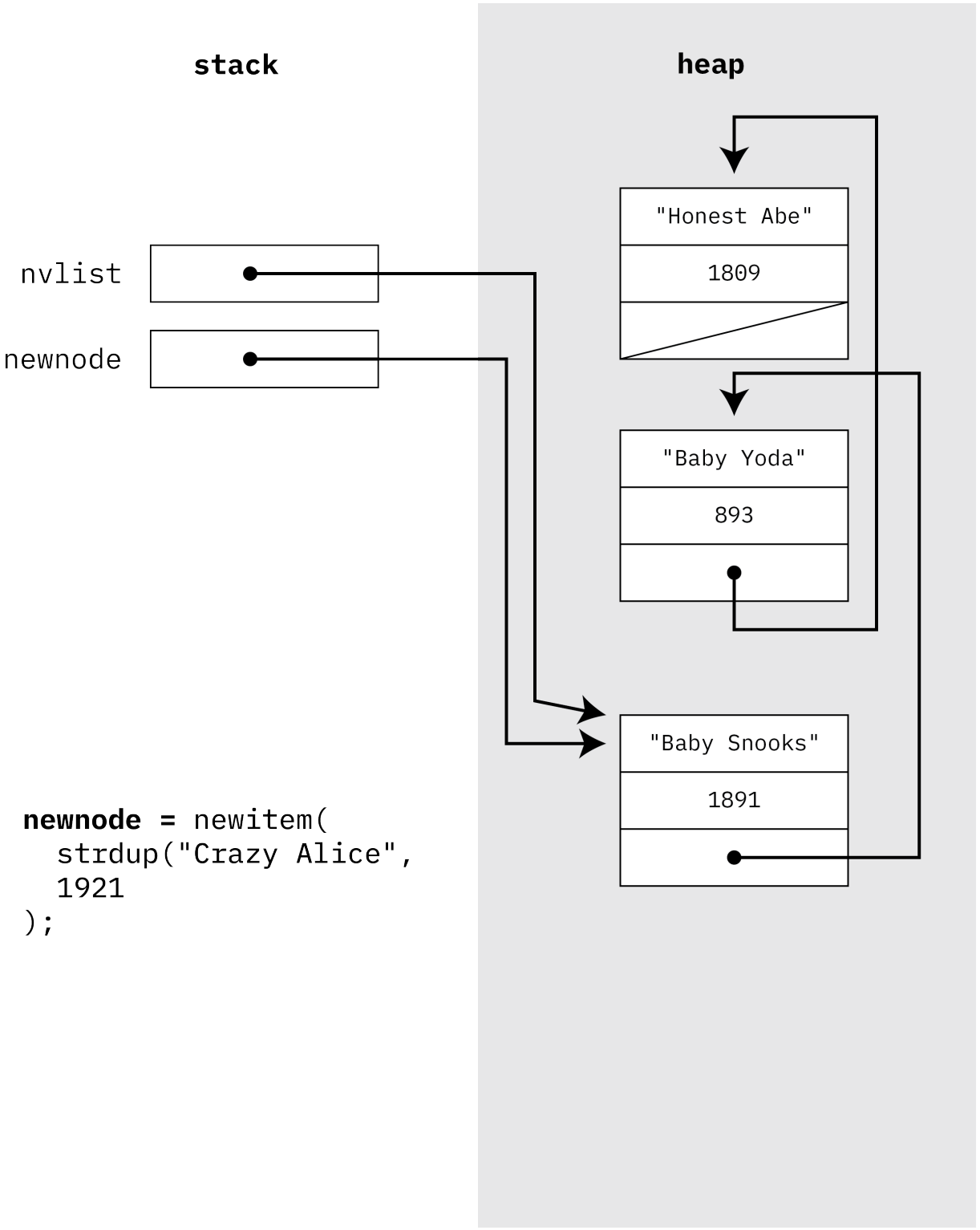
Adding an item to the end

- With a singly-linked list this is an O(n) operation
  - Traverse list until we reach the last node
  - Adjust that node's pointer to indicate the new node.
  - Note that the next field of node created by newitem is already set to NULL.

