

10001011  
01001100  
101111  
01100100  
10101101



**Systems and Software  
Verification Laboratory**



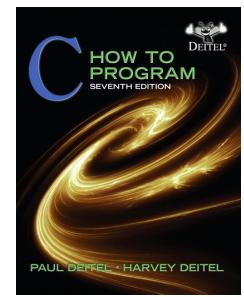
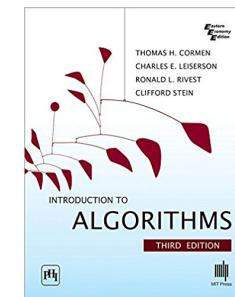
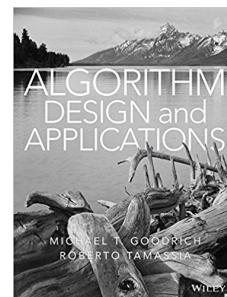
# Secure C Programming: Memory Management

**Lucas Cordeiro**  
**Department of Computer Science**  
[lucas.cordeiro@manchester.ac.uk](mailto:lucas.cordeiro@manchester.ac.uk)

# Secure C Programming

- Lucas Cordeiro (Formal Methods Group)
  - [lucas.cordeiro@manchester.ac.uk](mailto:lucas.cordeiro@manchester.ac.uk)
  - Office: 2.28
  - Office hours: 15-16 Tuesday, 14-15 Wednesday
- Textbook:
  - *Algorithm Design and Applications* (Chapter 2)
  - *Introduction to Algorithms* (Chapter 10)
  - *C How to Program* (Chapter 12)

These slides are based on the lectures notes of “C How to Program” and “SEI CERT C Coding Standard”



# Intended learning outcomes

- Understand **risk assessment** to guide **software developers**
- Review **dynamic data structures** (linked list)
- Provide rules for **secure coding** in the **C programming language**
- Develop **safe, reliable**, and **secure** systems
- Eliminate **undefined behaviours** that can lead to undefined program behaviours and **exploitable vulnerabilities**

# Intended learning outcomes

- Understand **risk assessment** to guide **software developers**
- Review **dynamic data structures** (linked list)
- Provide rules for **secure coding** in the **C programming language**
- Develop **safe, reliable, and secure systems**
- Eliminate **undefined behaviours** that can lead to undefined program behaviours and **exploitable vulnerabilities**

# Risk Assessment

- CERT C Coding Standard contains a **risk assessment** section
  - Indicate the **potential consequences** of not addressing a particular **rule** or **recommendation** in their code
- This information can be used to **prioritize the repair of rule violations** by a development team
  - The **metric** is designed primarily for **remediation** projects
- We assume that **new code** will be developed to be compliant with the entire
  - **coding standard**
  - **applicable recommendations**

# Severity

- How **serious** are the consequences of the **rule** being ignored?

Value	Meaning	Examples of Vulnerabilities
1	Low	Denial-of-service attack, abnormal termination
2	Medium	Data integrity violation, unintentional information disclosure
3	High	Run arbitrary code

# Likelihood

- How likely is it that a **flaw** introduced by ignoring the rule can lead to an **exploitable vulnerability**?

Value	Meaning
1	Unlikely
2	Probable
3	Likely

# Remediation Cost

- How **expensive** is it to comply with the **rule**?

Value	Meaning	Detection	Correction
1	High	Manual	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

