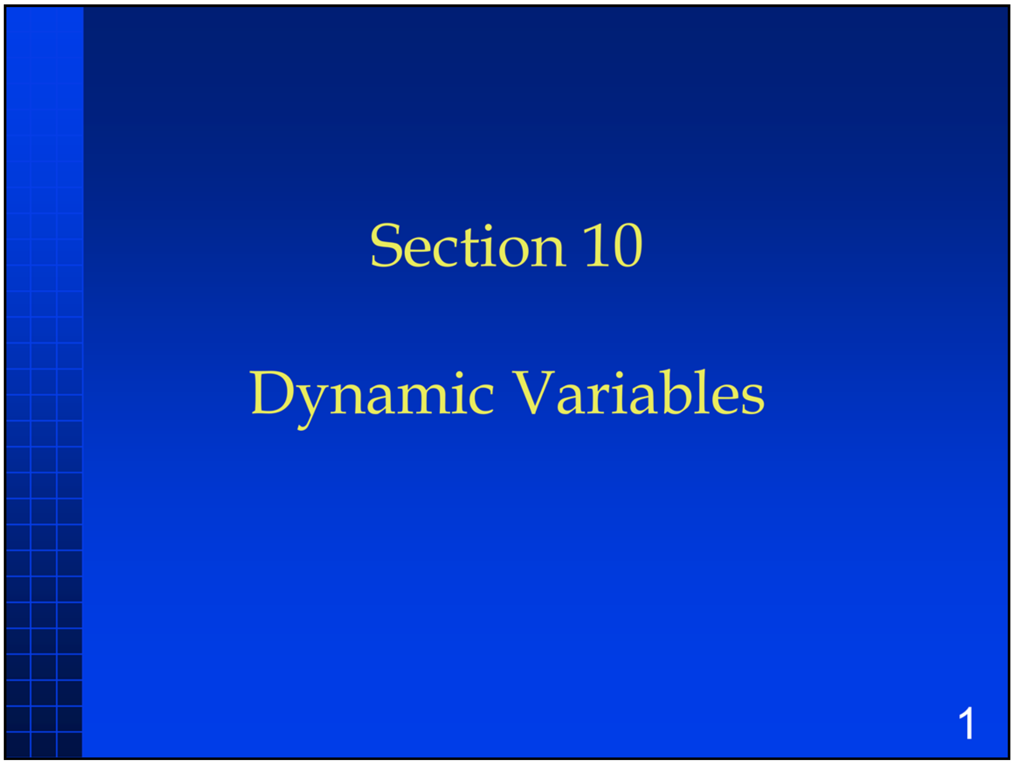
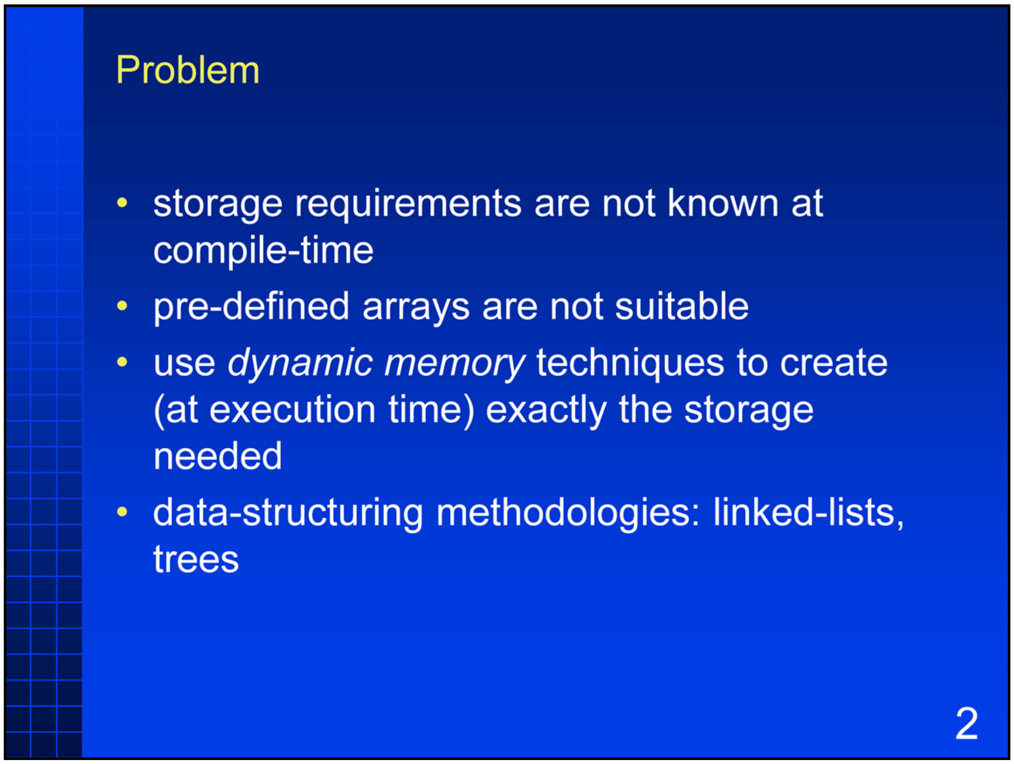


Section 10

Dynamic Variables

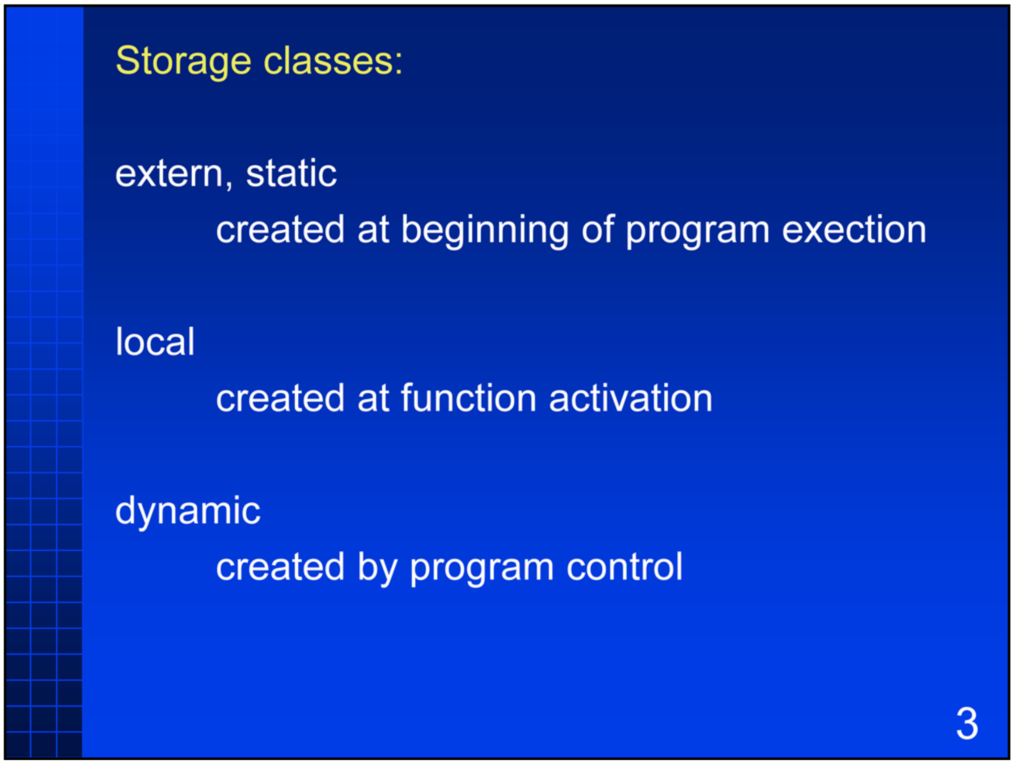
Memory Management is library based not language based



Problem

- storage requirements are not known at compile-time
- pre-defined arrays are not suitable
- use *dynamic memory* techniques to create (at execution time) exactly the storage needed
- data-structuring methodologies: linked-lists, trees