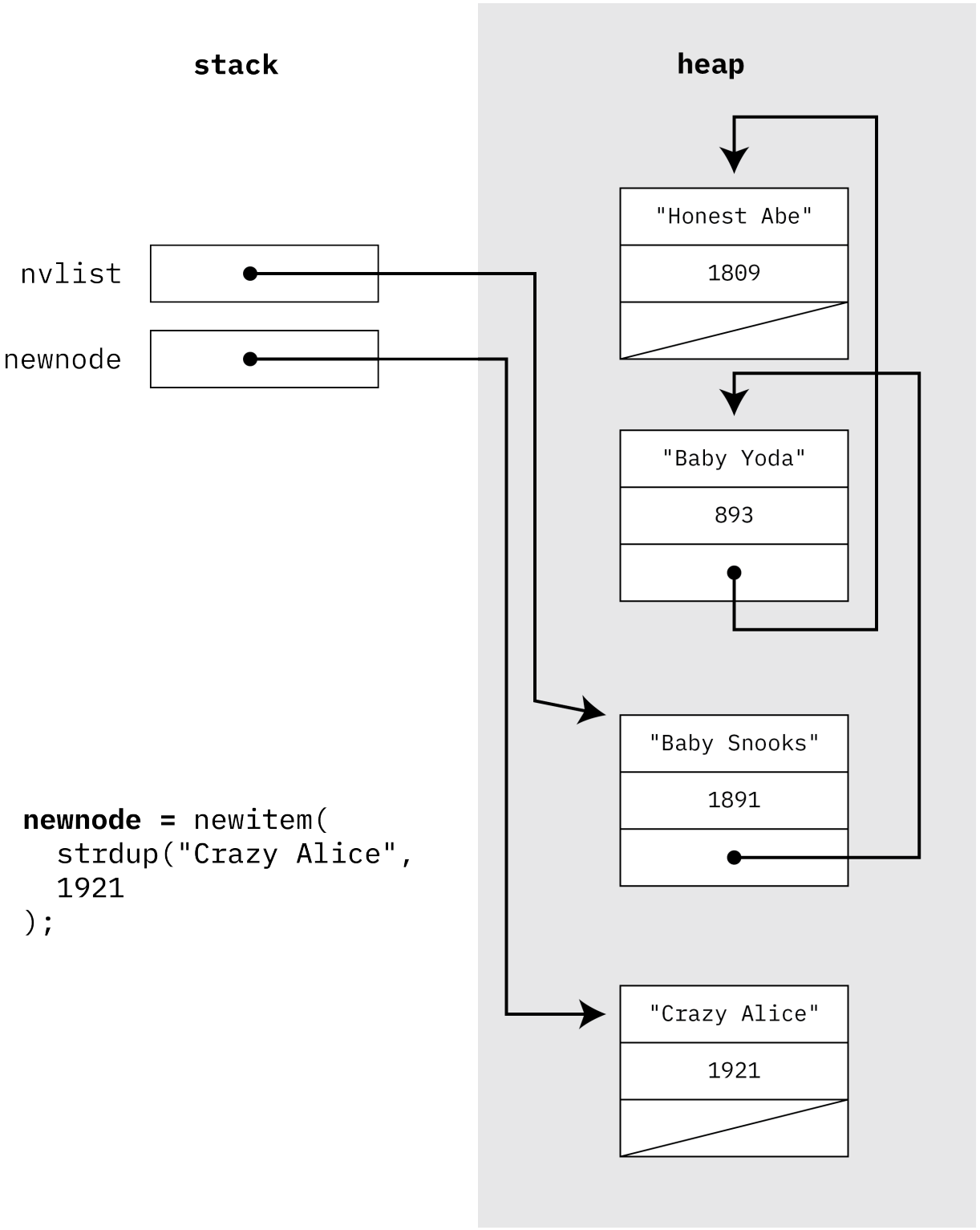
```
Nameval *addend(Nameval *listp, Nameval *newp)
{
    Nameval *p;

    if (listp == NULL) {
        return newp;
    }
    for (p = listp; p->next != NULL; p = p->next)
        ;
    p->next = newp;
    return listp;
}
```

```
...
Nameval *newnode = newitem(strdup("Demetri"), 30);
nvlist = addend(nvlist, newnode);
```

