G52CPP
C++ Programming
Lecture 8

Dr Jason Atkin

Last lecture

```
#include <cstdio>

#pragma pack(1)

struct A { int i; char c; };
union B { int i; char c; };
struct C { int i, j; char c; };
union D { int i; A a; };

int main( int argc, char** argv )
{
   A a;   B b;   C c;   D d;
   printf( "char: %d %d\n", sizeof(char), sizeof(a.c) );
   printf( "int:  %d %d\n", sizeof(int) , sizeof(a.i) );
   printf( "A: %d %d\n", sizeof(A), sizeof(a) );
   printf( "B: %d %d\n", sizeof(B), sizeof(b) );
   printf( "C: %d %d\n", sizeof(C), sizeof(c) );
   printf( "D: %d %d\n", sizeof(D), sizeof(d) );
   return 0;
}
```

Example:

```
char : 1 ?
 int : 4 ?
   A : ? ?
   B : ? ?
   C : ? ?
   D : ? ?
```

Last lecture

```
#include <cstdio>

#pragma pack(1)

struct A { int i; char c; };
union B { int i; char c; };
struct C { int i, j; char c; };
union D { int i; A a; };

int main( int argc, char** argv )
{
   A a;   B b;   C c;   D d;
   printf( "char: %d %d\n", sizeof(char), sizeof(a.c) );
   printf( "int:  %d %d\n", sizeof(int) , sizeof(a.i) );
   printf( "A: %d %d\n", sizeof(A), sizeof(a) );
   printf( "B: %d %d\n", sizeof(B), sizeof(b) );
   printf( "C: %d %d\n", sizeof(C), sizeof(c) );
   printf( "D: %d %d\n", sizeof(D), sizeof(d) );
   return 0;
}
```

Example:

```
char : 1 1
 int : 4 4
   A : 5 5
   B : 4 4
   C : 9 9
   D : 5 5
```

This lecture

- Dynamic memory allocation
- Memory re-allocation to grow arrays
- Linked lists

Allocating memory from the heap

Process structure in memory

Stack

Data area that grows downwards towards the heap

LIFO data structure, for local variables and parameters

Heap

Data area that grows upwards towards stack

Specially allocated memory (malloc, free, …, probably new, delete)

Data and BSS segment

Read-only:	Constants	String literals
Read/write:	Global variables	Static local variables

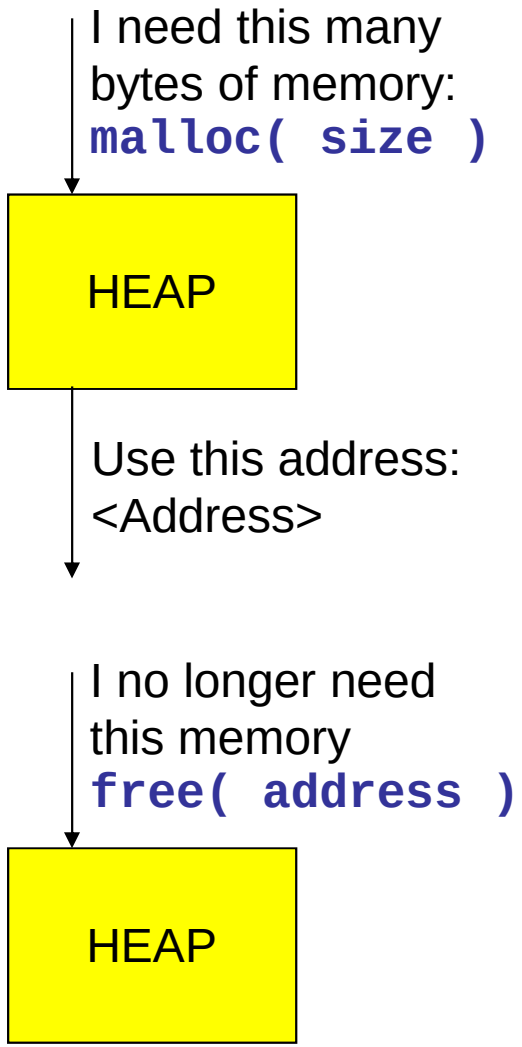
Code (or text) segment

The program code

The heap and malloc()

- A big store of memory
- You can ask for memory from it using **`malloc()`**, **`calloc()`**, **`realloc()`** functions
- You can tell it that you no longer need memory it has given to you

  `free()` function



It gives you generic memory

- **`malloc()`** etc will allocate bytes of memory
- They will not (directly) allocate a string for you, or an array, or an **`int`** (unlike **`new`** in C++ or Java)
- You should store the returned address in a pointer of the type you wish to use it as
  - i.e. treat the memory as if it was that type
- **`malloc()`** returns a **`void*`**
  - In C there is an implicit conversion to/from **`void*`**
  - In C++ you need to cast the returned value
- You should **`#include <cstdlib>`**
  - Declares the various functions

5 steps to dynamic memory bliss

Step 1: Work out how much memory you need to allocate
- Remember the `sizeof()` operator!

Step 2: Ask for that amount of memory
- Use `malloc( memory_size )`

Step 3: Store the returned pointer e.g. :

```
int* pInt = (int*)malloc( sizeof(int) );
```

Note: C++ needs the cast, C does not

Step 4: Use the memory through the pointer, as if it was the correct type

```
*pInt = 5; (*pInt)++; *pInt += 12;
```

Step 5: When finished, free the memory

```
free( pInt );
```

malloc, calloc and realloc

- All of these functions return **`NULL`** on failure

```
void* malloc(size_t sz);
```

- Allocate **`sz`** bytes of uninitialised memory

```
void* calloc(size_t count, size_t sz);
```

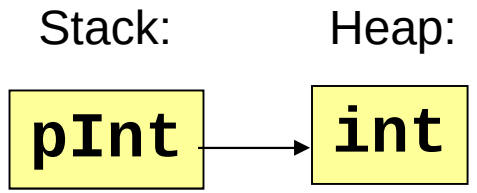
- Allocate memory for **`count`** elements of size **`sz`** each
- The memory is initialised to zeroes!!!

```
void* realloc(void *old_pointer, size_t sz);
```

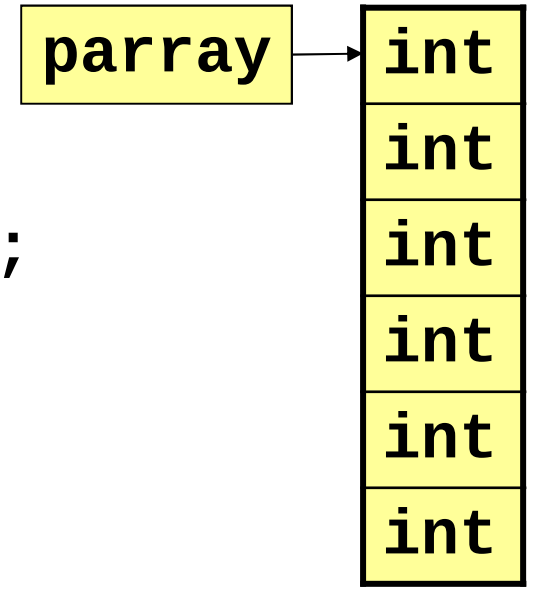
- **`old_pointer`** is a pointer from an existing **`malloc()`**
- If possible, grow or shrink the existing memory allocation to be size **`sz`** bytes
- If not, then allocate new memory for the new size (**`sz bytes`**), copy the bytes of the existing memory to the new address and free the old memory
- If it fails (returns NULL) the old memory will be unchanged

Creating a simple array

```
int* pInt = (int*)malloc( sizeof(int) );
*pInt = 5;
(*pInt)++;
*pInt += 12;
free( pInt );
```



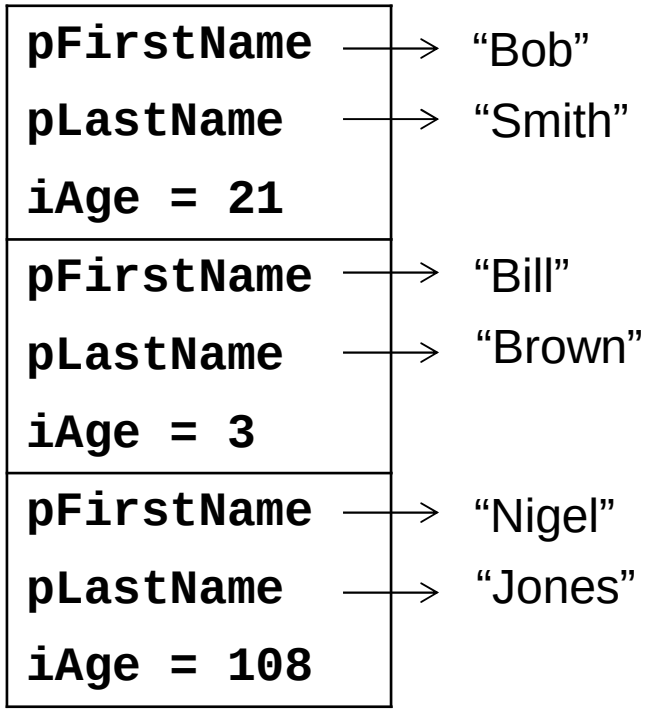
```
int iSize = 6;
int* parray = (int*)malloc(
            iSize * sizeof(int) );
*parray = 3; /* Index 0 */
parray[5] = 5;
free( parray );
```