- example: storing varying amounts of data provided interactively
- arrays either waste space, or occasionally too small
- pre-defined arrays are not suitable in general
- many data-structuring techniques, add-on products



Storage classes:

extern, static

created at beginning of program exection

local

created at function activation

dynamic

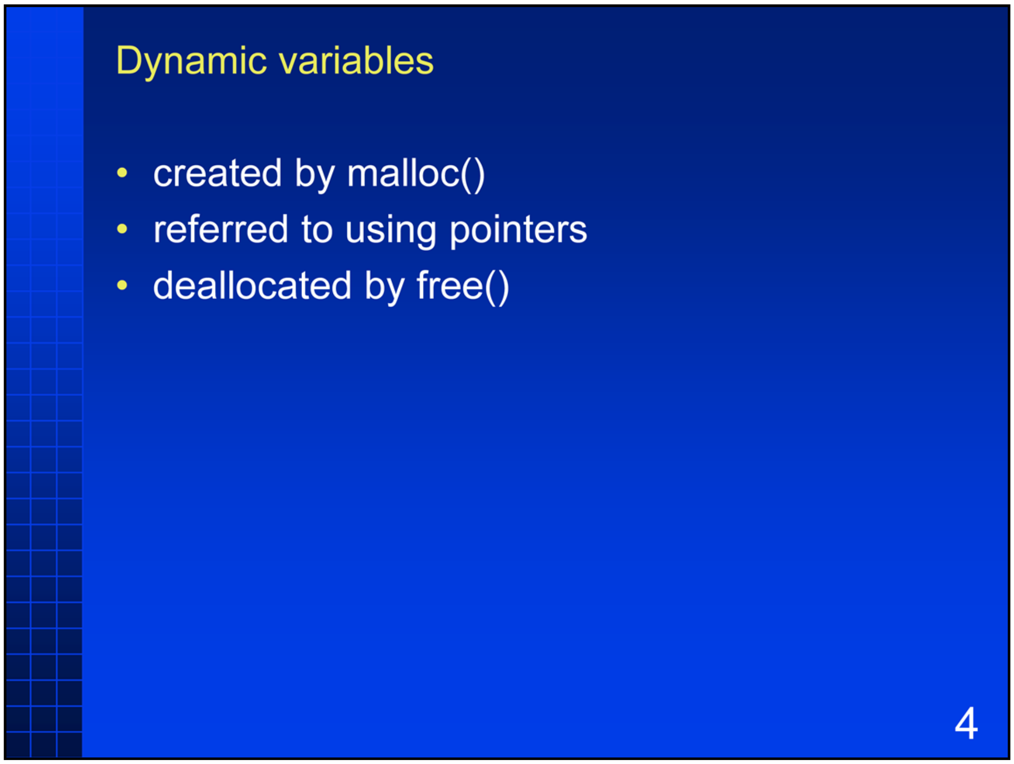
created by program control

extern -- global scope

static -- module/file scope

local -- function scope

dynamic -- known only if provided with address (not exactly scope in the traditional sense)



Dynamic variables

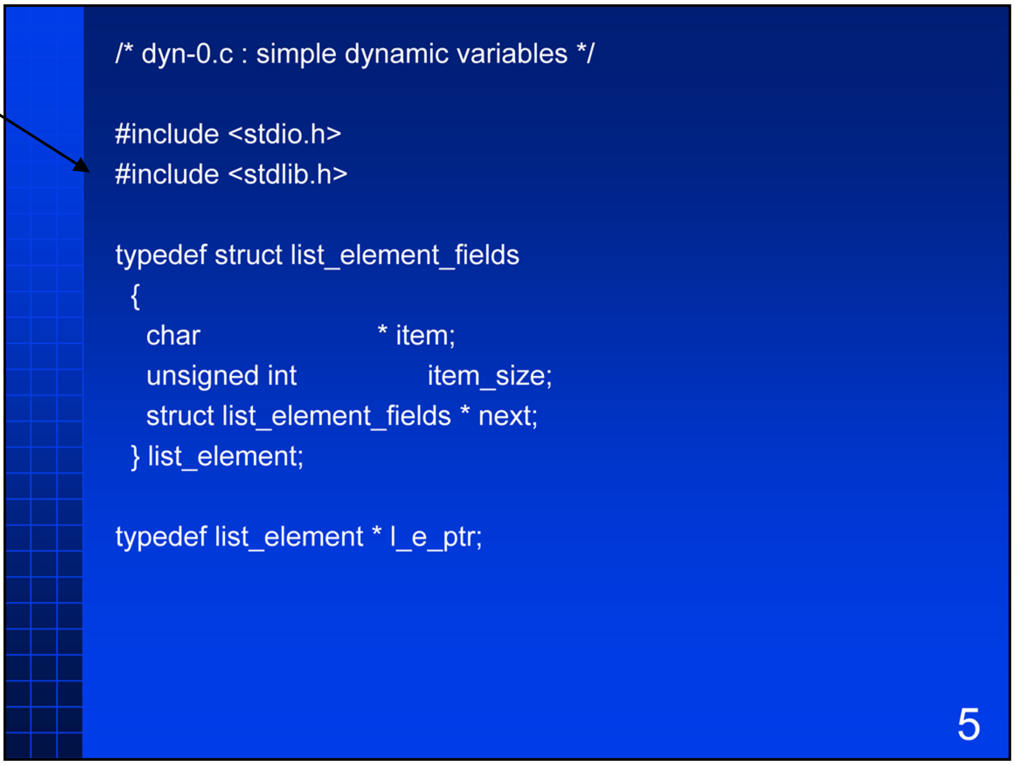
- created by malloc()
- referred to using pointers
- deallocated by free()

ANSI standard functions:

malloc - get a piece of storage from operating environment

free - give storage back

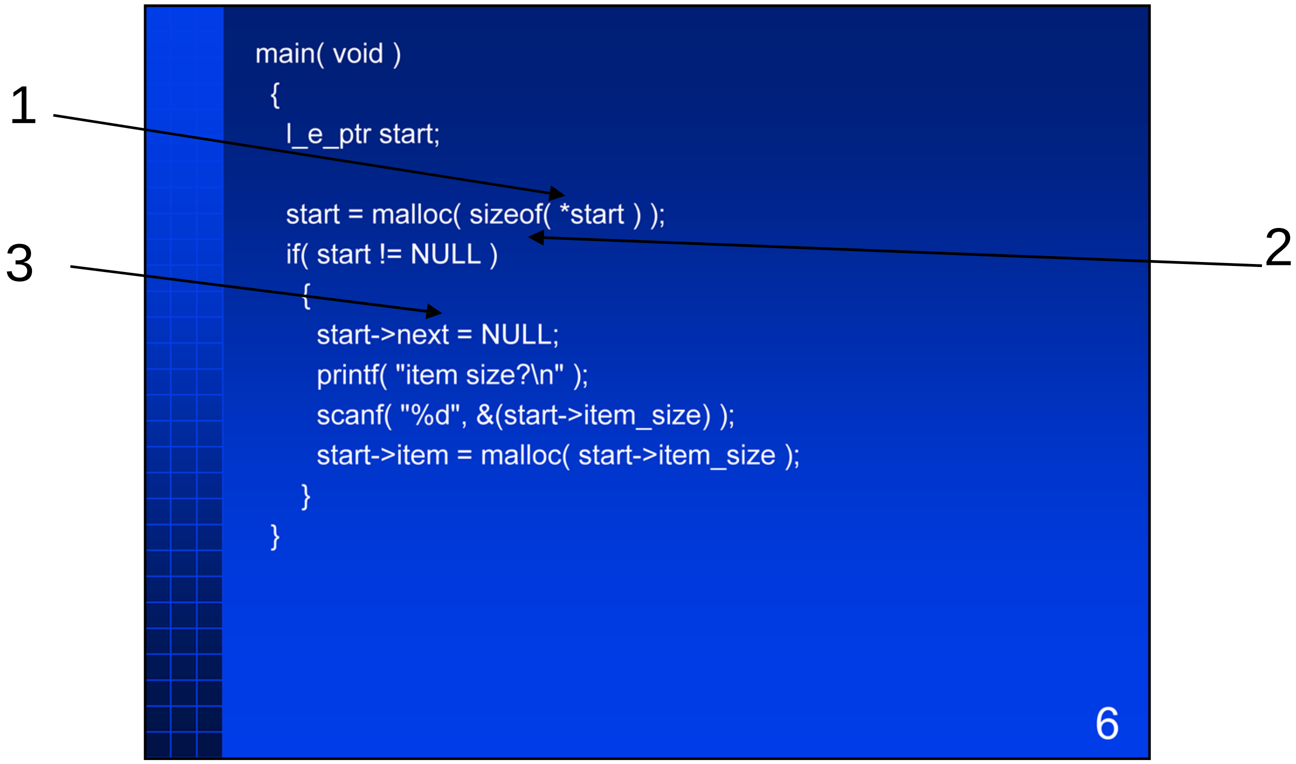
1



```
/* dyn-0.c : simple dynamic variables */

#include <stdio.h>
#include <stdlib.h>

typedef struct list_element_fields
  {
    char                        * item;
    unsigned int                  item_size;
    struct list_element_fields * next;
  } list_element;

typedef list_element * l_e_ptr;
```