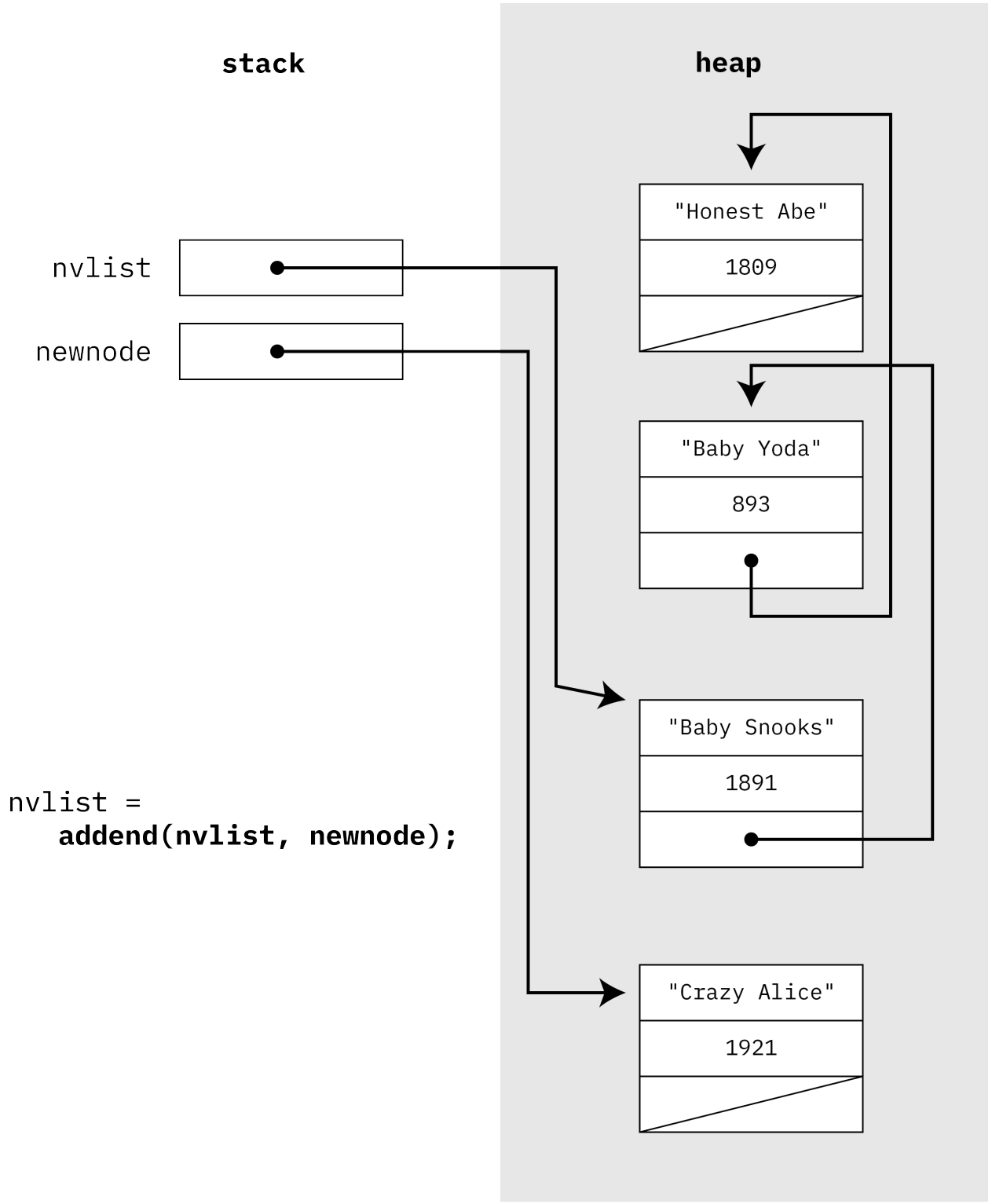

stack
heap
nvlist
newnode
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891