Storing details in a dynamic array

A program to grow an array…

```
struct Person
{
        char* pFirstName;
        char* pLastName;
        int iAge;
};

Person* g_pArrayPeople = NULL;
int g_iPersonCount = 0;
```



lec8_array_malloced.cpp

```
void GetInput()
{
        char strFirstName[1024];
        char strLastName[1024];
        int iAge = 0;
        … READ IN THE VALUES …
        StorePerson(strFirstName, strLastName, iAge );
}
```

Allocating memory for array

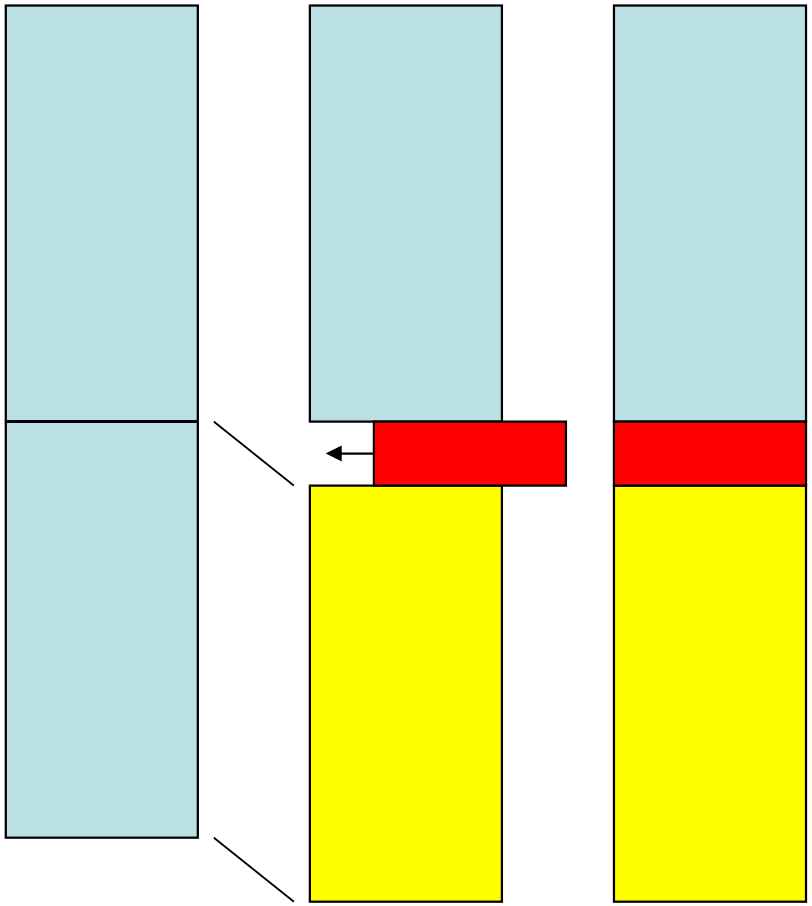
```
void StorePerson(
        char* strFirstName,
        char* strLastName,
        int iAge )
{
  if ( g_pArrayPeople == NULL )
  {
    g_pArrayPeople = (Person*)malloc( sizeof( Person ) );
  }
  else
  {
    g_pArrayPeople = (Person*)realloc( g_pArrayPeople,
                (g_iPersonCount+1) * sizeof( Person ) );
    // Should really check for NULL return!
  }
...
```

Populate the new array entry

```
...
        // Use pointer as if it pointed to an array
        g_pArrayPeople[g_iPersonCount].pFirstName
              = (char*)malloc ( strlen(strFirstName) + 1 );
        strcpy( g_pArrayPeople[g_iPersonCount].pFirstName,
              strFirstName );

        g_pArrayPeople[g_iPersonCount].pLastName =
              (char*)malloc(strlen(strLastName)+1);
        strcpy( g_pArrayPeople[g_iPersonCount].pLastName,
              strLastName );

        g_pArrayPeople[g_iPersonCount].iAge = iAge;

        g_iPersonCount++; /* Increment the count */
}
```

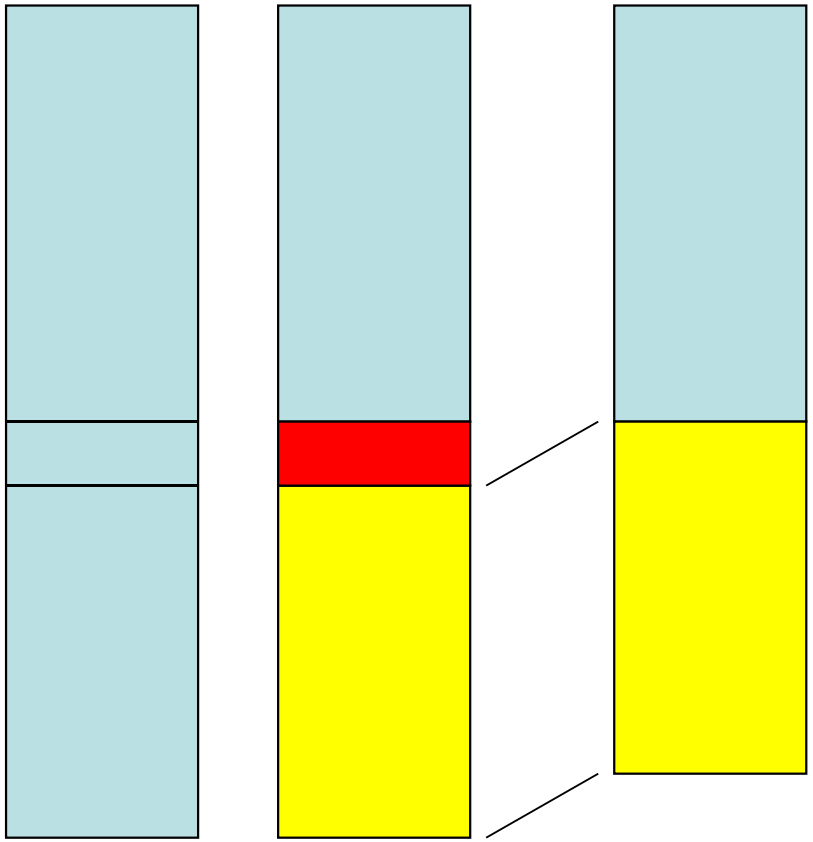
Insertion and deletion

Inserting into an array



Copy the yellow part to later
And add the red part

Deleting from an array



Copy the yellow part to earlier
Overwriting the red part

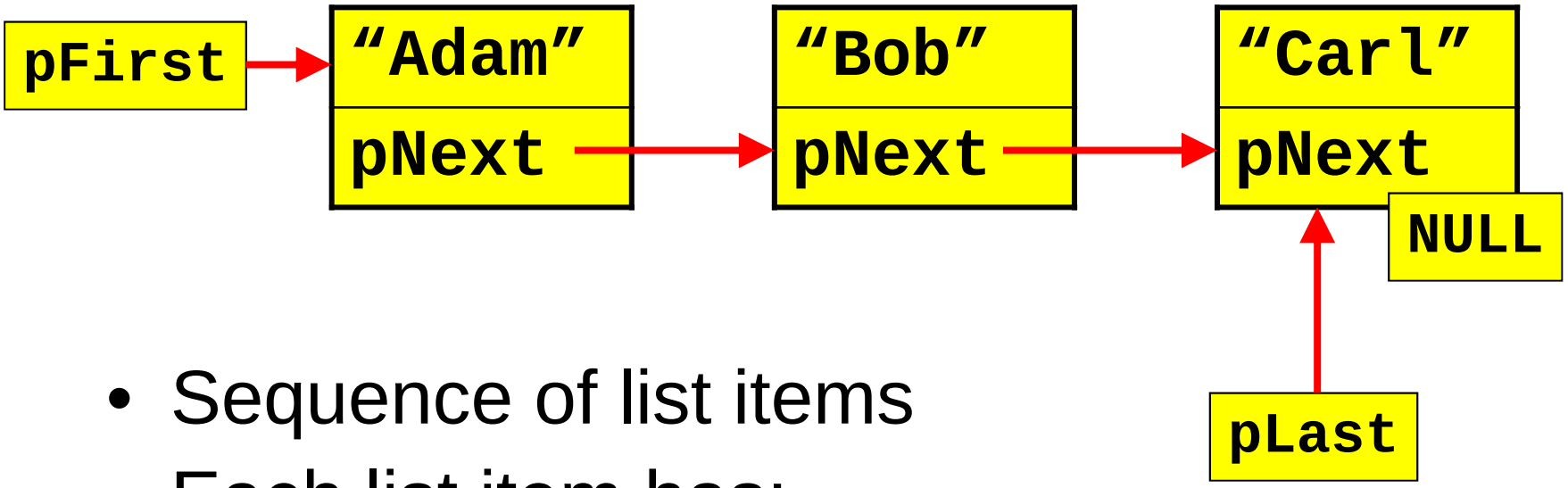
Problems of array insert/delete

- Arrays are far from ideal if:
  - Items have to be inserted in the middle
  - Items need to be added and the array grown
  - Items have to be deleted from the middle