# Remediation Cost

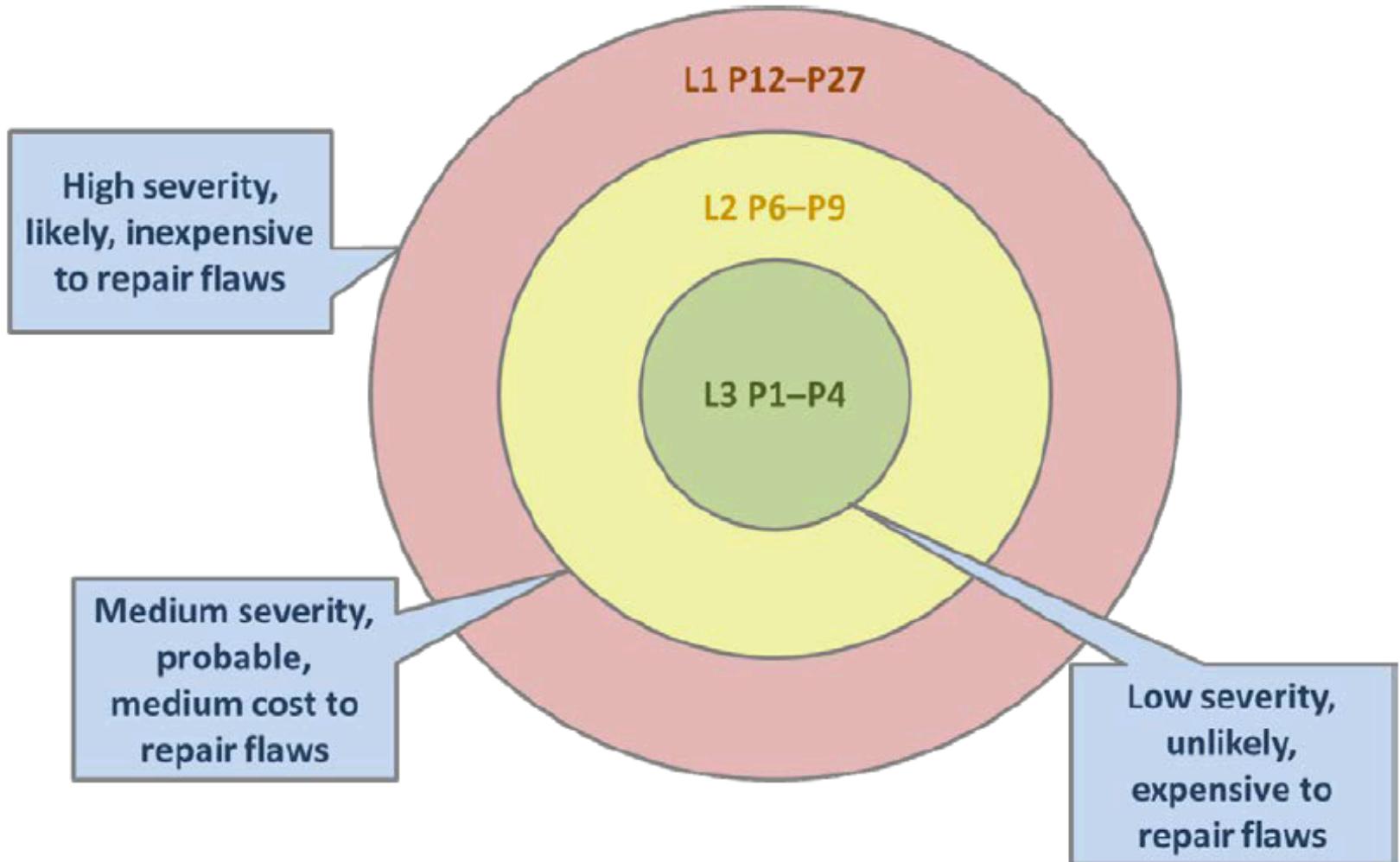
- The **three values** are then **multiplied** together for each rule
  - Provides a **measure** that can be used in **prioritizing the application of the rules**
- The products range from **1 to 27**, although only the following **10 distinct values** are possible: **1, 2, 3, 4, 6, 8, 9, 12, 18, and 27**
- Rules and recommendations with a priority in the range of
  - **1 to 4** are **Level 3**
  - **6 to 9** are **Level 2**
  - **12 to 27** are **Level 1**

# Priorities and Levels

Level	Priorities	Examples of Vulnerabilities
L1	12, 18, 27	High severity, likely, inexpensive to repair
L2	6, 8, 9	Medium severity, probable, medium cost to repair
L3	1, 2, 3, 4	Low severity, unlikely, expensive to repair

Specific projects may begin **remediation** by implementing all rules at a **particular level** before proceeding to the lower priority rules

# Priorities and Levels



# **Memory Management**

## **(SEI CERT C Coding Standard)**

- **MEM30-C:** Do not access freed memory
- **MEM31-C:** Free dynamically allocated memory when no longer needed
- **MEM33-C:** Allocate and copy structures containing a flexible array member dynamically
- **MEM34-C:** Only free memory allocated dynamically
- **MEM35-C:** Allocate sufficient memory for an object
- **MEM36-C:** Do not modify the alignment of objects by calling realloc()

# Risk Assessment Summary

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	High	Likely	Medium	P18	L1
MEM31-C	Medium	Probable	Medium	P8	L2
MEM33-C	Low	Unlikely	Low	P3	L3
MEM34-C	High	Likely	Medium	P18	L1
MEM35-C	High	Probable	High	P6	L2
MEM36-C	Low	Probable	High	P2	L3

<https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard>

# Intended learning outcomes

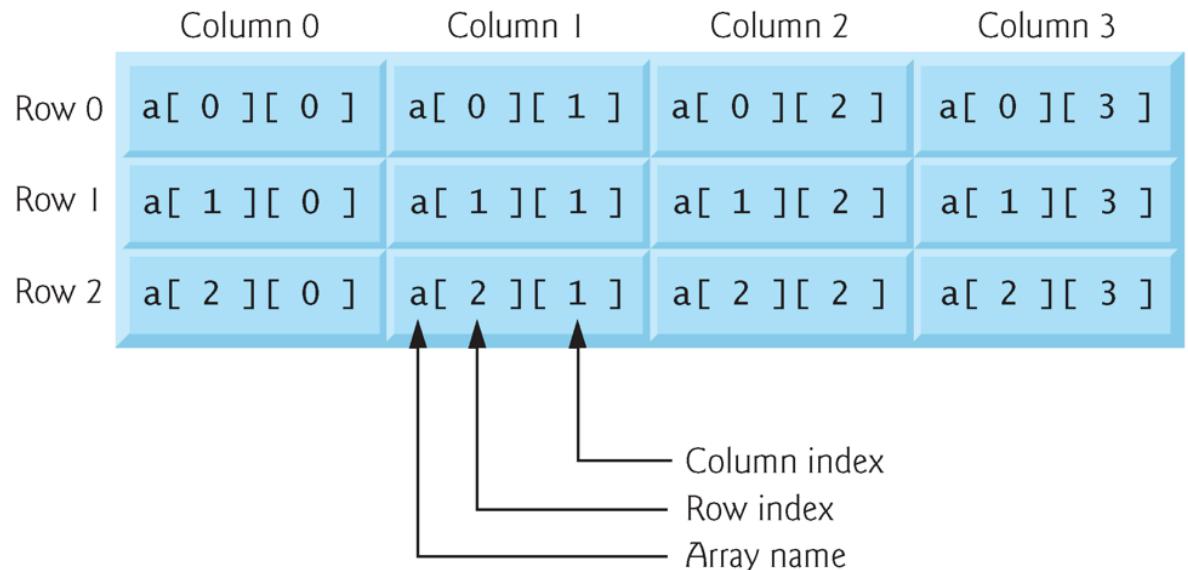
- Understand **risk assessment** to guide **software developers**
- Review **dynamic data structures** (linked list)
- Provide rules for **secure coding** in the **C programming language**
- Develop **safe, reliable, and secure systems**
- Eliminate **undefined behaviours** that can lead to undefined program behaviours and **exploitable vulnerabilities**