1) stdlib.h -- standard file containing memory-manipulation functions

2 define struct and pointer types



```
main( void )
 {
  l_e_ptr start;

  start = malloc( sizeof( *start ) );
  if( start != NULL )
   {
    start->next = NULL;
    printf( "item size?\n" );
    scanf( "%d", &(start->item_size) );
    start->item = malloc( start->item_size );
   }
 }
```

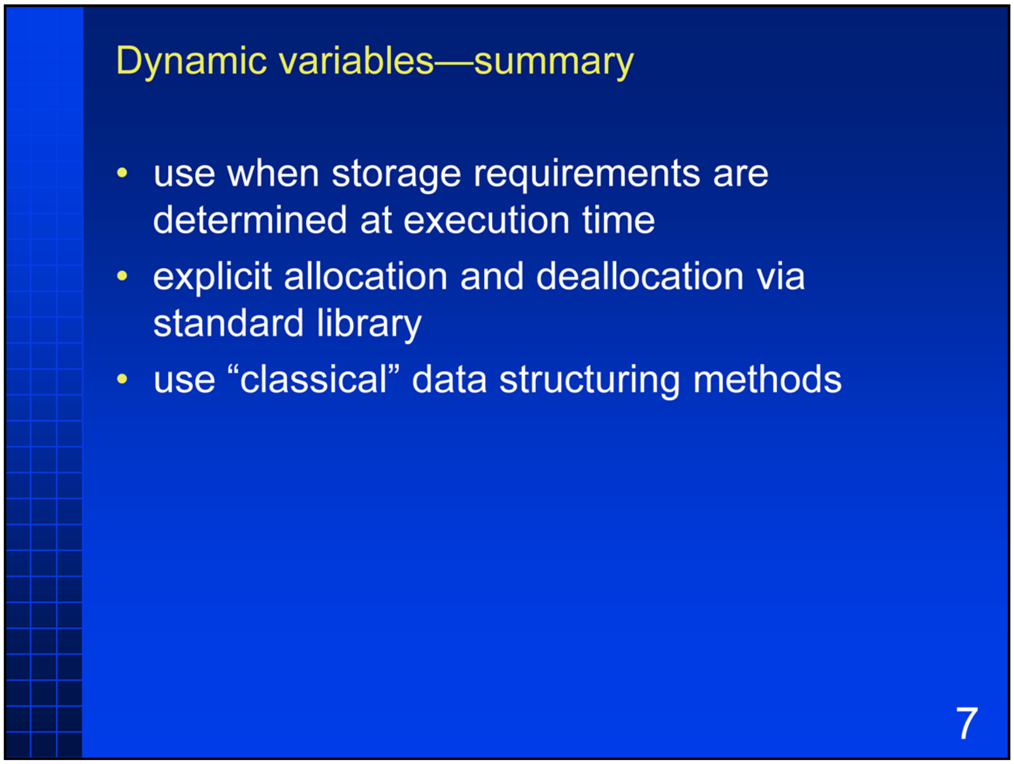
1) sizeof == compile-time; “get storage big enough to hold thing to which start points), not sizeof(start), which is size of pointer

2) malloc returns null if not available

3) notation : dereference, then field select == (*start).next

NULL: sort of zero, actually “a pointer value toat does’t point at anything”

[chalkboard walkthough]



Dynamic variables—summary

- use when storage requirements are determined at execution time
- explicit allocation and deallocation via standard library
- use “classical” data structuring methods

1) library not language

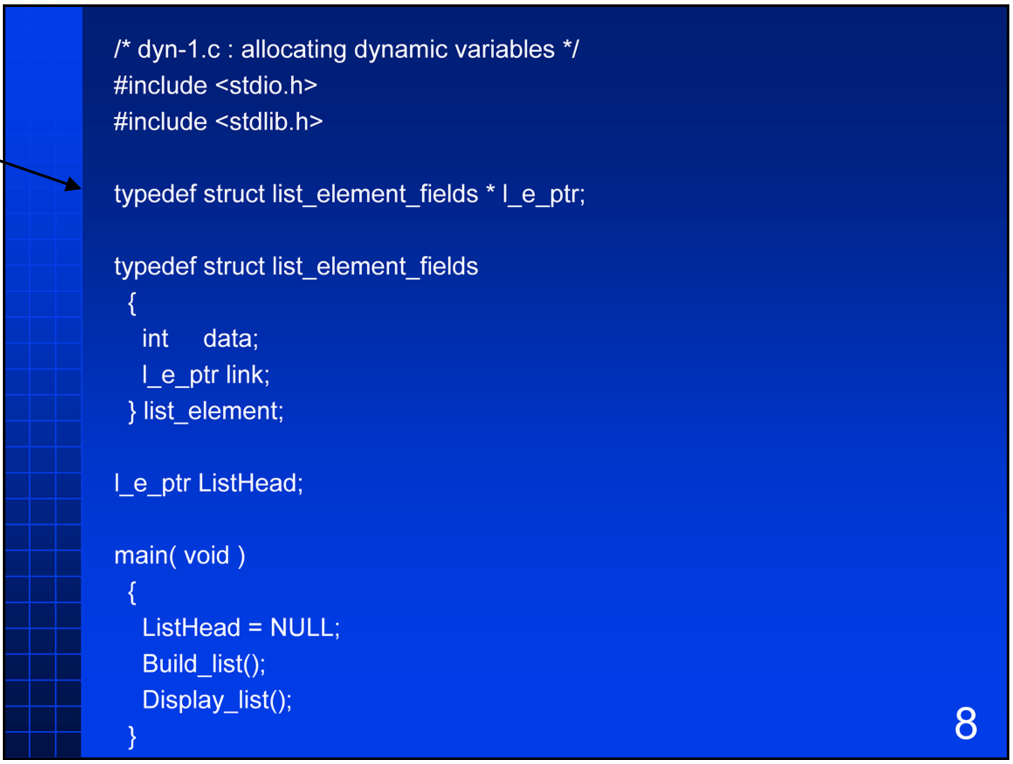
2) standard defines names and basic semmantics; implementations are free to implement as suits [sys call or not, efficiency, garbage collect etc.]