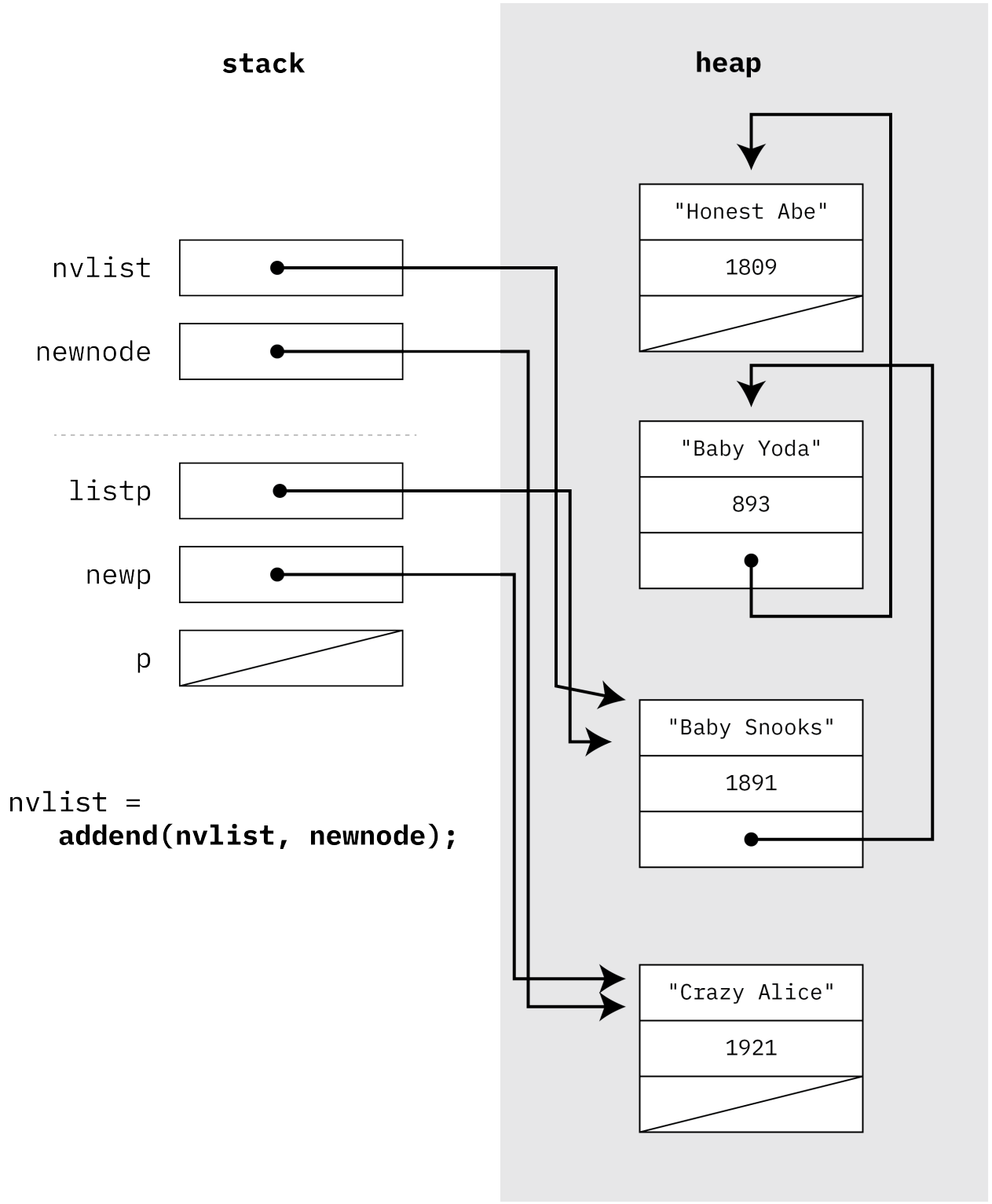


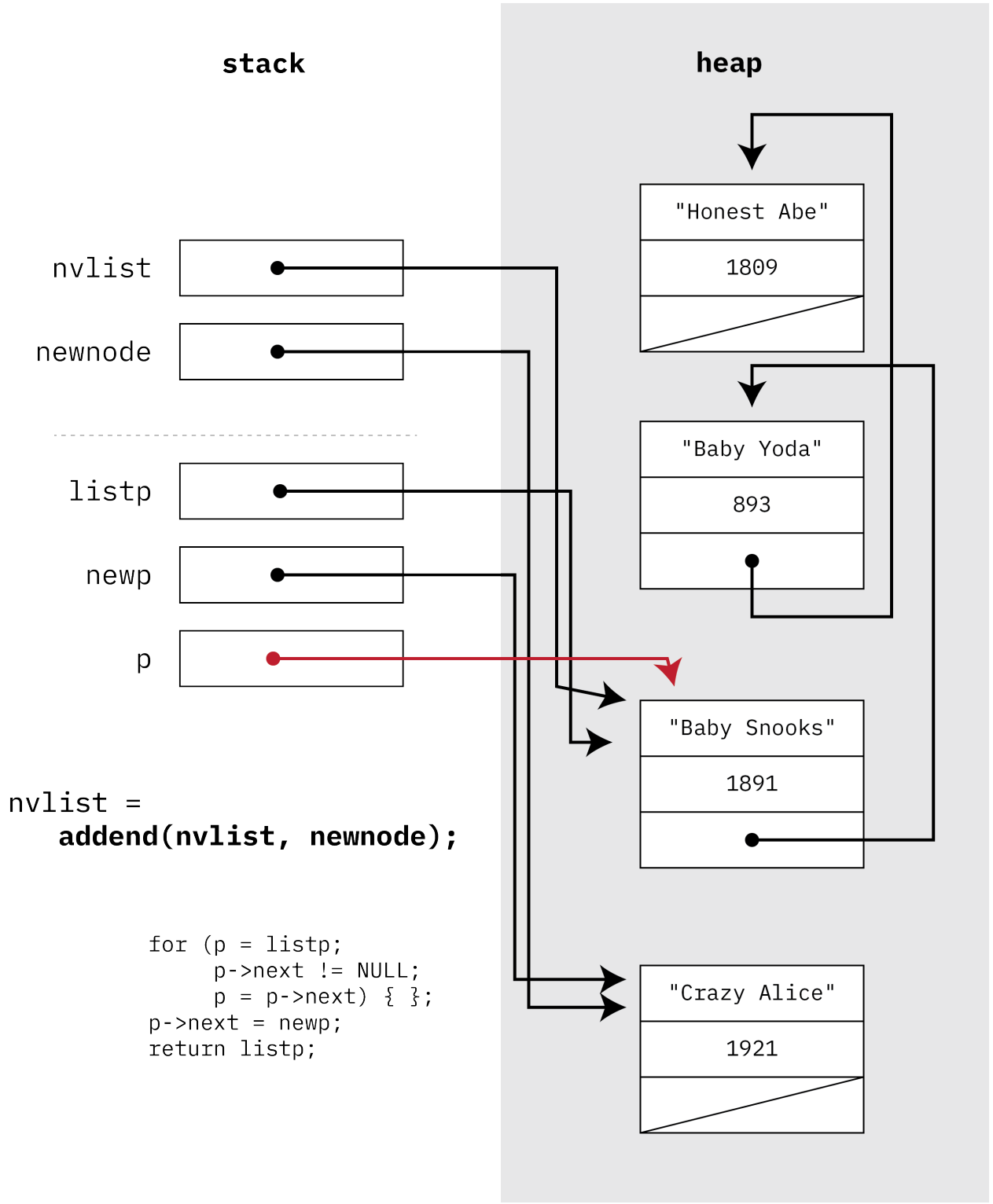

```
newnode = newitem(
  strdup("Crazy Alice",
  1921
);
```