# Dynamic data structures

- We've studied fixed-size data structures such as **single-subscripted arrays, double-subscripted arrays and structs**

```
typedef struct account {  
    unsigned short age;  
    char name[100];  
} accountt;
```

```
int main()  
{  
    int x[3];  
    int a[3][4];  
    accountt account;  
    return 0;  
}
```



# Dynamic data structures

- We've studied fixed-size data structures such as **single-subscripted arrays**, **double-subscripted arrays** and **structs**
- Dynamic data structures
  - They can **grow** and **shrink** during execution
- Linked lists
  - Allow **insertions** and **removals** anywhere in a linked list



# Self-referential structures

- Self-referential structures
  - Structure that contains a pointer to a structure of the same type
  - Terminated with a NULL pointer (0)

```
typedef struct node {  
    int data;  
    struct node *nextPtr;  
} nodet;
```

Not setting the link in the last node of a list to NULL can lead to runtime errors
  - nextPtr
    - o Points to an object of type node
    - o Referred to as a link
    - o Ties one **node** to another **node**
  - Can be linked together to form useful data structures such as **lists**, **queues**, **stacks** and **trees**

# Dynamic memory allocation

- Dynamic memory allocation
  - Obtain and release memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - o Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - o A `void *` pointer may be assigned to any pointer
    - o If no memory available, returns `NULL`
  - Example: `nodet *newPtr = (nodet *)malloc(sizeof(nodet));`
- `free`
  - Always deallocates memory allocated by `malloc` to avoid **memory leak**
  - Takes a pointer as an argument
    - o `free (newPtr);`

# Dynamic memory allocation

Two self-referential structures linked together

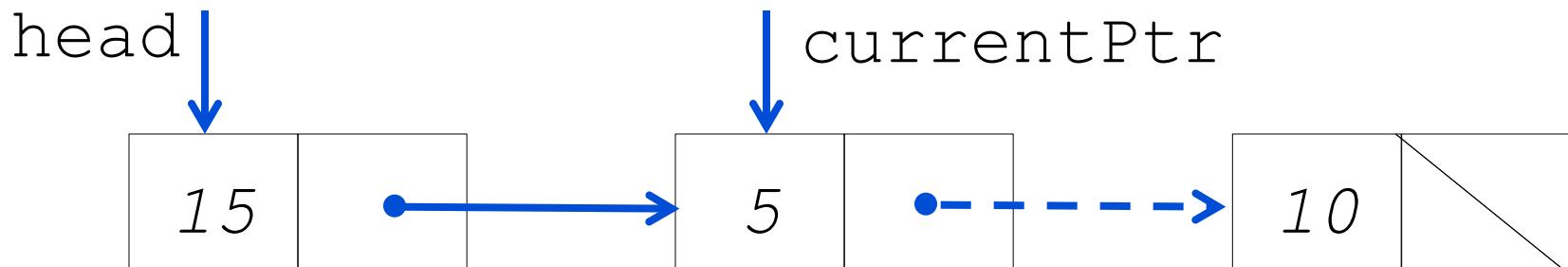


```
int main() {  
    // allocates memory  
    nodet *node1 = (nodet *)malloc(sizeof(nodet));  
    nodet *node2 = (nodet *)malloc(sizeof(nodet));  
    node1->data = 15;  
    node2->data = 10;  
    // Link node1 to node2  
    node1->nextPtr = node2;  
    node2->nextPtr = NULL;  
    // Deallocates memory allocated by malloc  
    free(node1);  
    free(node2);  
    return 0;  
}
```

If there exists no memory available, then malloc returns NULL

# Linked lists properties

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to `NULL` to mark the list's end



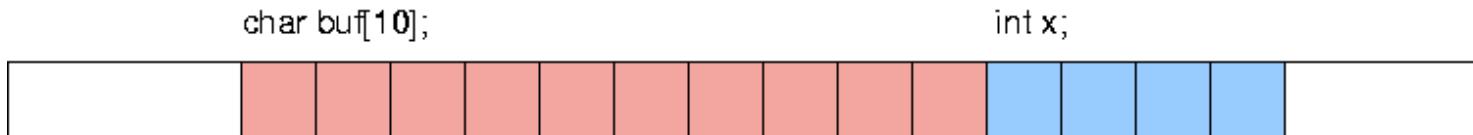
# Linked lists properties

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links
  - Accessed via a pointer to the first node of the list
  - Subsequent nodes are accessed via the link-pointer member of the current node
  - Link pointer in the last node is set to `NULL` to mark the list's end
- Use a **linked list** instead of an **array** when
  - You have an **unpredictable** number of elements
  - Your list needs to be **sorted quickly**