3) almost always layered for real programs

4) there exists exception handling libraries for extraordinary situations

Code reusability becomes important for data-manipulation etc. Need well-designed libraries

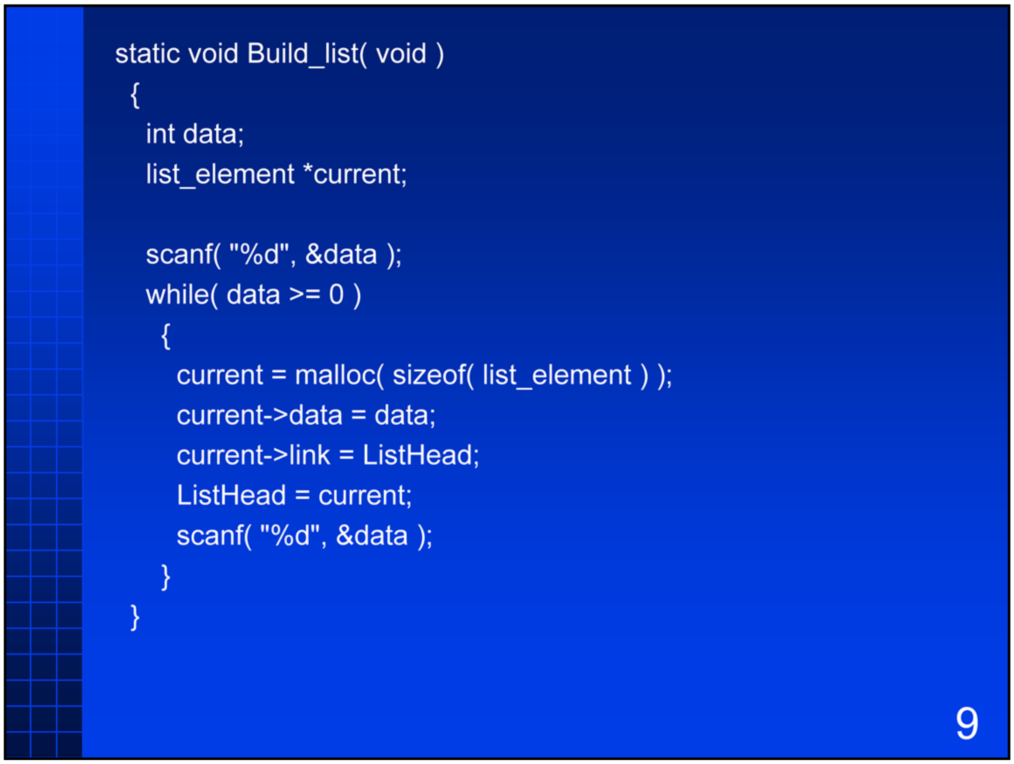
1



```
/* dyn-1.c : allocating dynamic variables */
#include <stdio.h>
#include <stdlib.h>

typedef struct list_element_fields * l_e_ptr;

typedef struct list_element_fields
 {
   int     data;
   l_e_ptr link;
 } list_element;

l_e_ptr ListHead;

main( void )
 {
   ListHead = NULL;
   Build_list();
   Display_list();
 }
```

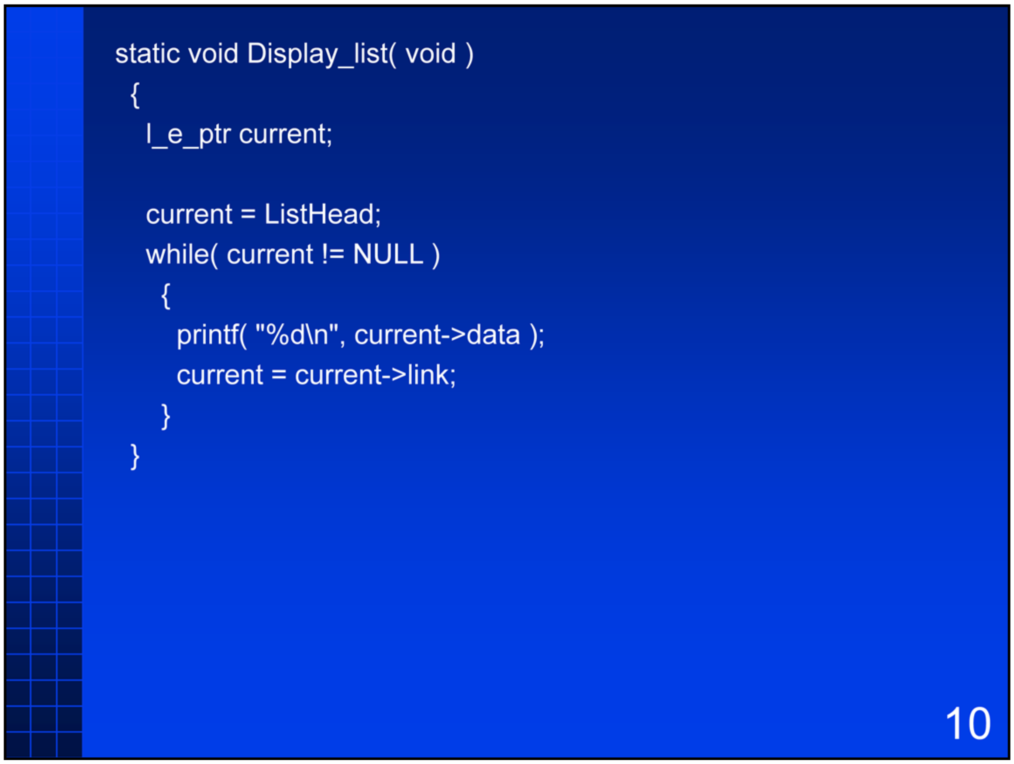
Example, build and display a singly-linked list

1) unresolved forward pointer declaration



```
static void Build_list( void )
 {
   int data;
   list_element *current;

   scanf( "%d", &data );
   while( data >= 0 )
    {
      current = malloc( sizeof( list_element ) );
      current->data = data;
      current->link = ListHead;
      ListHead = current;
      scanf( "%d", &data );
    }
 }
```