- Vector has the same problem even though it hides it!
- Has to allocate new memory when it grows

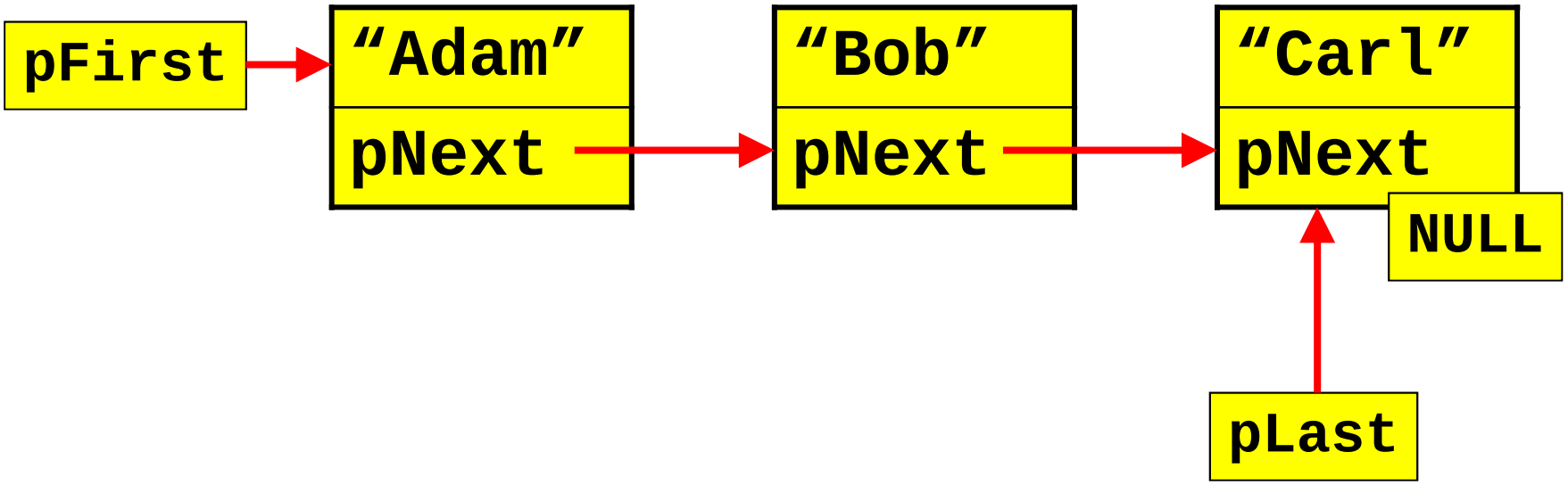
Linked lists

Single-linked list



- Sequence of list items
- Each list item has:
  - Some data
  - A (single) link to the next item in the list
- A pointer to the first item
- A pointer to the last item (optional)

Single-linked list



A definition of a struct : (for the list items)

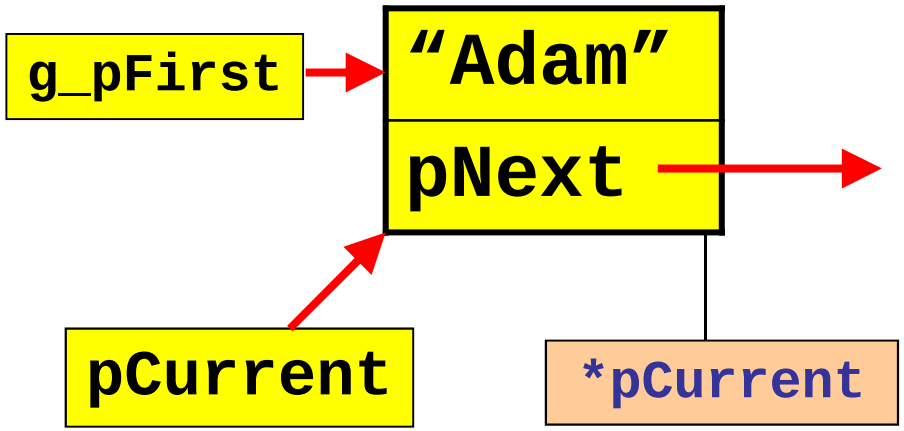
```
struct SLLEntry
{
    SLLEntry* pNext;
    char* pData;
};
```

Some pointers to list items:

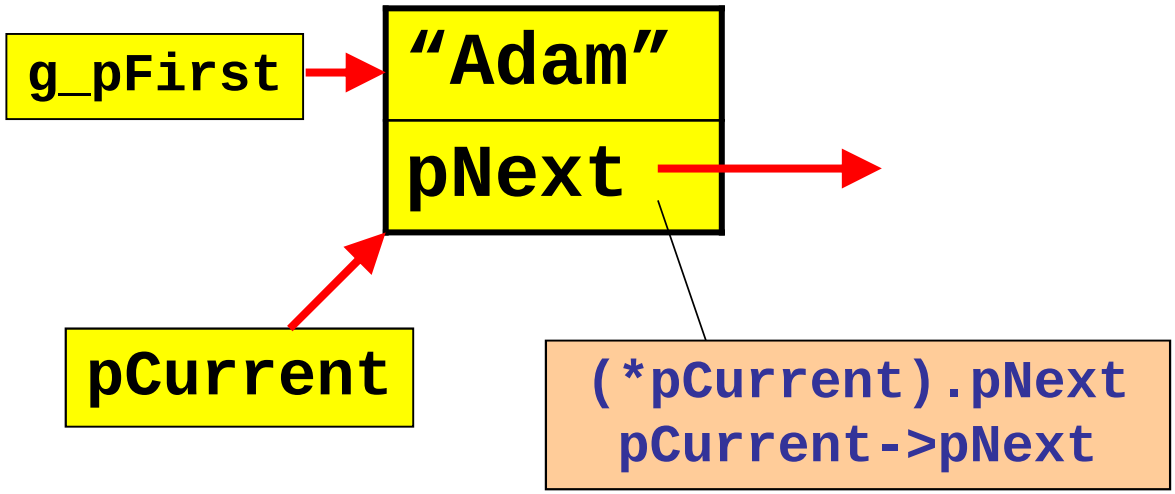
```
SLLEntry* g_pFirst = NULL;
SLLEntry* g_pLast = NULL;
```