# Linked lists properties

- Linked lists are **dynamic**, so the length of a list can **increase** or **decrease** as necessary
- Can we change the array size after compiling the program? What are the problems here?
  - **Arrays can become full**

o An array can be declared to contain more elements than the number of data items expected, but this can waste memory



`strcpy (buf, "14 characters");`



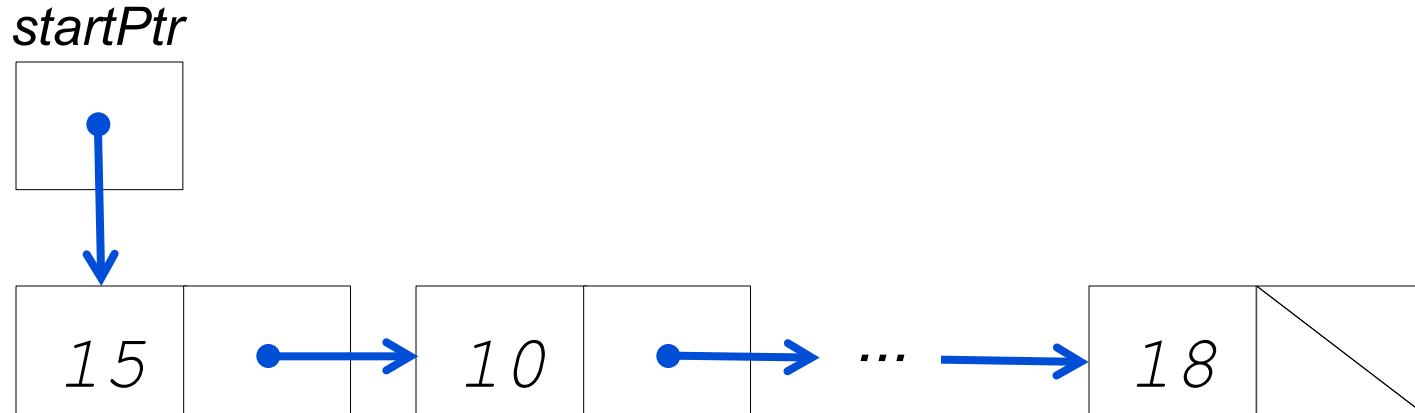
# Linked lists properties

- Linked lists are **dynamic**, so the length of a list can **increase** or **decrease** as necessary
- Can we change the array size after compiling the program? What are the problems here?
  - **Arrays** can become **full**
    - An array can be declared to contain more elements than the number of data items expected, but this can waste memory
- **Linked lists** become **full** only when the system has **insufficient memory** to satisfy dynamic storage allocation requests
  - It can provide better **memory utilization**

# Linked lists properties

- Linked-list nodes are **normally not stored contiguously in memory**
  - How arrays are stored in memory? What would be the advantage here?
    - This allows **immediate access** since the address of any element can be calculated directly based on its position relative to the beginning of the array
      - \* Linked lists do not afford such immediate access
- Logically, however, the nodes of a linked list appear to be contiguous
  - Pointers take up **space** and dynamic memory allocation incurs the **overhead of function calls**

# A graphical representation of a linked list



```
int main() {  
    ...  
    // Link the nodes  
    startPtr = node1;  
    node1->nextPtr = node2;  
    node2->nextPtr = node3;  
    node3->nextPtr = NULL;  
    ...  
    return 0;  
}
```

Pointers should be initialised before they're used

A structure's size is not necessarily the sum of the size of its members (machine-dependent boundary alignment)

# Error prevention when using linked lists

- If dynamically allocated memory is no longer needed, use **free** to return it to the system
  - Why must we set that pointer to NULL?
    - eliminate the possibility that the program could refer to memory that's been reclaimed and which may have already been allocated for another purpose
- Is it an error to free memory not allocated dynamically with `malloc`?
  - Referring to memory that has been freed is an error, which results in the program crashing (**double free**)

# Exercise

- Fill in the blanks in each of the following:
  - A self- \_\_\_\_\_ structure is used to form dynamic data structures.
  - Function \_\_\_\_\_ is used to dynamically allocate memory.
  - A(n) \_\_\_\_\_ is a specialized version of a linked list in which nodes can be inserted and deleted only from the start of the list.
  - Functions that look at a linked list but do not modify it are referred to as \_\_\_\_\_.
  - Function \_\_\_\_\_ is used to reclaim dynamically allocated memory.