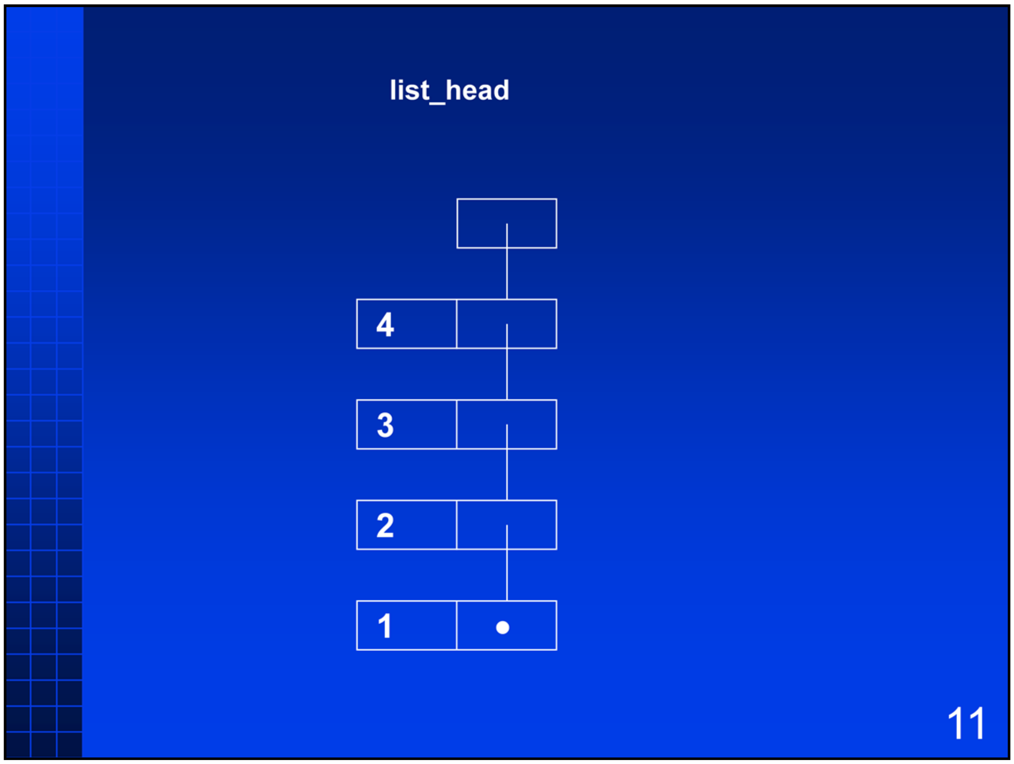
builds the list backwards



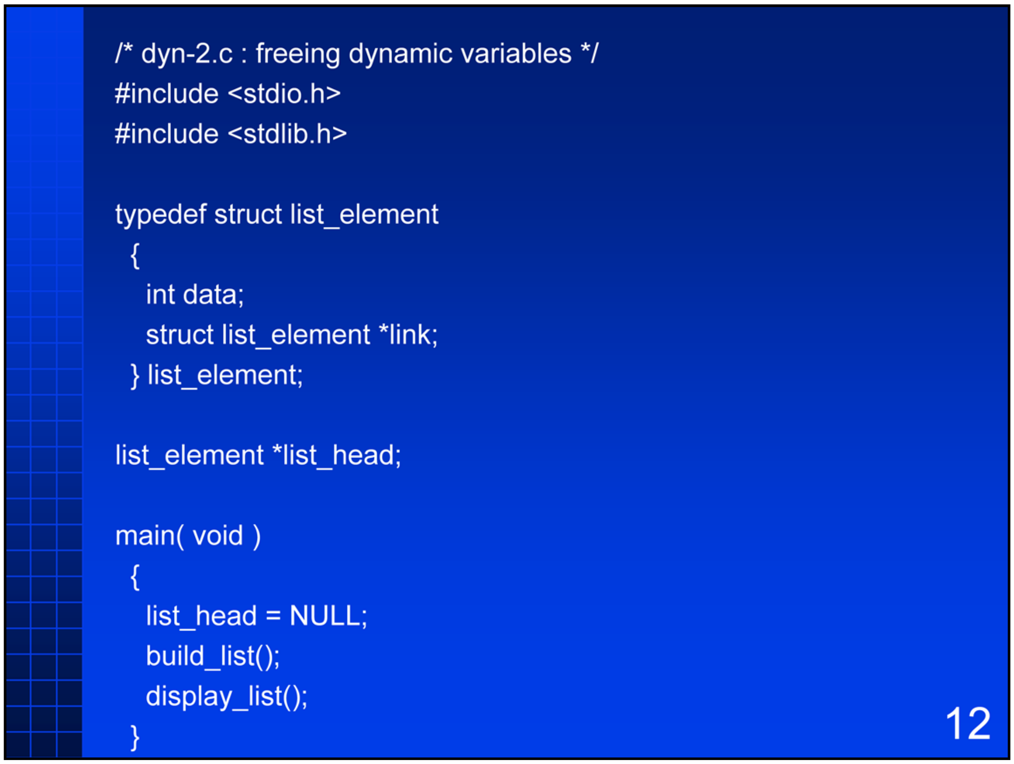
```
static void Display_list( void )
 {
   l_e_ptr current;

   current = ListHead;
   while( current != NULL )
    {
      printf( "%d\n", current->data );
      current = current->link;
    }
 }
```

displays the list. to free as you go:

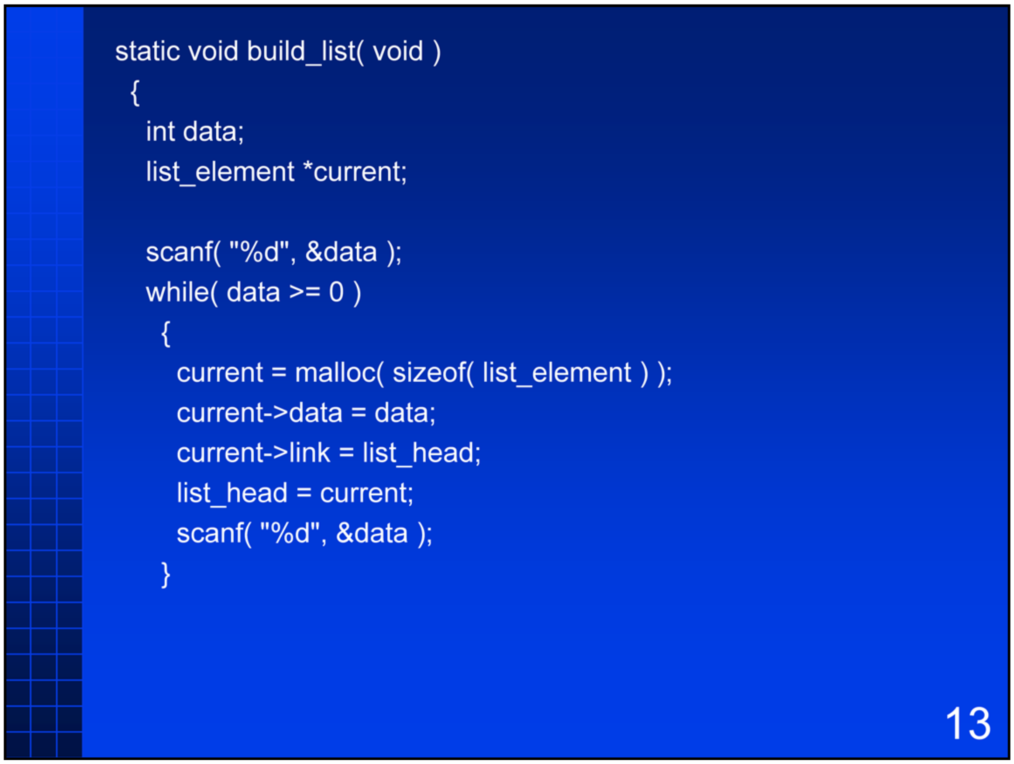
```
static void display_list( void )
 {
   list_element *current, *next;

   current = list_head;
   while( current != NULL )
    {
      printf( "%d\n", current->data );
      next = current->link;
      free( current );
      current = next;
    }
 }
```