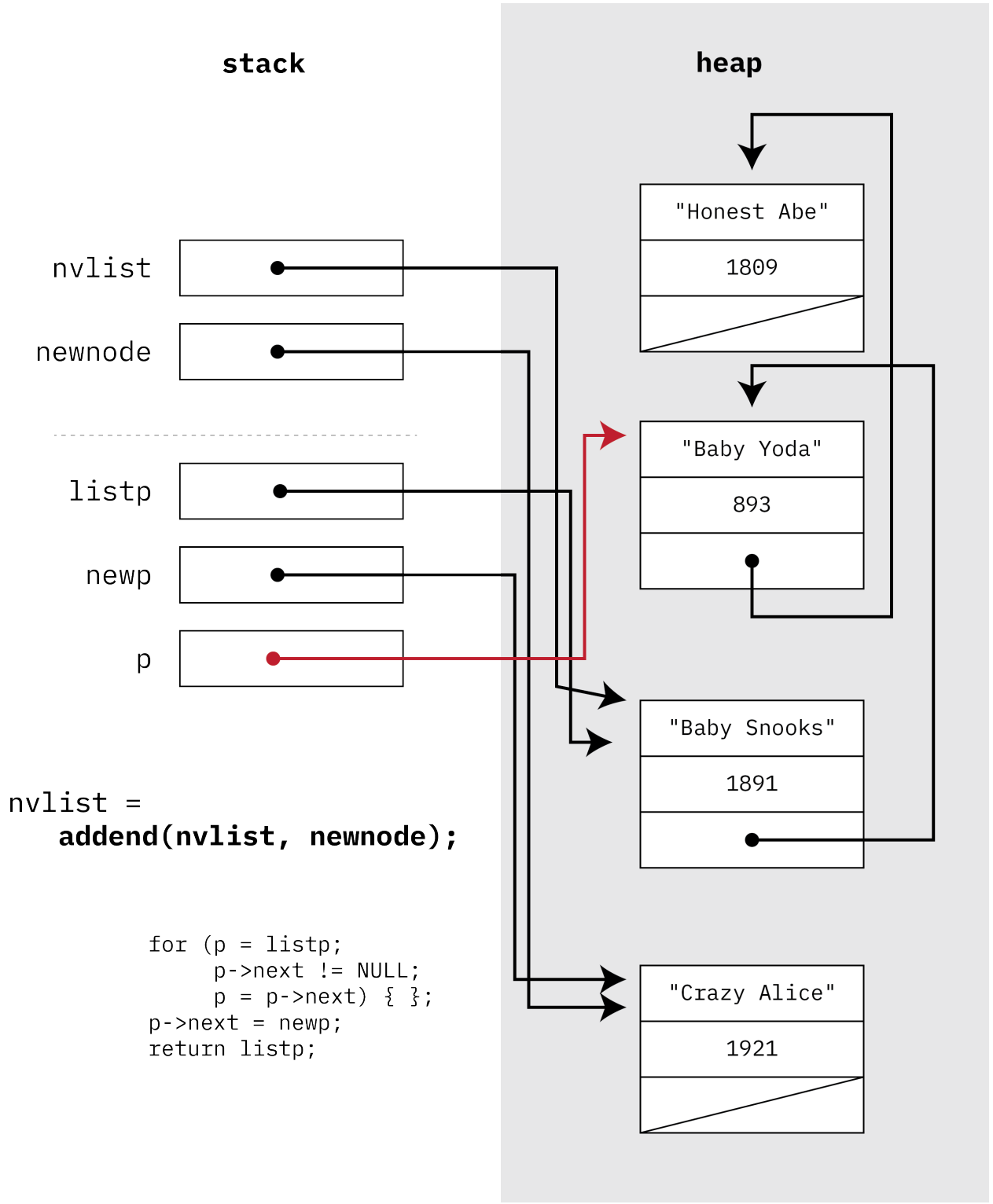


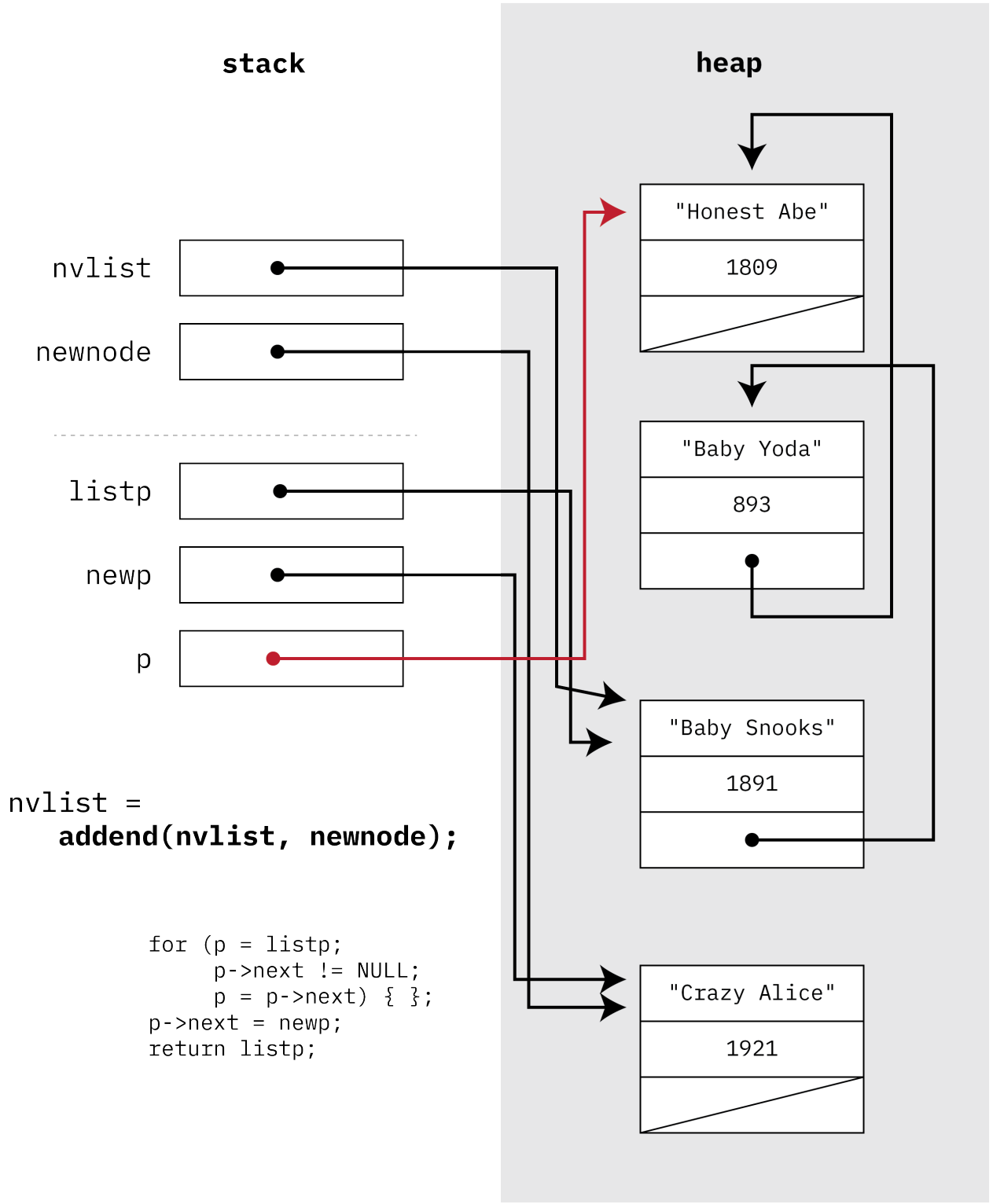

```
newnode = newitem(
  strdup("Crazy Alice",
  1921
);
```