Using an existing linked list

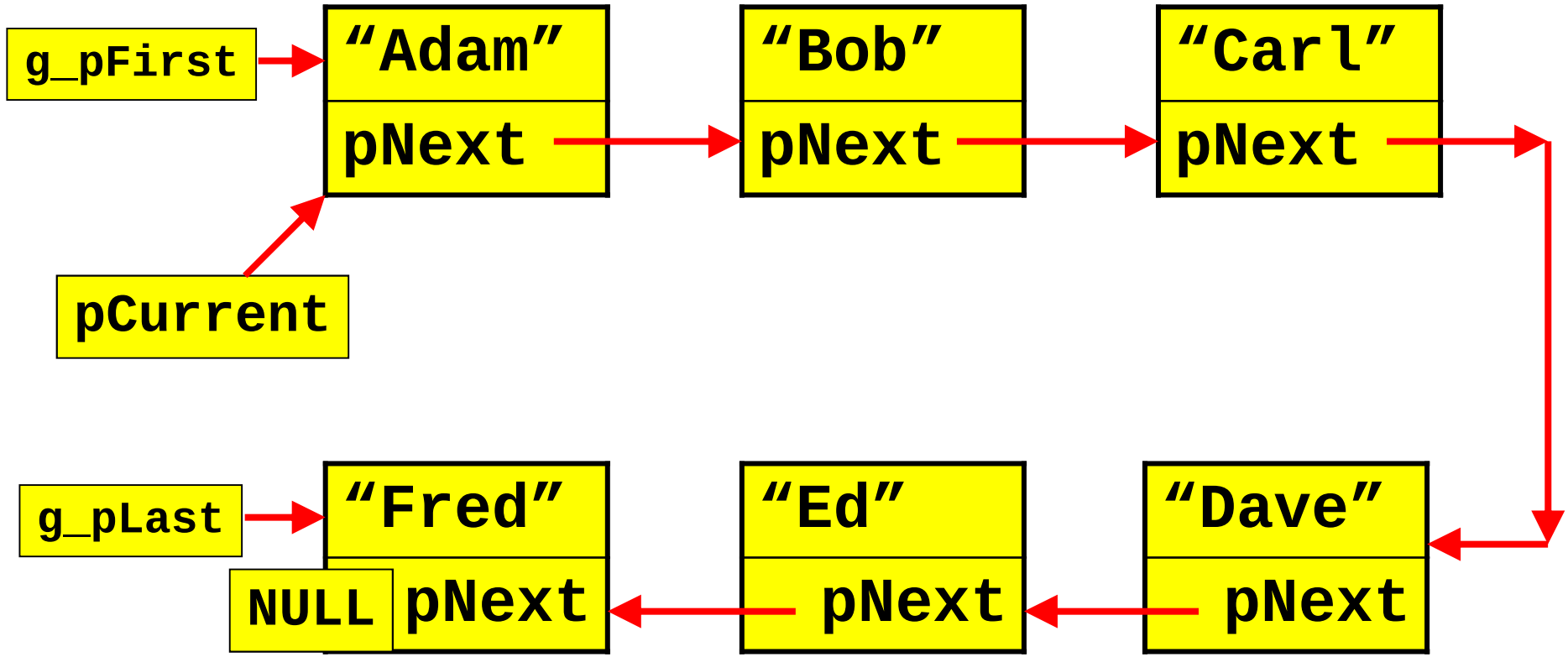
Print all items



Print all items

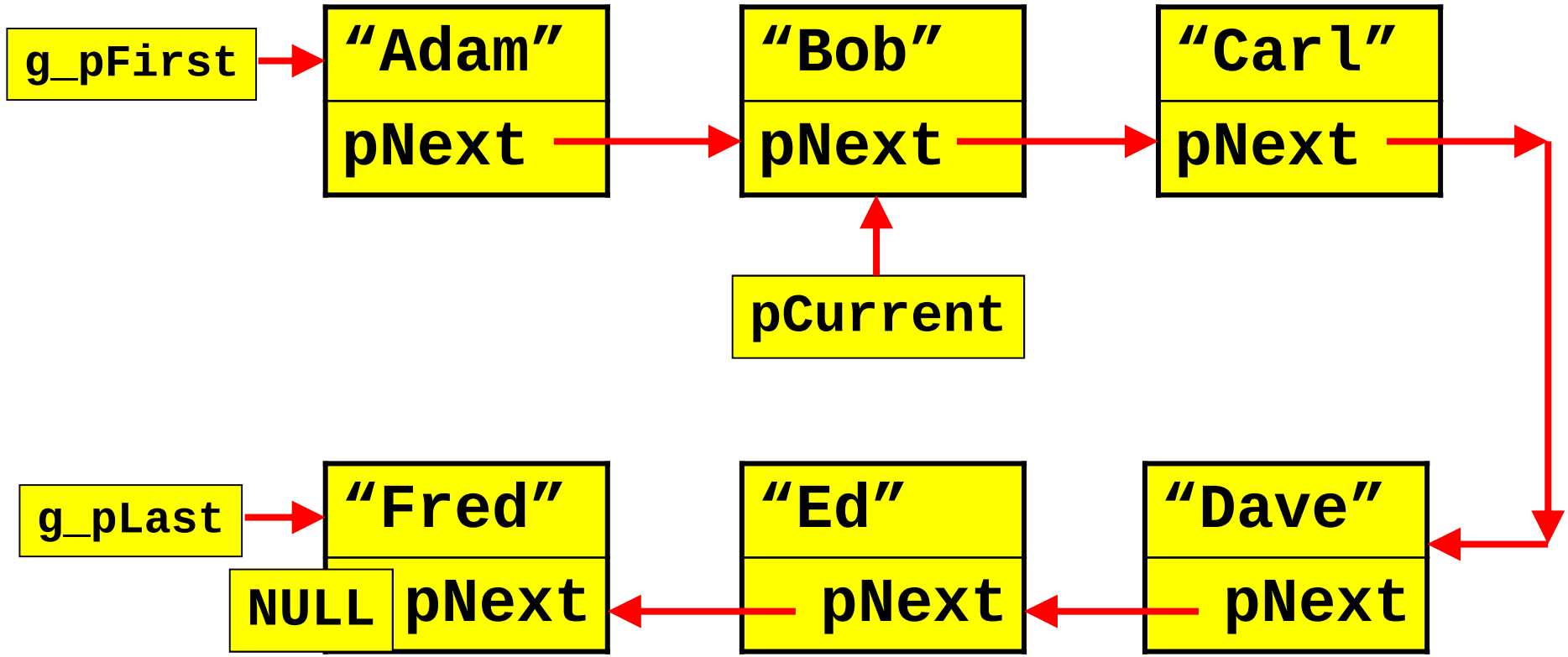


Print all items



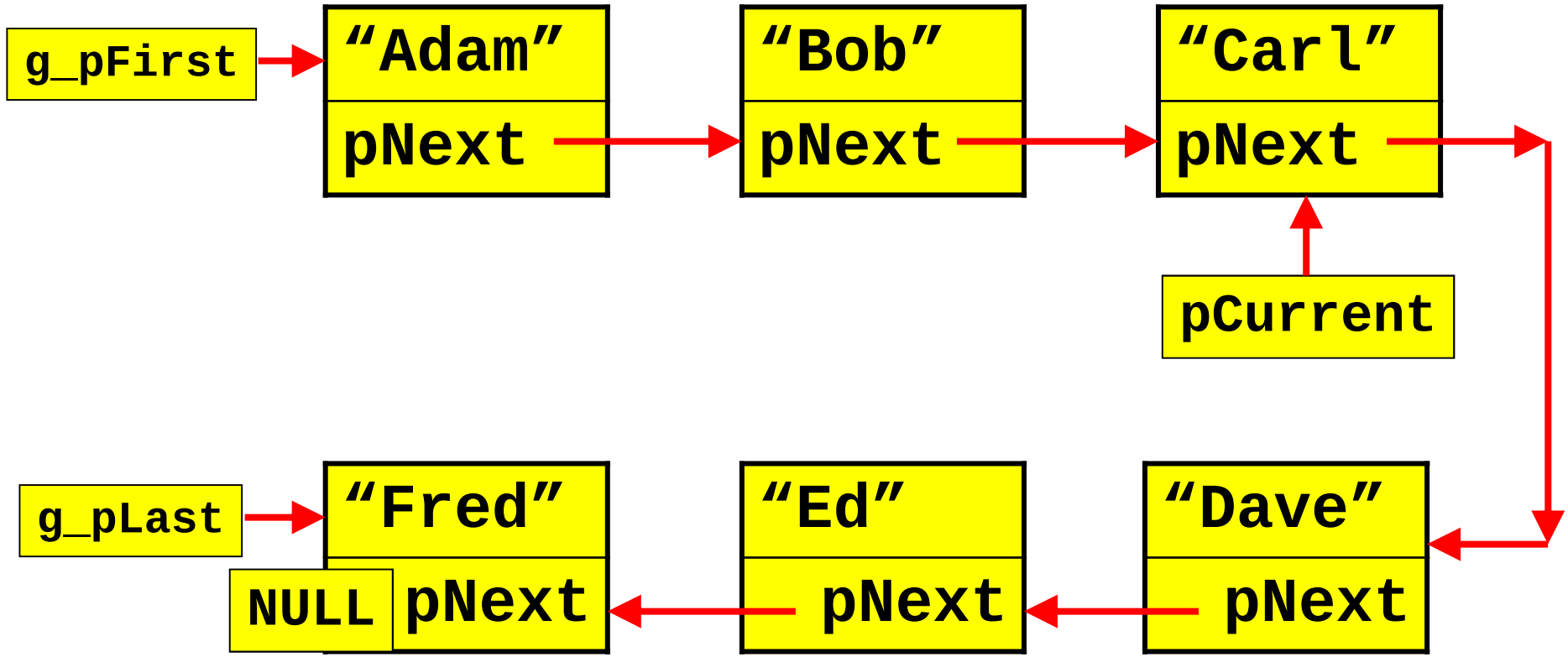

```
pCurrent = g_pFirst; // Initialisation
if ( pCurrent == NULL ) return;  // STOP!
printf( "%s\n", pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Print all items



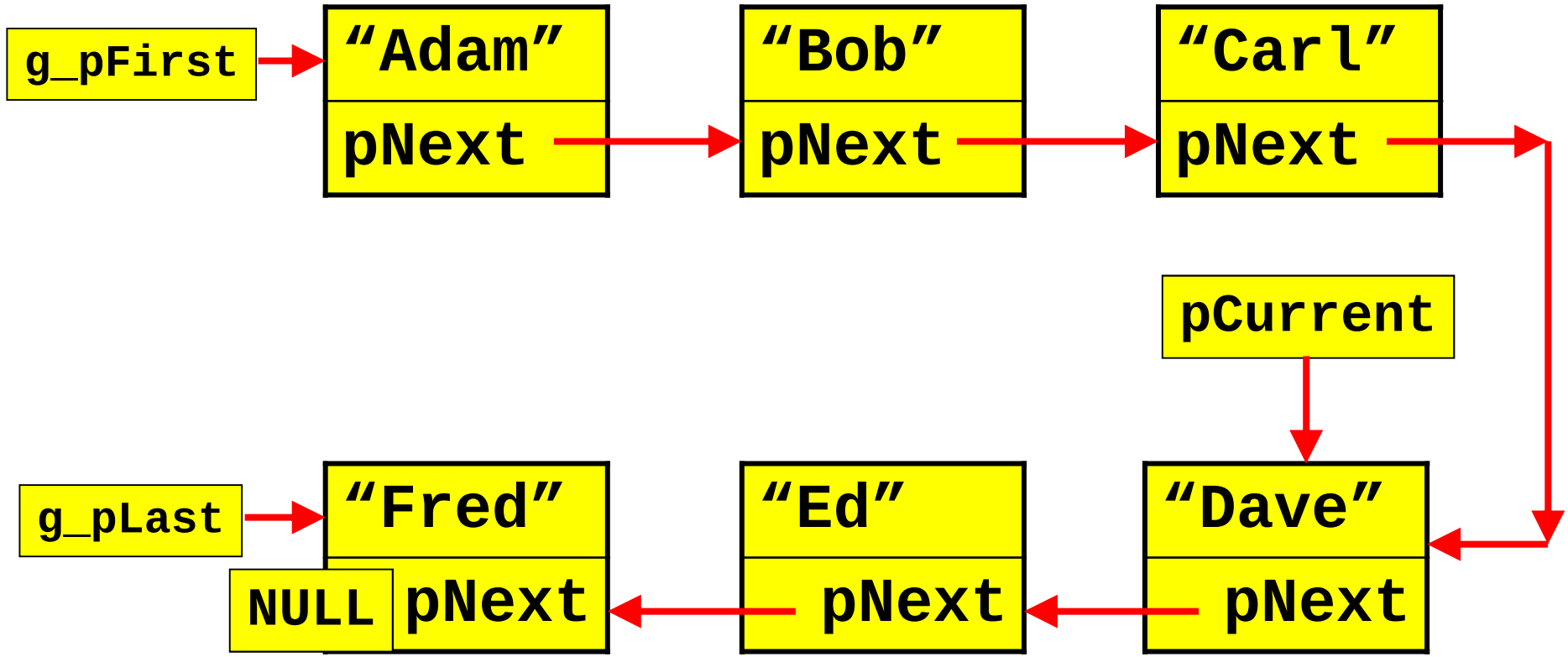