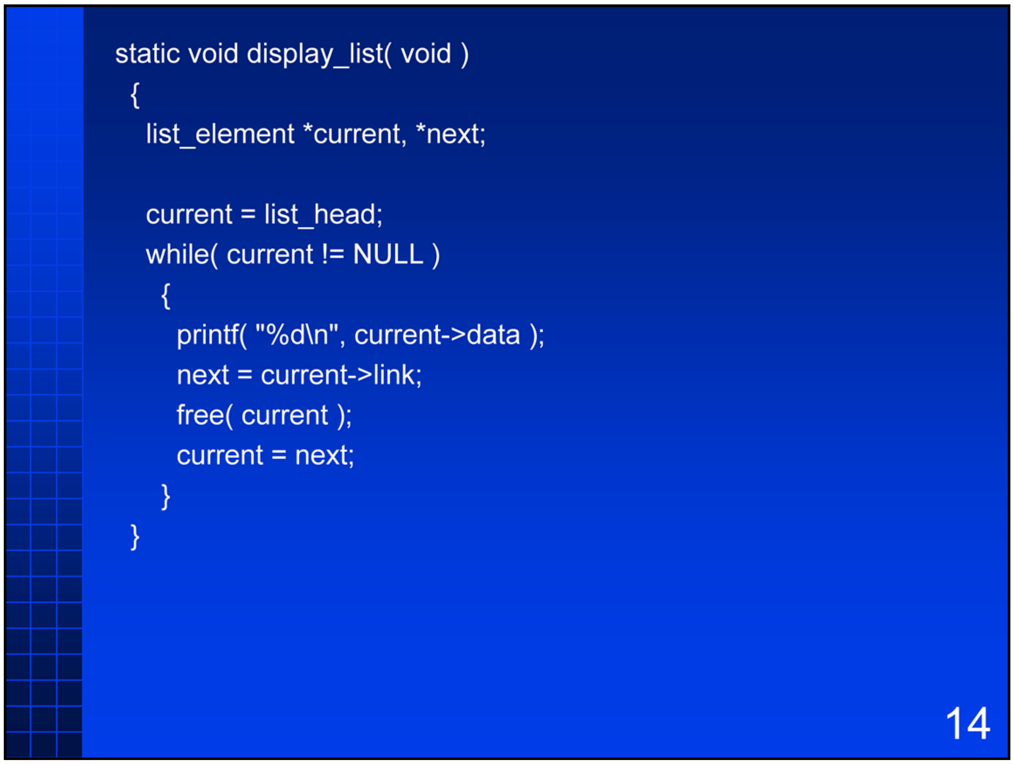
input 1 2 3 4 -1



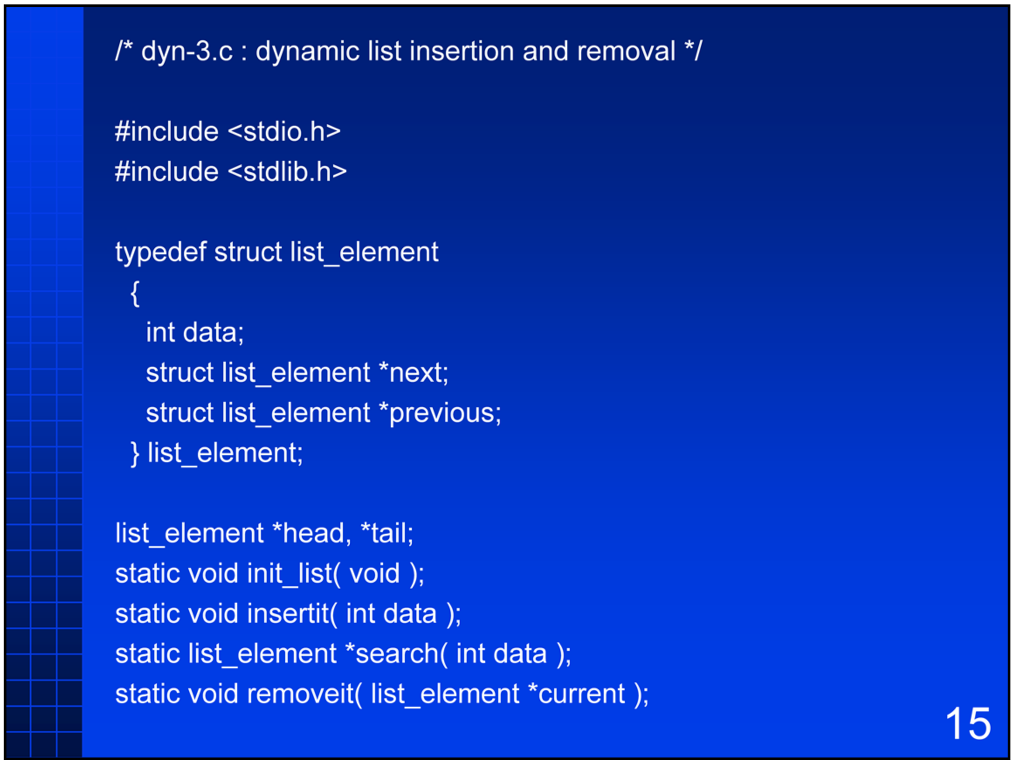
```
/* dyn-2.c : freeing dynamic variables */
#include <stdio.h>
#include <stdlib.h>

typedef struct list_element
  {
    int data;
    struct list_element *link;
  } list_element;

list_element *list_head;

main( void )
  {
    list_head = NULL;
    build_list();
    display_list();
  }
```



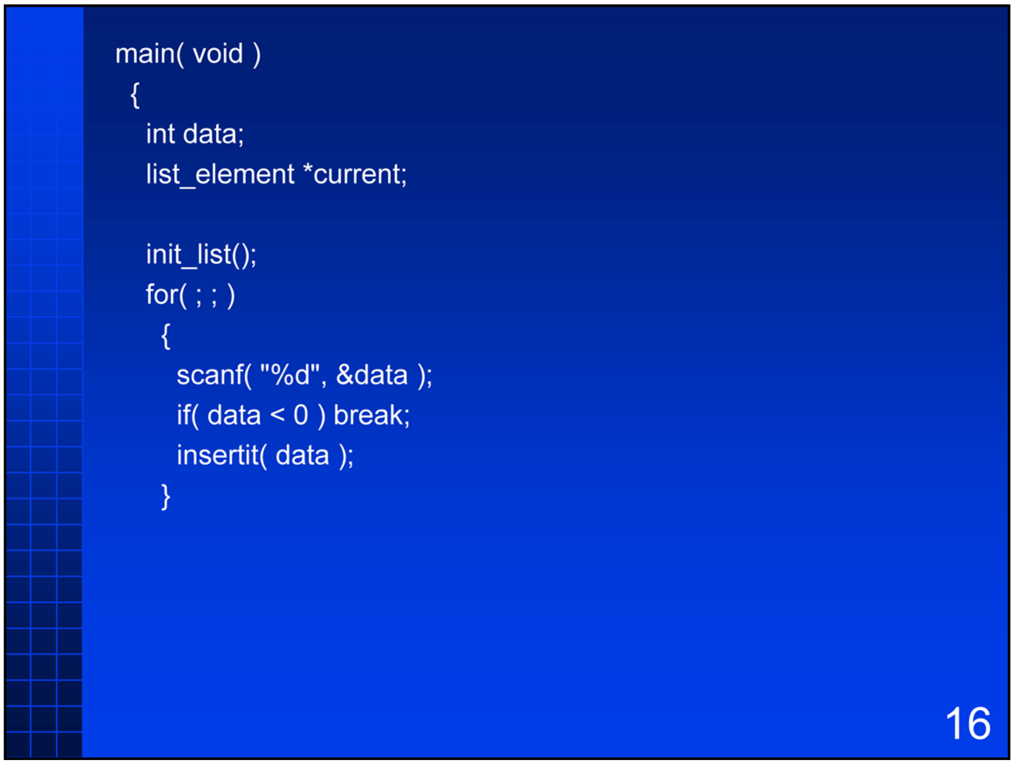
```
static void build_list( void )
 {
  int data;
  list_element *current;

  scanf( "%d", &data );
  while( data >= 0 )
   {
    current = malloc( sizeof( list_element ) );
    current->data = data;
    current->link = list_head;
    list_head = current;
    scanf( "%d", &data );
   }
```



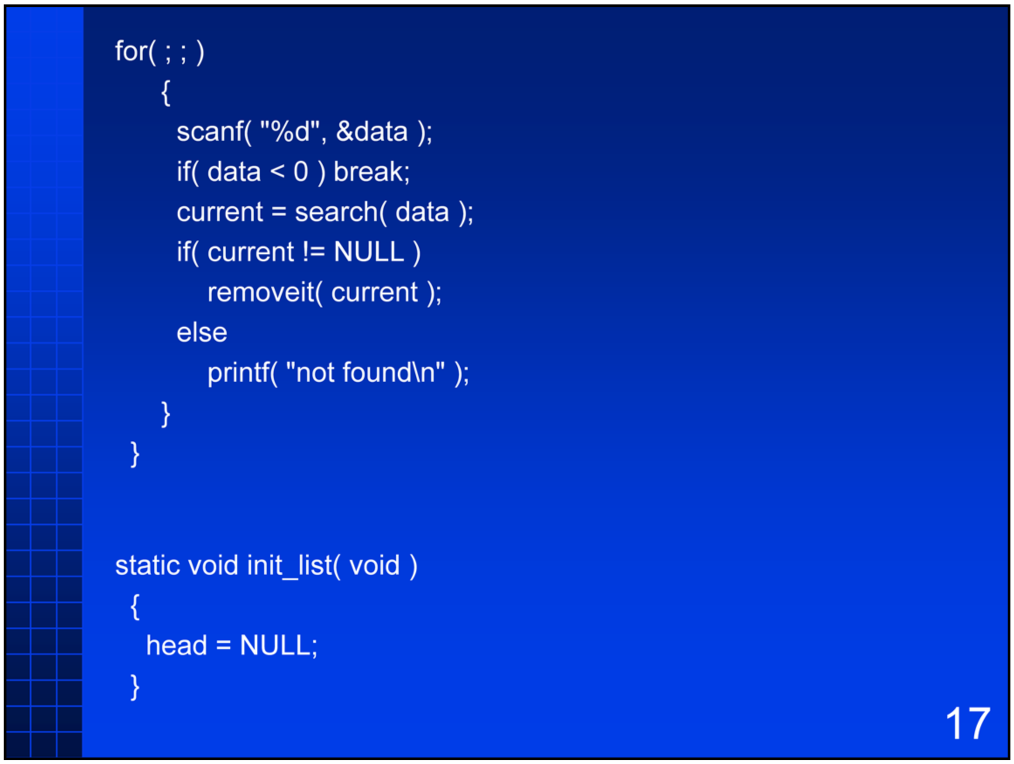
```
static void display_list( void )
 {
  list_element *current, *next;

  current = list_head;
  while( current != NULL )
   {
    printf( "%d\n", current->data );
    next = current->link;
    free( current );
    current = next;
   }
 }
```