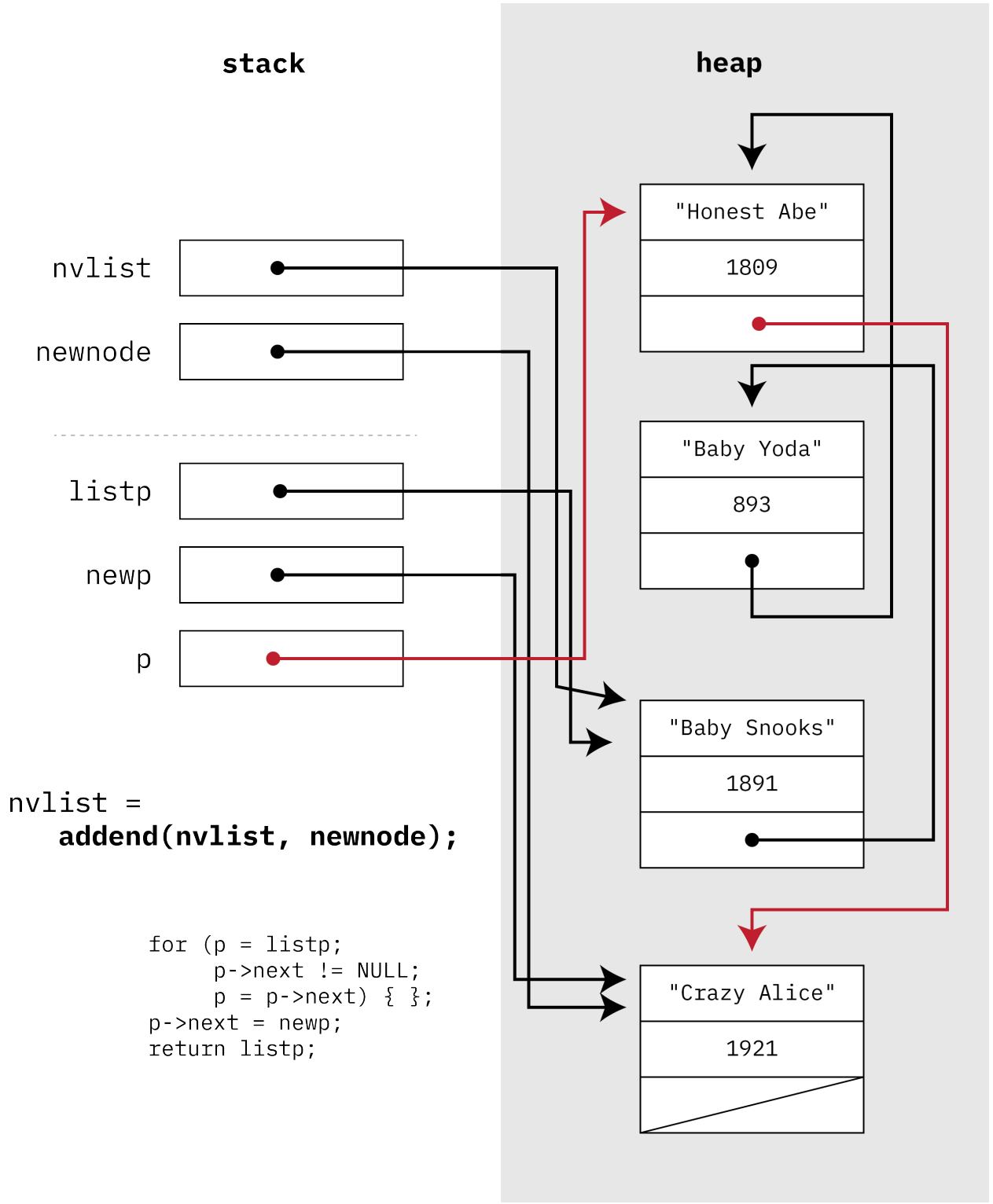

stack
heap
nvlist
newnode
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891
"Crazy Alice"
1921
nvlist =
addend(nvlist, newnode);