```
if ( pCurrent == NULL ) return;  // STOP!
printf( “%s\n”, pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Print all items



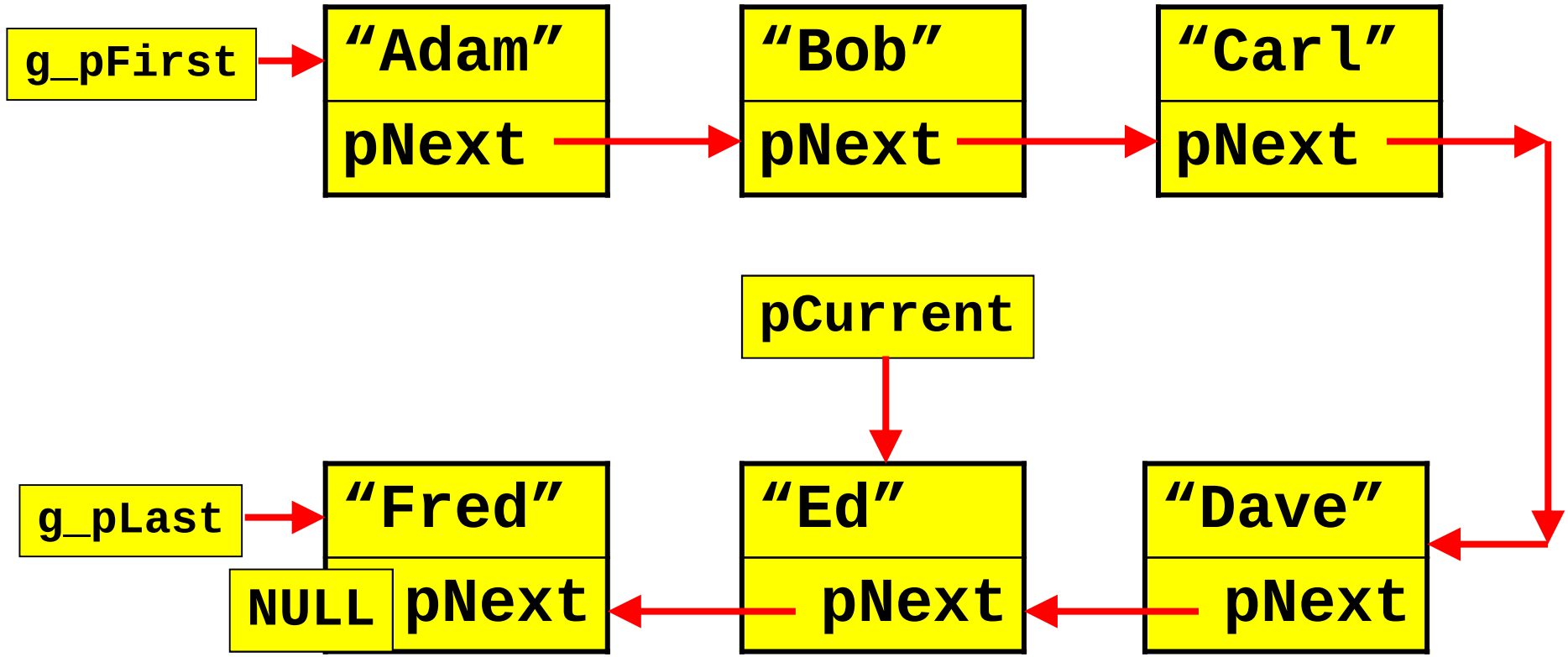

```
if ( pCurrent == NULL ) return;  // STOP!
printf( “%s\n”, pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Print all items



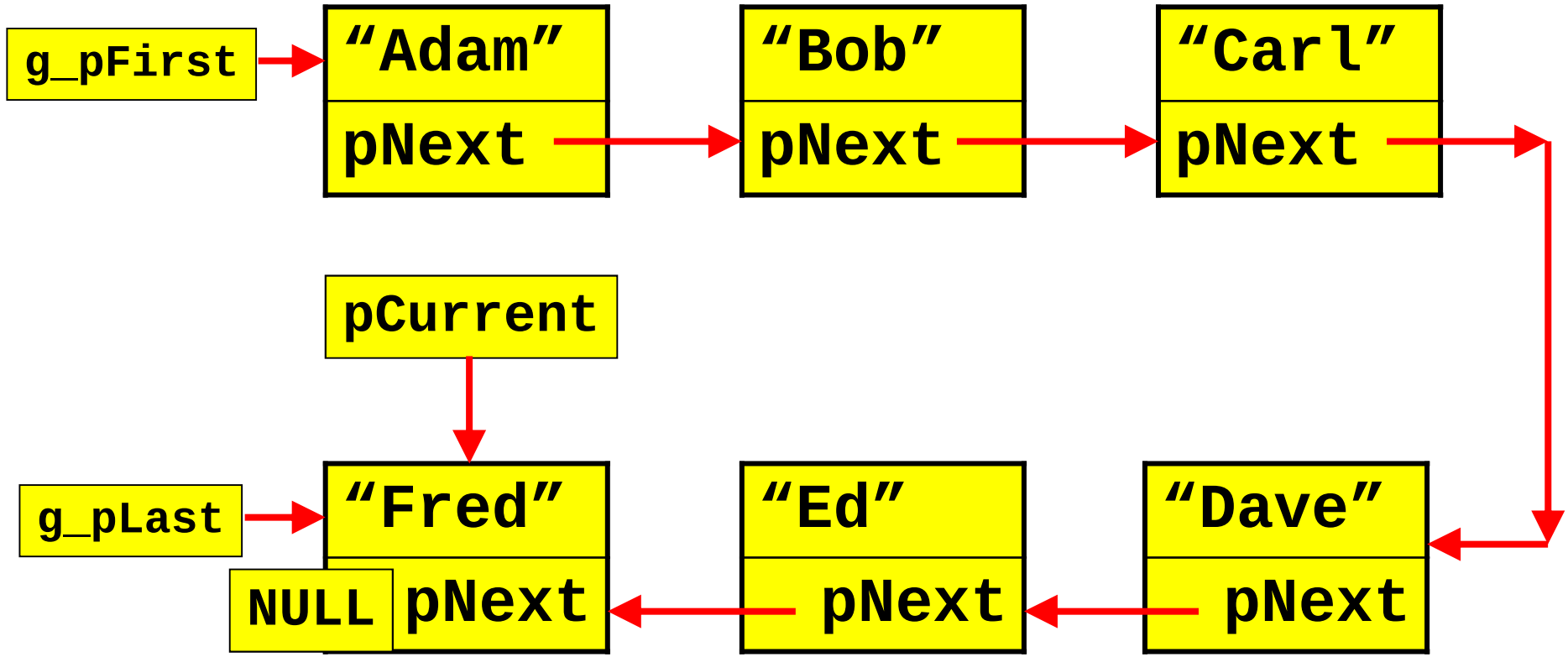

```
if ( pCurrent == NULL ) return;  // STOP!
printf( “%s\n”, pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Print all items