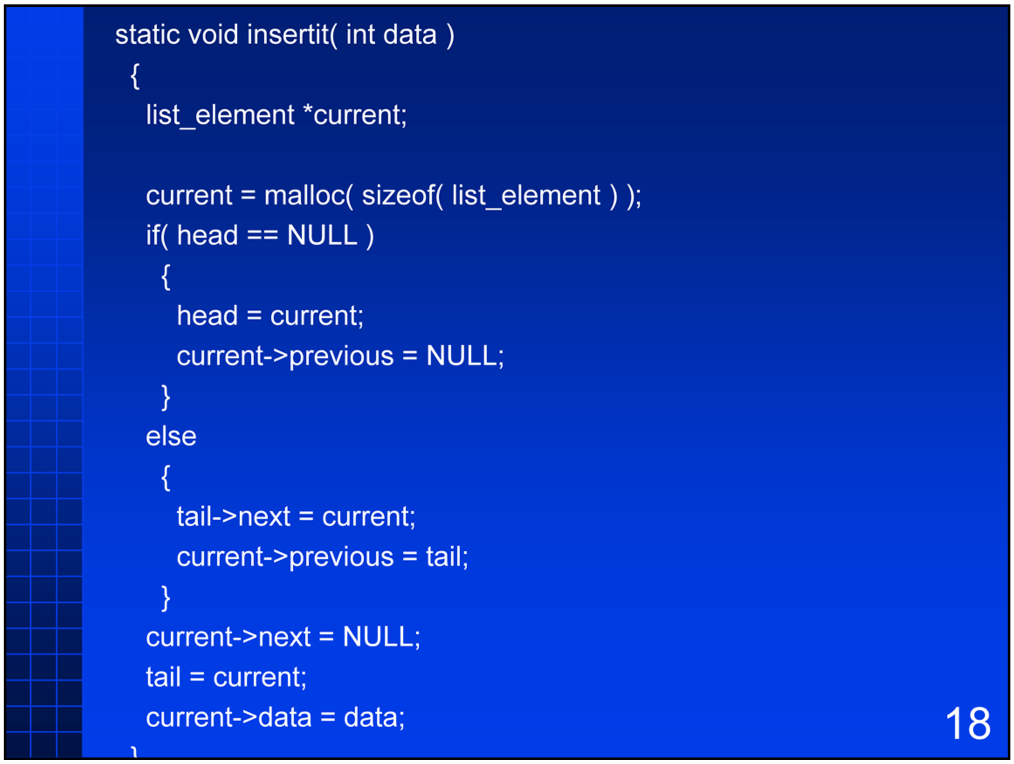
```
/* dyn-3.c : dynamic list insertion and removal */

#include <stdio.h>
#include <stdlib.h>

typedef struct list_element
  {
    int data;
    struct list_element *next;
    struct list_element *previous;
  } list_element;

list_element *head, *tail;
static void init_list( void );
static void insertit( int data );
static list_element *search( int data );
static void removeit( list_element *current );
```



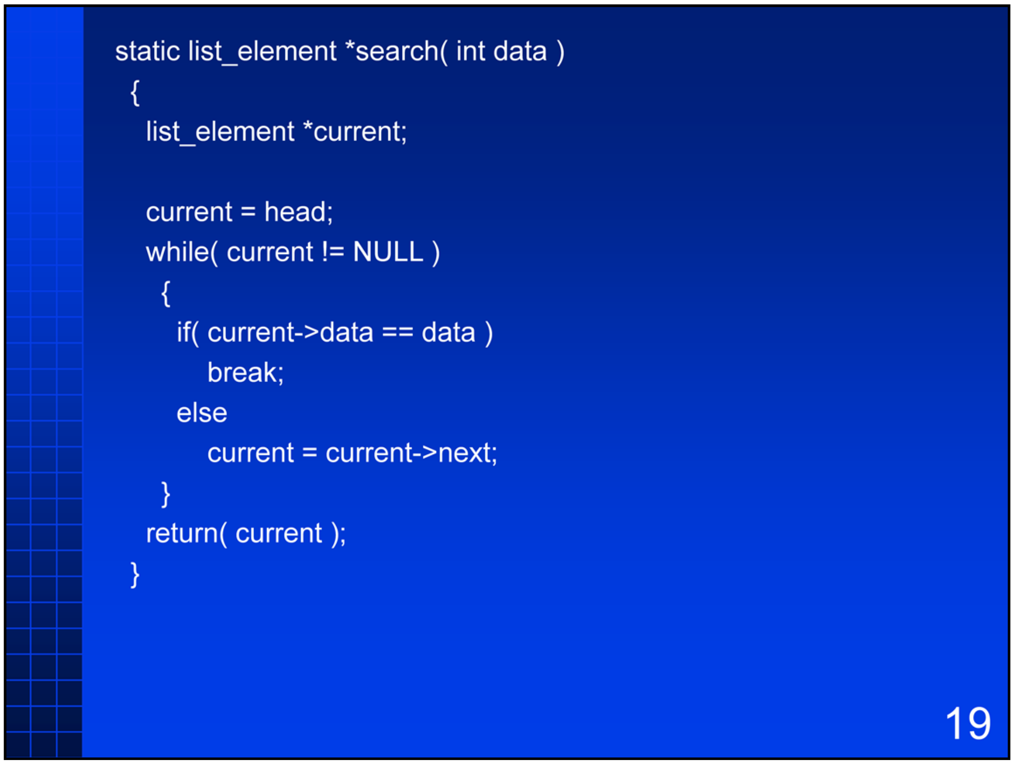
```
main( void )
 {
   int data;
   list_element *current;

   init_list();
   for( ; ; )
    {
      scanf( "%d", &data );
      if( data < 0 ) break;
      insertit( data );
    }
```