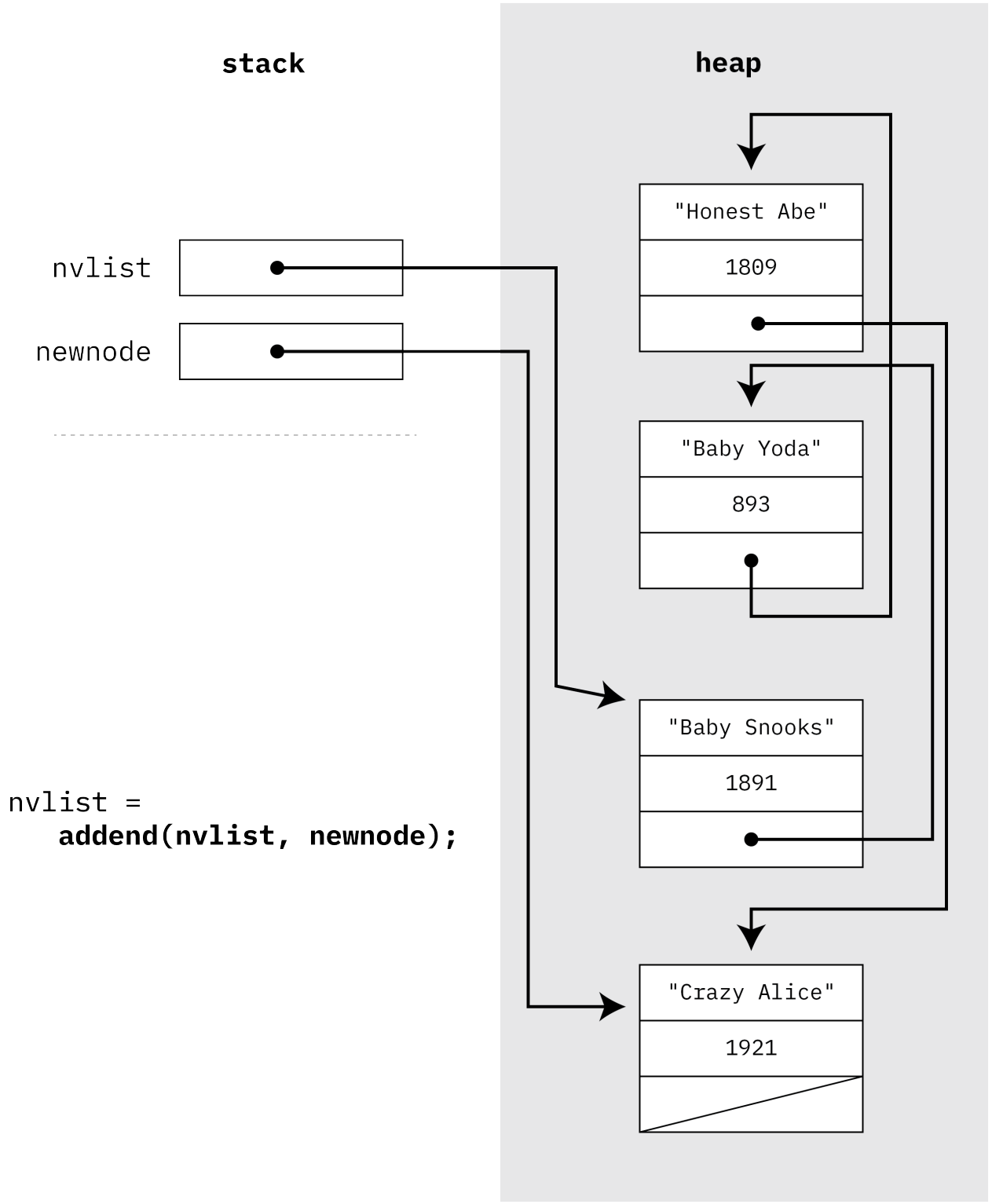

stack
heap
nvlist
newnode
listp
newp
p
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891
"Crazy Alice"
1921
nvlist =
addend(nvlist, newnode);