# Illustrative example about linked lists

- We will show an example of linked list that manipulates a list of characters
- You can insert a character in the list in alphabetical order (function `insert`) or to delete a character from the list (function `delete`)

```
1 // Fig. 12.3: fig12_03.c
2 // Inserting and deleting nodes in a list
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 // self-referential structure
7 struct listNode {
8     char data; // each listNode contains a character
9     struct listNode *nextPtr; // pointer to next node
10};
11
12 typedef struct listNode ListNode; // synonym for struct listNode
13 typedef ListNode *ListNodePtr; // synonym for ListNode*
14
15 // prototypes
16 void insert(ListNodePtr *sPtr, char value);
17 char delete(ListNodePtr *sPtr, char value);
18 int isEmpty(ListNodePtr sPtr);
19 void printList(ListNodePtr currentPtr);
20 void instructions(void);
21
22 int main(void)
23 {
```

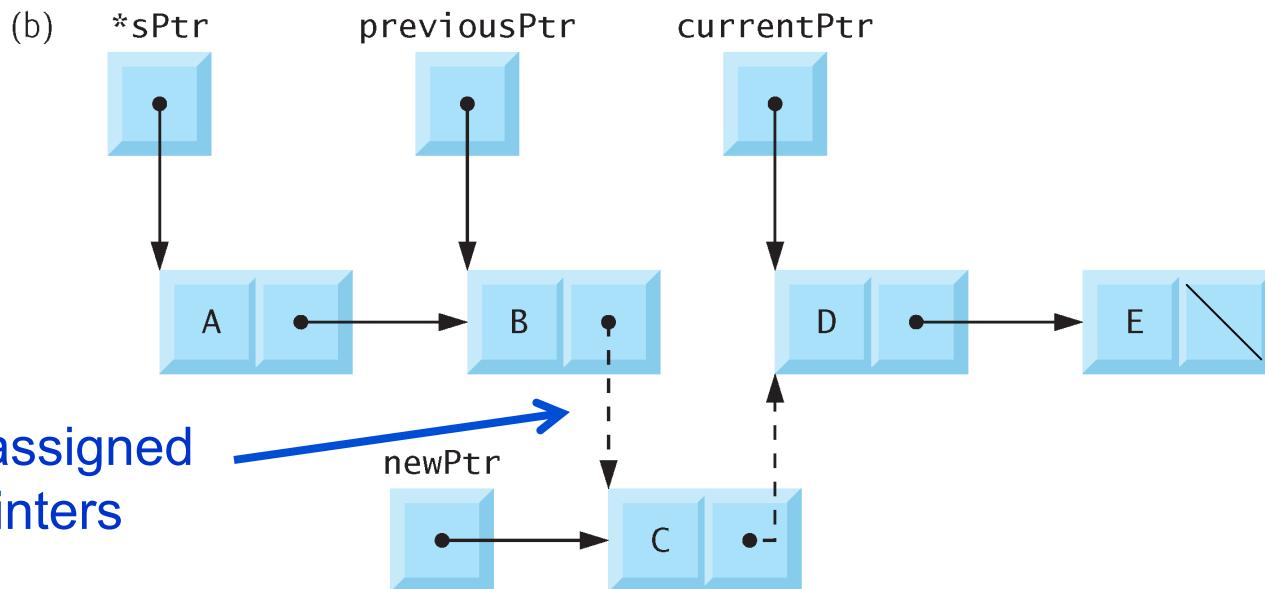
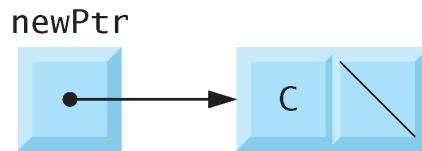
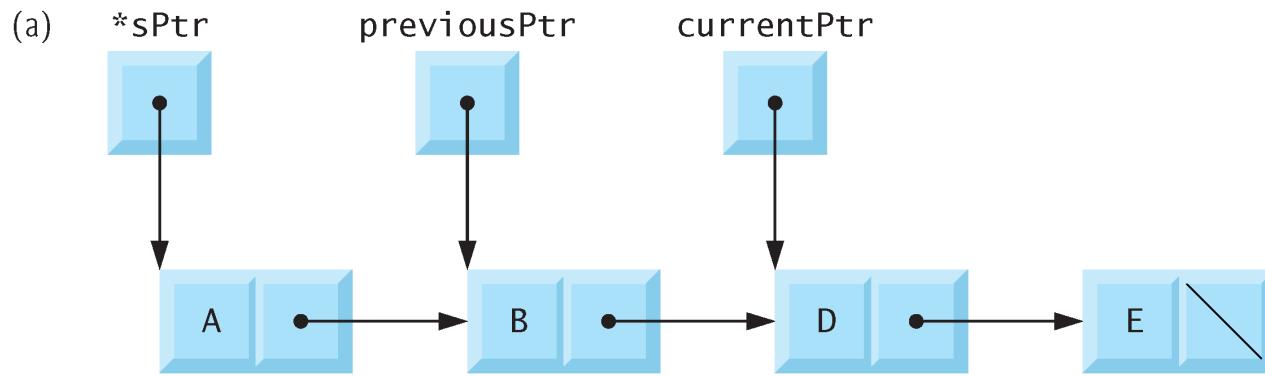
Inserting and deleting nodes in a list (Part 1 of 8)

```
24 ListNodePtr startPtr = NULL; // initially there are no nodes
25 char item; // char entered by user
26
27 instructions(); // display the menu
28 printf("%s", "? ");
29 unsigned int choice; // user's choice
30 scanf("%u", &choice);
31
32 // Loop while user does not choose 3
33 while (choice != 3) {
34
35     switch (choice) {
36         case 1:
37             printf("%s", "Enter a character: ");
38             scanf("\n%c", &item);
39             insert(&startPtr, item); // insert item in list
40             printList(startPtr);
41             break;
42         case 2: // delete an element
43             // if list is not empty
44             if (!isEmpty(startPtr)) {
45                 printf("%s", "Enter character to be deleted: ");
46                 scanf("\n%c", &item);
47             }
48         }
49     }
50 }
```