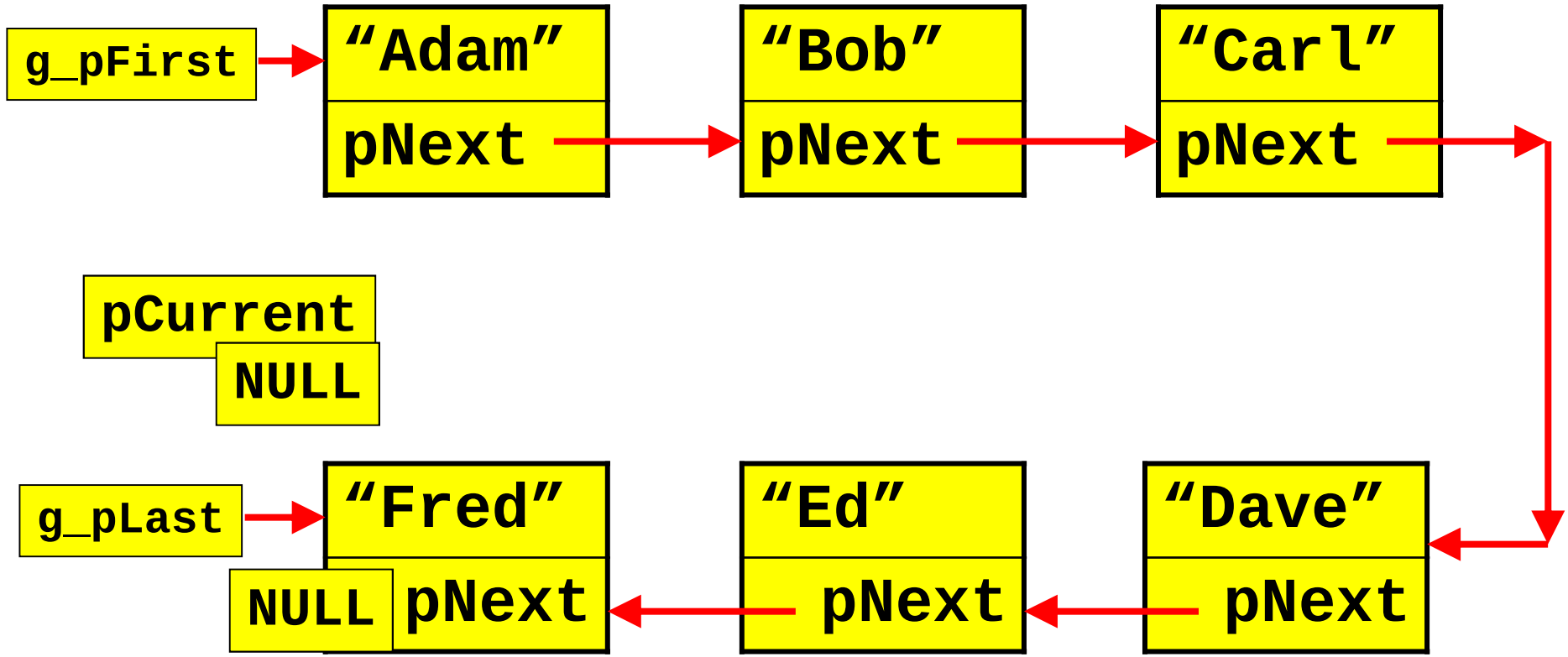

```
if ( pCurrent == NULL ) return;  // STOP!
printf( “%s\n”, pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Print all items




```
if ( pCurrent == NULL ) return;  // STOP!
printf( “%s\n”, pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Print all items



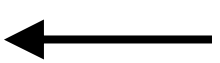

```
if ( pCurrent == NULL ) return;  // STOP!
printf( “%s\n”, pCurrent->pData );
pCurrent = pCurrent->pNext;
```

Sample code: listing items

```
SLLEntry* g_pFirst;
```

```
struct SLLEntry
{
  SLLEntry* pNext;
  char* pData;
};
```

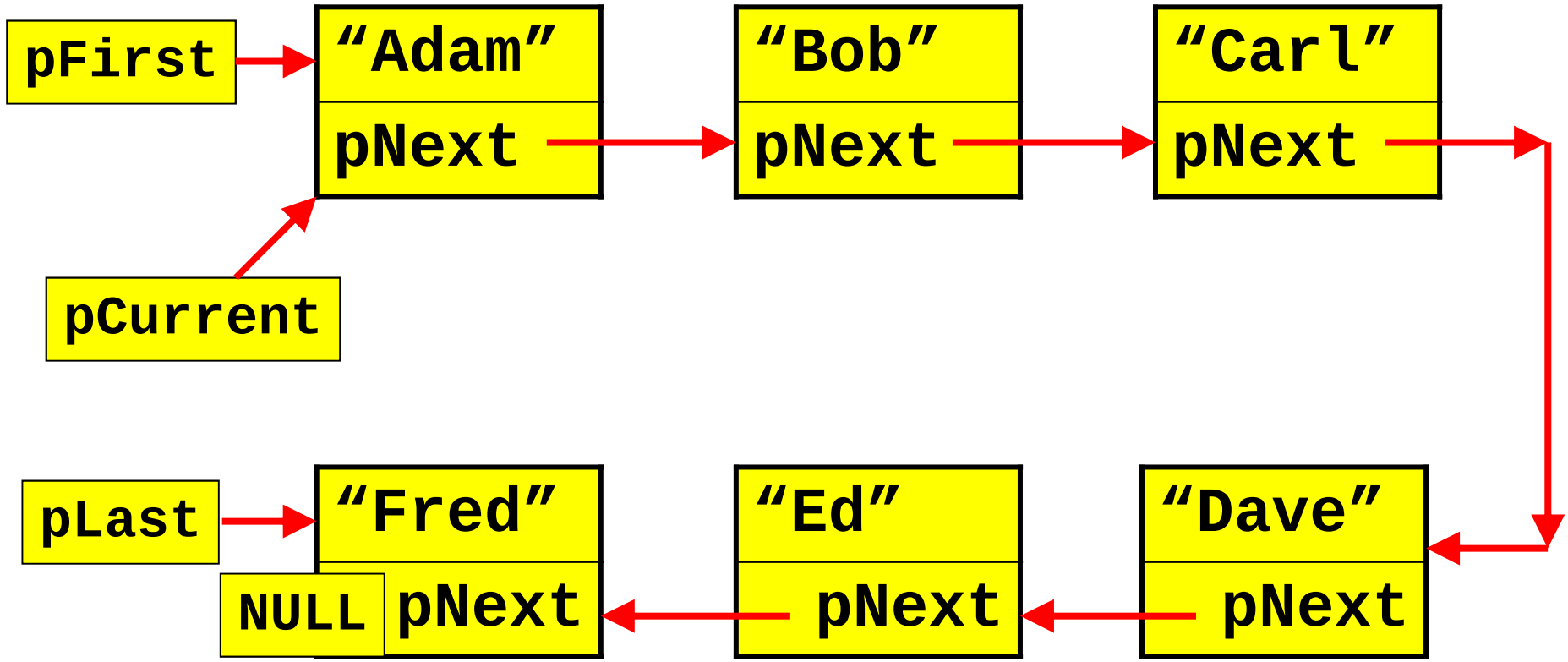
```
void DebugPrintListEntries()
{
  SLLEntry* pCurrent = g_pFirst;
  while ( pCurrent != NULL )
  {
     PrintItem( "\t\t", pCurrent );
     pCurrent = pCurrent->pNext;
  }
}
```