stack
heap
nvlist
newnode
listp
newp
p
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891
"Crazy Alice"
1921
nvlist =
addend(nvlist, newnode);
for (p = listp;
p->next != NULL;
p = p->next) { };
p->next = newp;
return listp;


stack
heap
nvlist
newnode
listp
newp
p
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891
"Crazy Alice"
1921
nvlist =
addend(nvlist, newnode);
for (p = listp;
p->next != NULL;
p = p->next) { };
p->next = newp;
return listp;


stack
heap
nvlist
newnode
listp
newp
p
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891
"Crazy Alice"
1921
nvlist =
addend(nvlist, newnode);
for (p = listp;
p->next != NULL;
p = p->next) { };
p->next = newp;
return listp;


stack
heap
nvlist
newnode
listp
newp
p
"Honest Abe"
1809
"Baby Yoda"
893
"Baby Snooks"
1891
"Crazy Alice"
1921
nvlist =
addend(nvlist, newnode);
for (p = listp;
p->next != NULL;
p = p->next) { };
p->next = newp;
return listp;




```
nvlist =
   addend(nvlist, newnode);
```

Find an item

- As with adding to the end, we have an operation that is O(n)
  - Unlike a sorted array, binary search does not work on list.
  - However, the code is uncomplicated and its main loop is similar to that in addend.
- The function returns the node even though it is searching on the name
  - If the function succeeds, the return value will be a memory location on the list which can be dereferenced.
  - Otherwise the return value is NULL (i.e., the lookup failed)

```
Nameval *lookup(Nameval *listp, char *name)
{
    for ( ; listp != NULL; listp = listp->next) {
        if (strcmp(name, listp->name) == 0) {
            return listp;
        }
    }
    return NULL;
}
```

```
Nameval *t = lookup(nvlist, "Bobby");
if (t) {
    printf("%d\n", t->value);
} else {
    printf("Couldn't find the value\n");
}
```

An observation about lists

- Many other operations on list have a similar structure
  - Traverse through the list...
  - ... and while doing so, compute some value / perform some comparison / etc.
  - After traversing the list, return some node address
- One approach is to write many such functions with this structure.
- Another approach is to write a more general-purpose function…
  - which traverses through the list…
  - … and applies some function to each element in the list.
  - Let's call this function **apply**
  - It will take three arguments (the list; a function to be applied to each element on the list; and an argument for that function)

apply

```
/* apply: execute fn for each element of listp */

void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next) {
        (*fn)(listp, arg);  /* call the function */
    }
}
```

```
void (*fn)(Nameval*, void*),
```

Declare fn to be a pointer to a void-valued function
(i.e., it is a variable that holds the address of a function
that returns void).

Such a function takes two arguments: an address to a Nameval
(list element) and a void * (a generic point to an argument
for the function being passed in).

example: printing out all elements

```
/* apply: execute fn for each element of listp */

void apply(Nameval *listp, void (*fn)(Nameval*, void*), void *arg)
{
    for ( ; listp != NULL; listp = listp->next) {
        (*fn)(listp, arg);  /* call the function */
    }
}
void printnv(Nameval *p, void *arg)
{
    char *fmt;

    fmt = (char *) arg;
    printf(fmt, p->name, p->value);
}
```

```
apply(nvlist, printnv, "%s: %x\n");
```

example: count of all elements

```
void inccounter(Nameval *p, void *arg)
{
    int *ip;

    /* p is not used -- all we care about is that this function
     * is called once per node.
     */
    ip = (int *)arg;
    (*ip)++;   /* Note the parentheses!!! */
}
```

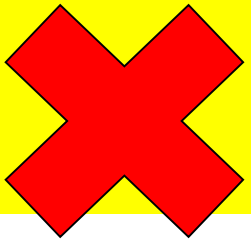
```
int n;

n = 0;
apply(nvlist, inccounter, &n);
printf("%d elements in nvlist\n", n);
```

Deleting elements from the list

- We have yet to see the use of free in the management of our lists
- Let's take the simplest case first: deleting the whole list
  - Here we must be rather careful
  - We cannot free an element if we need to dereference that same element later.
  - Also: free may itself modify the newly deallocated memory
- Must make good use of temporary variables

freeing the list

```
void bad_freeall(Nameval *listp)
{
    for ( ; listp != NULL; listp = listp->next ) {
        /* What is the value of listp->next after the next
         * operation?
         */
        free(listp);
    }
}
```



```
void freeall(Nameval *listp)
{
    Nameval *next;

    for ( ; listp != NULL; listp = next ) {
        next = listp->next;
        /* assume here the listp->name is freed someplace else */
        free(listp);
    }
}
```