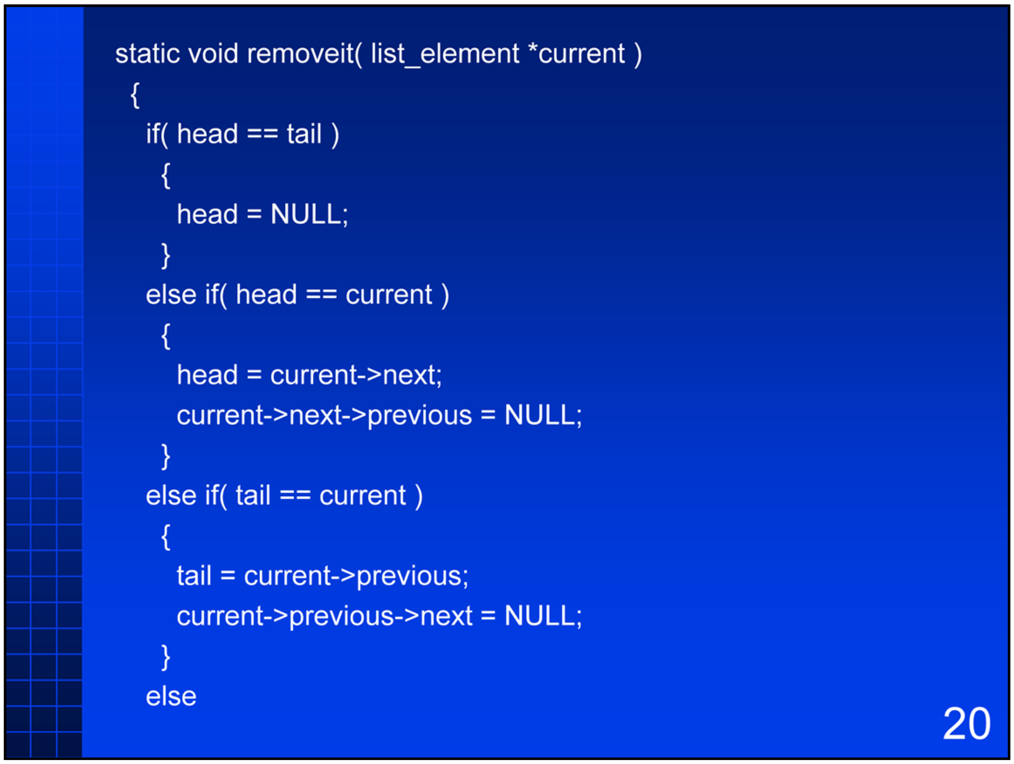
```
for( ; ; )
    {
     scanf( "%d", &data );
     if( data < 0 ) break;
     current = search( data );
     if( current != NULL )
        removeit( current );
     else
        printf( "not found\n" );
    }
 }


static void init_list( void )
 {
   head = NULL;
 }
```



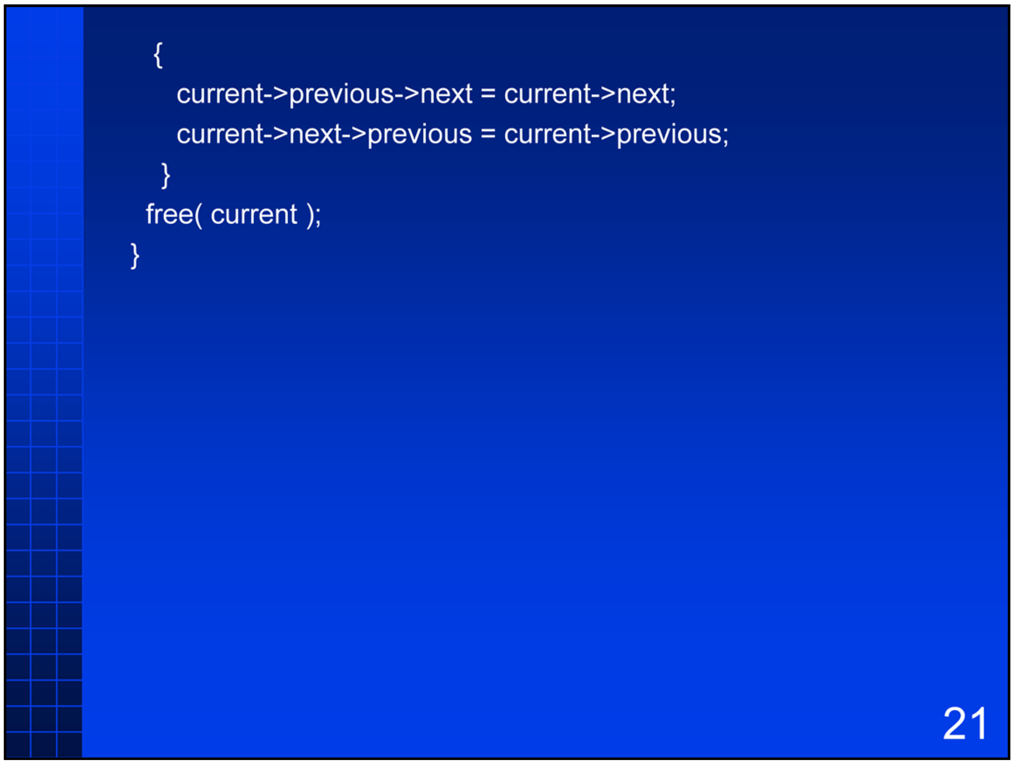
```
static void insertit( int data )
 {
   list_element *current;

   current = malloc( sizeof( list_element ) );
   if( head == NULL )
     {
       head = current;
       current->previous = NULL;
     }
   else
     {
       tail->next = current;
       current->previous = tail;
     }
   current->next = NULL;
   tail = current;
   current->data = data;
 }
```



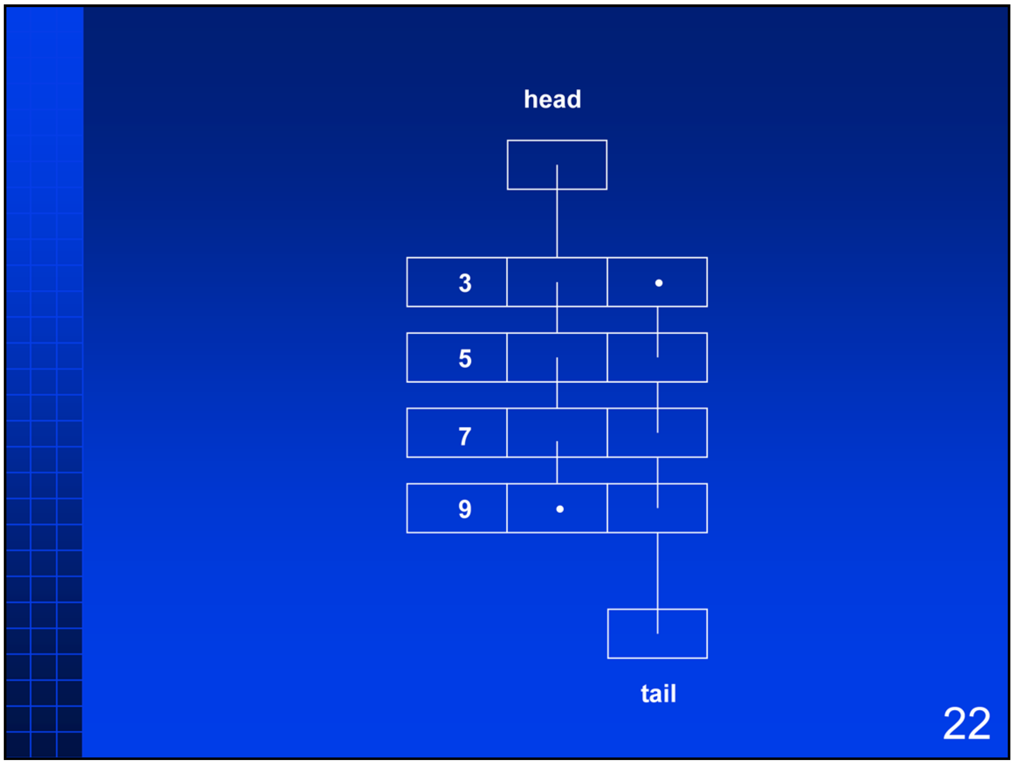
```
static list_element *search( int data )
 {
   list_element *current;

   current = head;
   while( current != NULL )
    {
      if( current->data == data )
         break;
      else
         current = current->next;
    }
   return( current );
 }
```



```
static void removeit( list_element *current )
 {
  if( head == tail )
   {
    head = NULL;
   }
  else if( head == current )
   {
    head = current->next;
    current->next->previous = NULL;
   }
  else if( tail == current )
   {
    tail = current->previous;
    current->previous->next = NULL;
   }
  else
```



```
    {
      current->previous->next = current->next;
      current->next->previous = current->previous;
     }
  free( current );
}
```


head
3
5
7
9
tail