i.e. stop when pCurrent == NULL

Finding an item

Finding an item



Sample code: finding an item

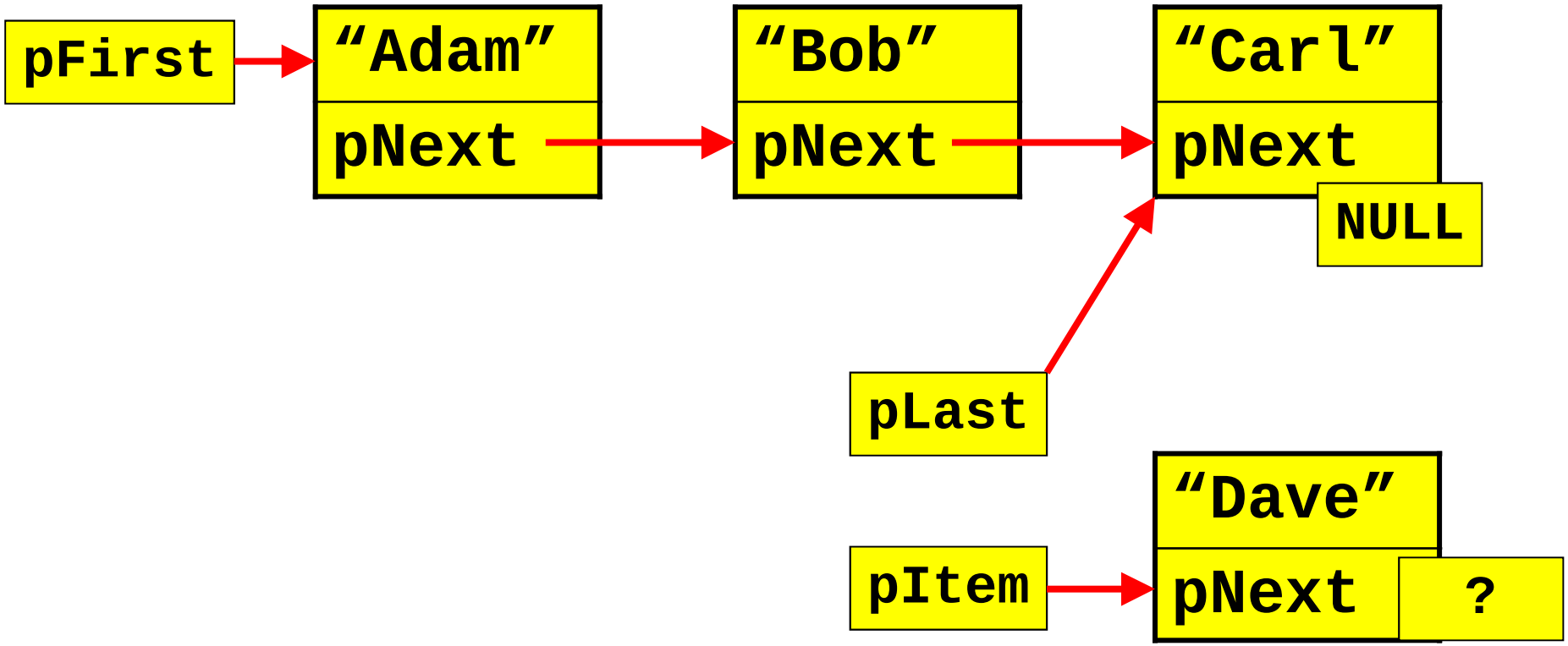
```
SLLEntry* g_pFirst;

char* pSeek = "????";
```

```
struct SLLEntry
{
  SLLEntry* pNext;
  char* pData;
};
```

```
SLLEntry* pCurrent = g_pFirst;

while ( pCurrent != NULL )
{
  if ( strcmp( pCurrent->pData, pSeek ) == 0 )
     return pCurrent;
  pCurrent=pCurrent->pNext;
}

return NULL; /* Not found */
```

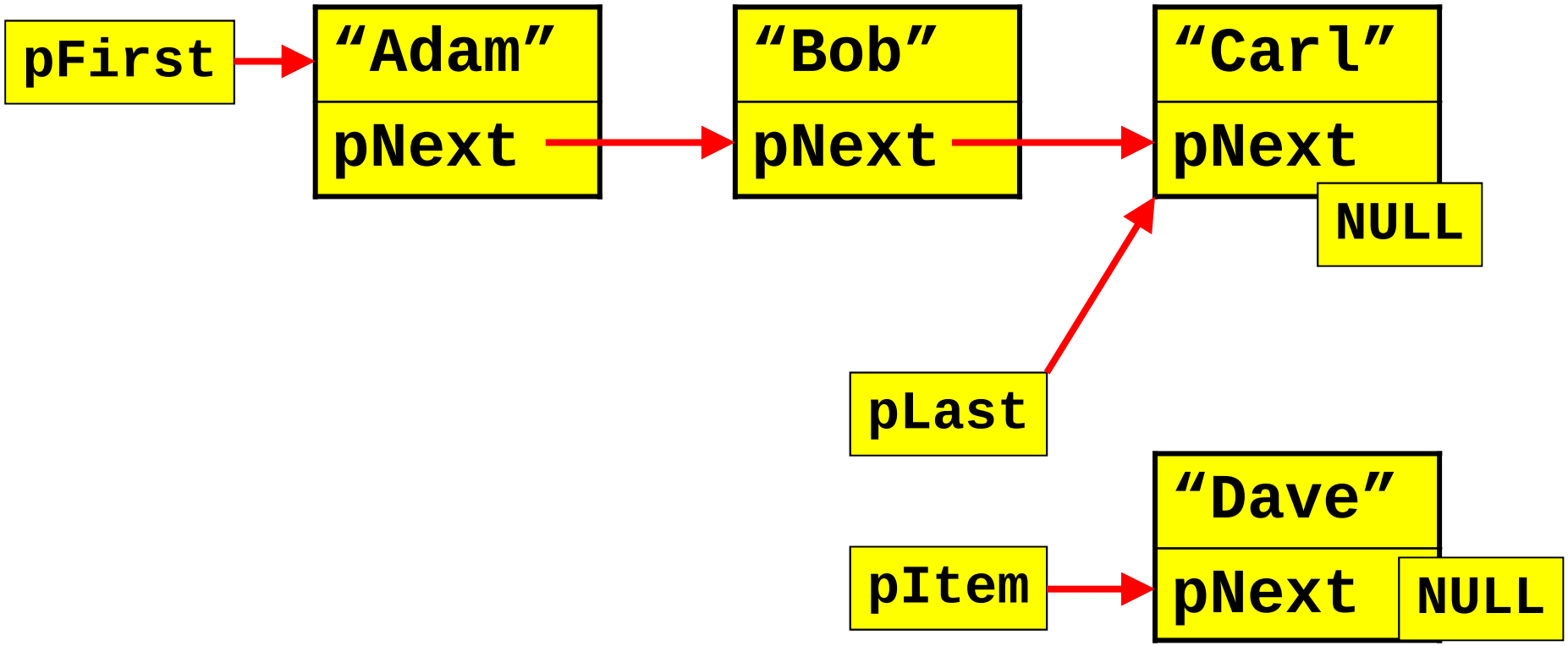
Adding items to the end of a linked list

Single-linked list : Addition (1)



1) Create the new item, and a pointer to point to it (**`pItem`**)

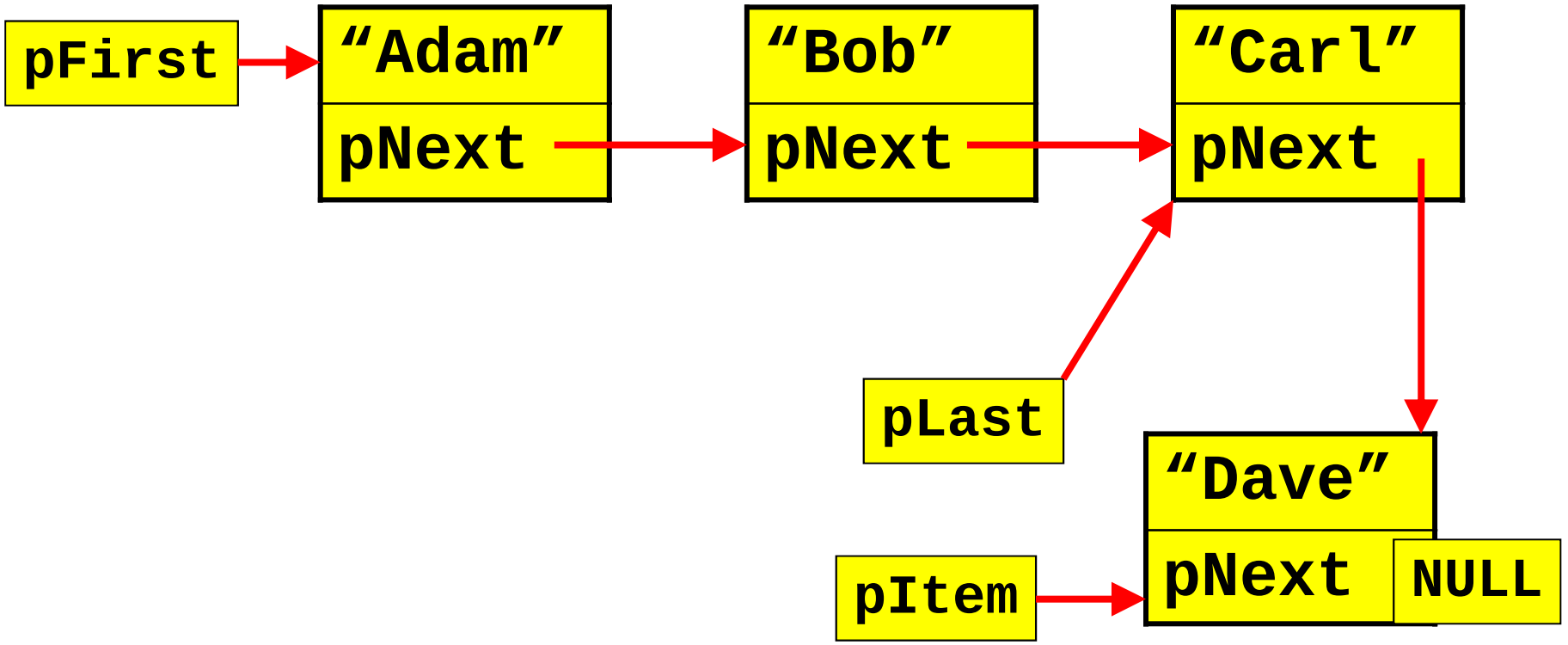
Single-linked list : Addition (2)