Inserting and deleting nodes in a list (Part 2 of 8)

```
48 // if character is found, remove it
49     if (delete(&startPtr, item)) { // remove item
50         printf("%c deleted.\n", item);
51         printList(startPtr);
52     }
53     else {
54         printf("%c not found.\n\n", item);
55     }
56 }
57 else {
58     puts("List is empty.\n");
59 }
60
61     break;
62 default:
63     puts("Invalid choice.\n");
64     instructions();
65     break;
66 }
67
68     printf("%s", "? ");
69     scanf("%u", &choice);
70 }
71 }
```

Inserting and deleting nodes in a list (Part 3 of 8)



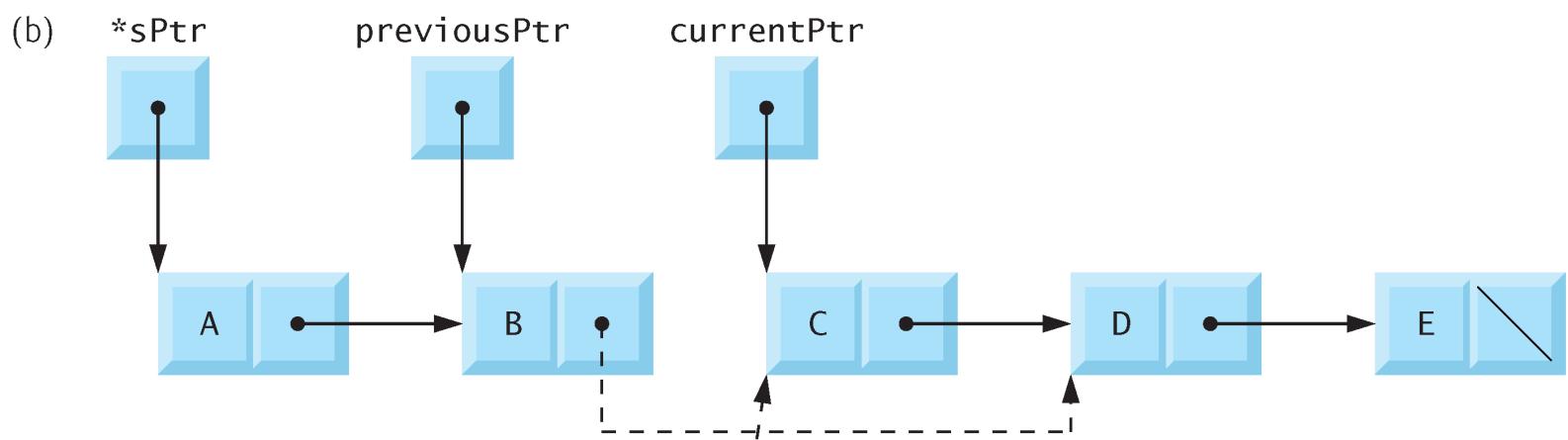
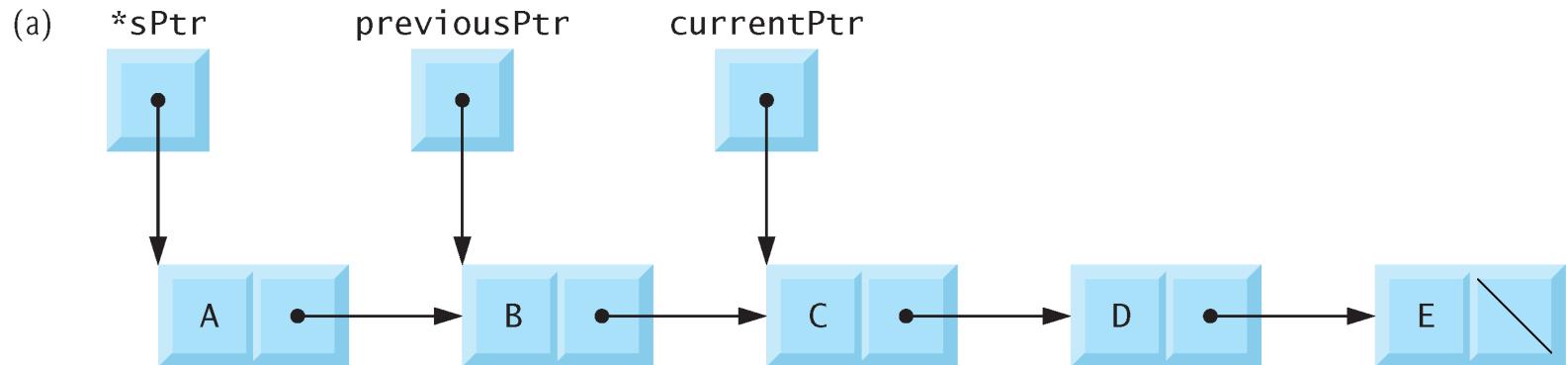
Inserting a node in order in a list

```
72     puts("End of run.");
73 }
74
75 // display program instructions to user
76 void instructions(void)
77 {
78     puts("Enter your choice:\n"
79         "    1 to insert an element into the list.\n"
80         "    2 to delete an element from the list.\n"
81         "    3 to end.");
82 }
83
84 // insert a new value into the list in sorted order
85 void insert(ListNodePtr *sPtr, char value)
86 {
87     ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node
88
89     if (newPtr != NULL) { // is space available?
90         newPtr->data = value; // place value in node
91         newPtr->nextPtr = NULL; // node does not link to another node
92
93         ListNodePtr previousPtr = NULL;
94         ListNodePtr currentPtr = *sPtr;
95 }
```