2) Set **pNext** on new item to NULL

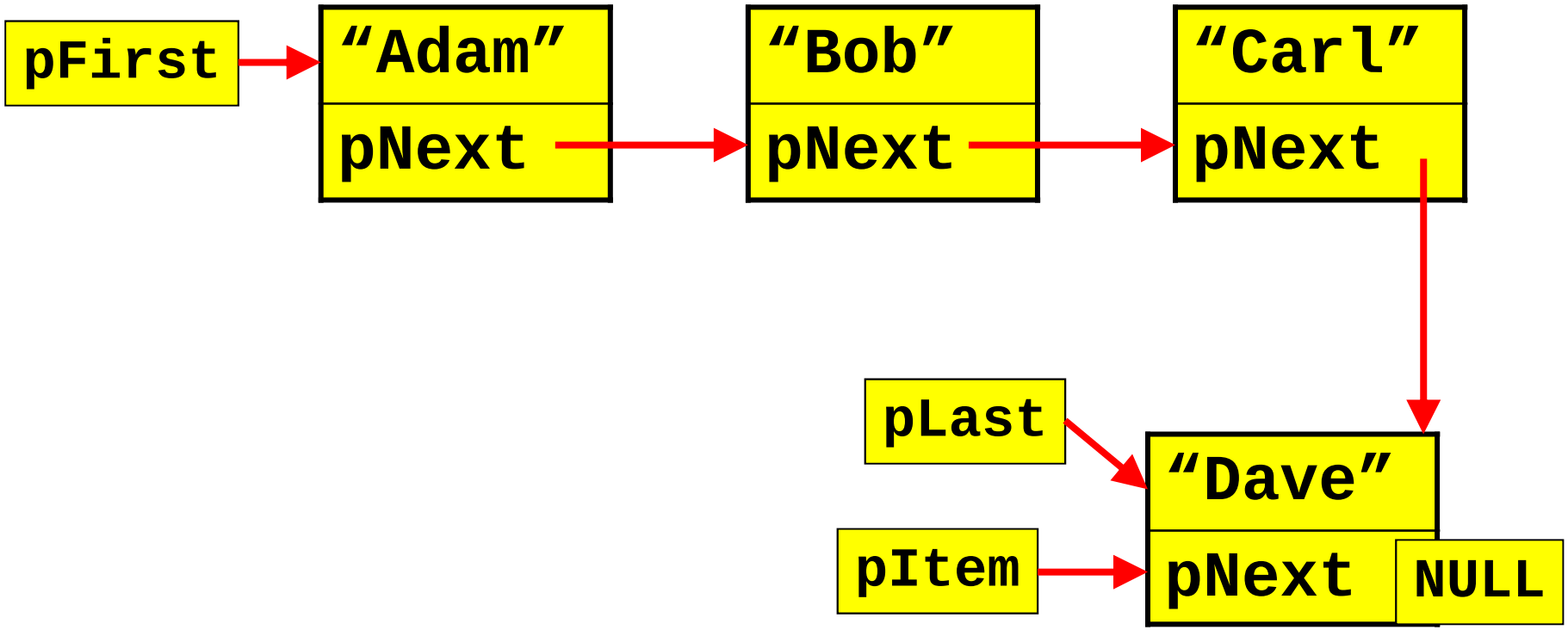
```
pItem->pNext = NULL;
```

Single-linked list : Addition (3)



3) Make **pNext** of previous **last** item point to this new item
i.e. set **pNext** of the item that **pLast** points at to point
at new item: **pLast->pNext = pItem;**

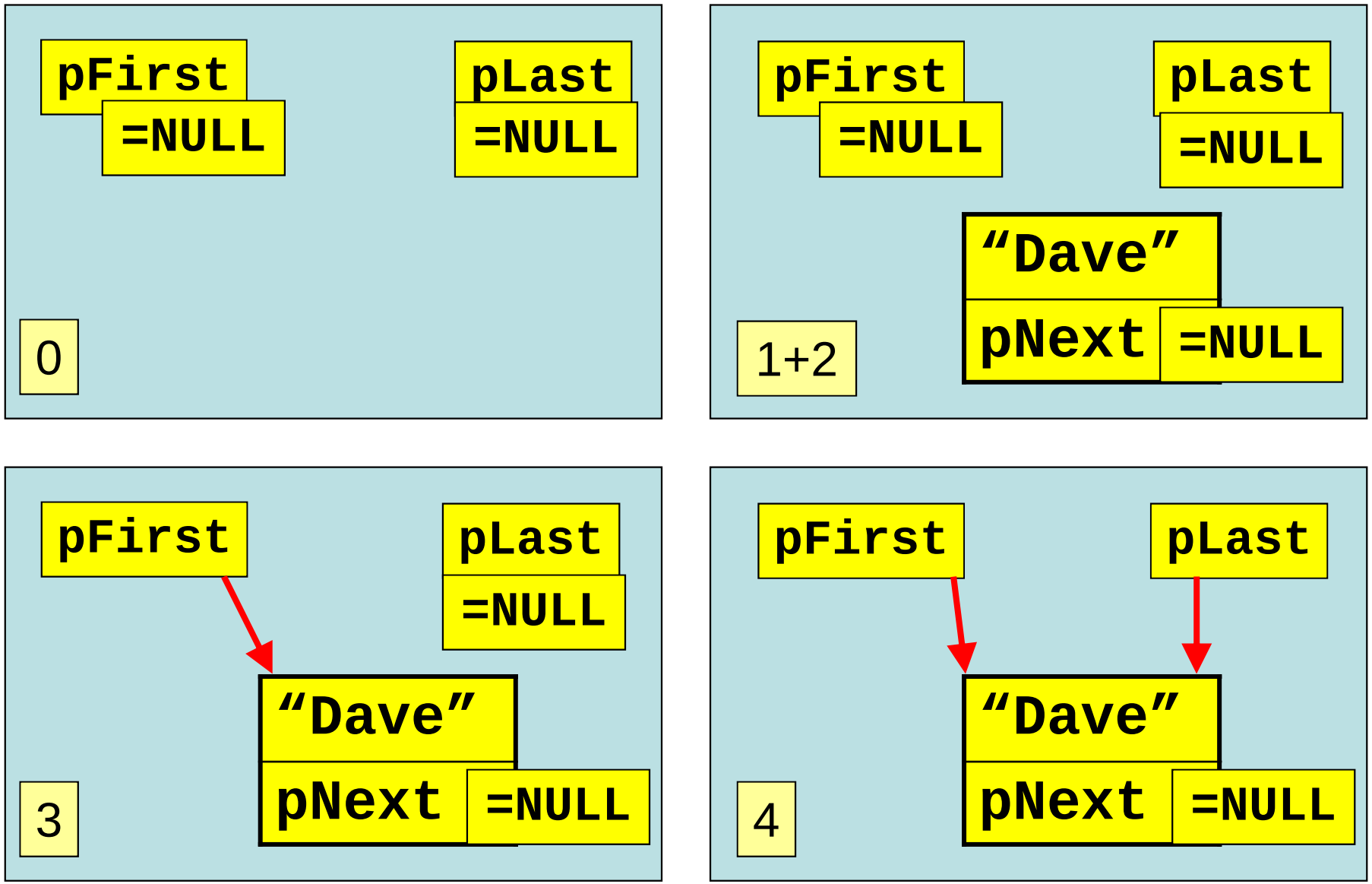
Single-linked list : Addition (4)



4) Update the value of **pLast**

```
pLast = pItem;
```

Special case: no list entries



Sample code : create new item

```
static void AddNewItemToEndOfList( char* pData )
{
  /* Create a new item to add */
  SLLEntry * pNewEntry =
            (SLLEntry *)malloc( sizeof(SLLEntry) );

  /* Create storage for string, copy string into it */
  pNewEntry->pData = (char*)malloc( strlen(pData)+1 );
  strcpy( pNewEntry->pData, pData );

  /* This will always be last entry in list
  i.e. no next entry so next pointer is NULL */
  pNewEntry->pNext = NULL;
```

Continued on next slide…

```
struct SLLEntry
{
  SLLEntry* pNext;
  char* pData;
};
```

Sample code : link into list

```
    /* Now link the item into the list */

    /* Special case for first item: */
    if ( g_pFirst == NULL )
    {
        g_pFirst = pNewEntry; /* This is first entry */
        g_pLast = pNewEntry; /* And last entry */
    }
    else /* Otherwise a list exists so insert after it*/
    {
        /* Link new item after the old last one */
        g_pLast->pNext = pNewEntry;
        /* Record the new item as the new last one */
        g_pLast = pNewEntry;
    }
}
```

Linked list vs Array

- Array:
  - Fast to find an entry at a specific index
  - Easier to visualise?
  - Problems on insert/delete?
  - Problems on resize?
- Linked list:
  - Quicker to insert? – no copy needed
  - Potentially slower to find an entry?
  - Note: could maintain a doubly linked list
    - Can find the previous as well as the next

Next lecture

- Classes (at last)
- Inline functions