Inserting and deleting nodes in a list (Part 4 of 8)

```
96     // loop to find the correct location in the list
97     while (currentPtr != NULL && value > currentPtr->data) {
98         previousPtr = currentPtr; // walk to ...
99         currentPtr = currentPtr->nextPtr; // ... next node
100    }
101
102    // insert new node at beginning of list
103    if (previousPtr == NULL) {
104        newPtr->nextPtr = *sPtr;
105        *sPtr = newPtr;
106    }
107    else { // insert new node between previousPtr and currentPtr
108        previousPtr->nextPtr = newPtr;
109        newPtr->nextPtr = currentPtr;
110    }
111    }
112    else {
113        printf("%c not inserted. No memory available.\n", value);
114    }
115 }
116 }
```

Inserting and deleting nodes in a list (Part 5 of 8)



tempPtr

tempPtr is a local automatic variable

Deleting a node from a list

```
117 // delete a list element
118 char delete(ListNodePtr *sPtr, char value)
119 {
120     // delete first node if a match is found
121     if (value == (*sPtr)->data) {
122         ListNodePtr tempPtr = *sPtr; // hold onto node being removed
123         *sPtr = (*sPtr)->nextPtr; // de-thread the node
124         free(tempPtr); // free the de-threaded node
125         return value;
126     }
127     else {
128         ListNodePtr previousPtr = *sPtr;
129         ListNodePtr currentPtr = (*sPtr)->nextPtr;
130
131         // loop to find the correct location in the list
132         while (currentPtr != NULL && currentPtr->data != value) {
133             previousPtr = currentPtr; // walk to ...
134             currentPtr = currentPtr->nextPtr; // ... next node
135         }
136     }
```

Inserting and deleting nodes in a list (Part 6 of 8)

```
137     // delete node at currentPtr
138     if (currentPtr != NULL) {
139         ListNodePtr tempPtr = currentPtr;
140         previousPtr->nextPtr = currentPtr->nextPtr;
141         free(tempPtr);
142         return value;
143     }
144 }
145
146     return '\0';
147 }
148
149 // return 1 if the list is empty, 0 otherwise
150 int isEmpty(ListNodePtr sPtr)
151 {
152     return sPtr == NULL;
153 }
154
```

Inserting and deleting nodes in a list (Part 7 of 8)

```
155 // print the list
156 void printList(ListNodePtr currentPtr)
157 {
158     // if list is empty
159     if (isEmpty(currentPtr)) {
160         puts("List is empty.\n");
161     }
162     else {
163         puts("The list is:");
164
165         // while not the end of the list
166         while (currentPtr != NULL) {
167             printf("%c --> ", currentPtr->data);
168             currentPtr = currentPtr->nextPtr;
169         }
170
171         puts("NULL\n");
172     }
173 }
```

Inserting and deleting nodes in a list (Part 8 of 8)

Enter your choice:

- 1 to insert an element into the list.
- 2 to delete an element from the list.
- 3 to end.

? 1

Enter a character: B

The list is:

B --> NULL

? 1

Enter a character: A

The list is:

A --> B --> NULL

? 1

Enter a character: C

The list is:

A --> B --> C --> NULL

? 2

Enter character to be deleted: D

D not found.

Sample output for the program (Part 1 of 2)

```
? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL
```

```
? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL
```

```
? 2
Enter character to be deleted: A
A deleted.
List is empty.
```

```
? 4
Invalid choice.
```

```
Enter your choice:
 1 to insert an element into the list.
 2 to delete an element from the list.
 3 to end.
? 3
End of run.
```

Sample output for the program (Part 2 of 2)

# Analysis of the linked list

## OPERATION

add to start of list  
add to end of list  
add at given index

find an object  
remove first element  
remove last element  
remove at given index

size

## RUNTIME (Big-O)

```

72     puts("End of run.");
73 }
74
75 // display program instructions to user
76 void instructions(void)
77 {
78     puts("Enter your choice:\n"
79         "    1 to insert an element into the list.\n"
80         "    2 to delete an element from the list.\n"
81         "    3 to end.");
82 }
83
84 // insert a new value into the list in sorted order
85 void insert(ListNodePtr *sPtr, char value)
86 {
87     ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node
88
89     if (newPtr != NULL) { // is space available?
90         newPtr->data = value; // place value in node
91         newPtr->nextPtr = NULL; // node does not link to another node
92
93         ListNodePtr previousPtr = NULL;
94         ListNodePtr currentPtr = *sPtr;
95

```

O(1)

Analysis of the linked list (insert) – Part 1 of 2

```
96     // loop to find the correct location in the list
97     while (currentPtr != NULL && value > currentPtr->data) {
98         previousPtr = currentPtr; // walk to ...
99         currentPtr = currentPtr->nextPtr; // ... next node
100    }
101
102    // insert new node at beginning of list
103    if (previousPtr == NULL) {
104        newPtr->nextPtr = *sPtr;
105        *sPtr = newPtr;
106    }
107    else { // insert new node between previousPtr and currentPtr
108        previousPtr->nextPtr = newPtr;
109        newPtr->nextPtr = currentPtr;
110    }
111    }
112    else {
113        printf("%c not inserted. No memory available.\n", value);
114    }
115 }
116 }
```

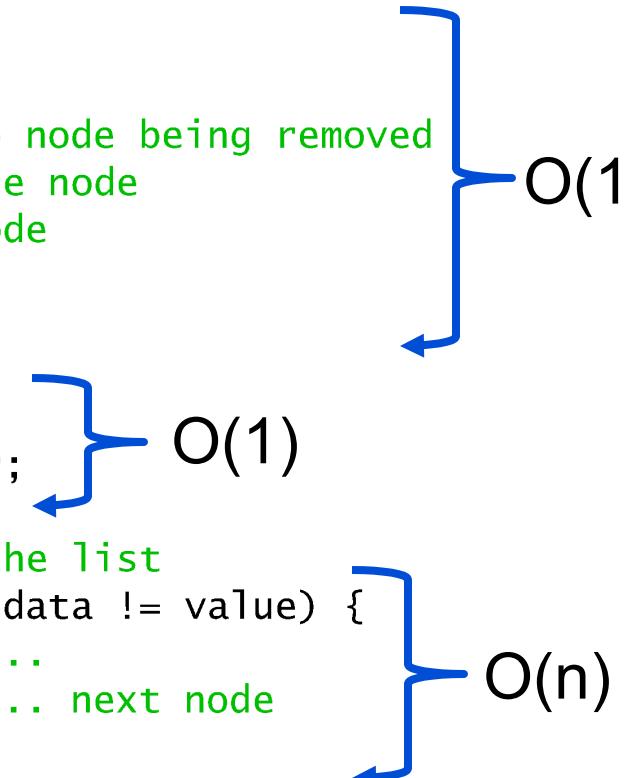
O(n)

O(1)

Analysis of the linked list (insert) – Part 2 of 2

**Insert -- runtime:  $O(1)+O(n)+O(1) = O(n)$**

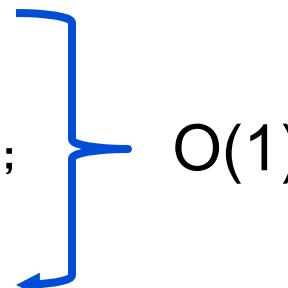
```
117 // delete a list element
118 char delete(ListNodePtr *sPtr, char value)
119 {
120     // delete first node if a match is found
121     if (value == (*sPtr)->data) {
122         ListNodePtr tempPtr = *sPtr; // hold onto node being removed
123         *sPtr = (*sPtr)->nextPtr; // de-thread the node
124         free(tempPtr); // free the de-threaded node
125         return value;
126     }
127     else {
128         ListNodePtr previousPtr = *sPtr;
129         ListNodePtr currentPtr = (*sPtr)->nextPtr;
130
131         // loop to find the correct location in the list
132         while (currentPtr != NULL && currentPtr->data != value) {
133             previousPtr = currentPtr; // walk to ...
134             currentPtr = currentPtr->nextPtr; // ... next node
135         }
136     }
}
```



The code implements a linked list deletion. It handles two cases: deleting the head node if it matches the value, or traversing the list to find the correct node. The traversal part is annotated with  $O(n)$ , while the head deletion part is annotated with  $O(1)$ .

Analysis of the linked list (delete) – Part 1 of 2

```
137     // delete node at currentPtr
138     if (currentPtr != NULL) {
139         ListNodePtr tempPtr = currentPtr;
140         previousPtr->nextPtr = currentPtr->nextPtr;
141         free(tempPtr);
142         return value;
143     }
144 }
145
146     return '\0';
147 }
148
149 // return 1 if the list is empty, 0 otherwise
150 int isEmpty(ListNodePtr sPtr)
151 {
152     return sPtr == NULL;
153 }
154
```



Analysis of the linked list (delete) – Part 2 of 2

**Delete -- runtime:  $O(1)+O(n)+O(1) = O(n)$**

# Analysis of the linked list

<u>OPERATION</u>	<u>RUNTIME (Big-O)</u>
add to start of list	$O(1)$
add to end of list	$O(n)$
add at given index	$O(n)$
find an object	$O(n)$
remove first element	$O(1)$
remove last element	$O(n)$
remove at given index	$O(n)$
size	$O(1)$

# Intended learning outcomes

- Understand **risk assessment** to guide **software developers**
- Review **dynamic data structures** (linked list)
- Provide rules for **secure coding** in the **C programming language**
- Develop **safe**, **reliable**, and **secure** systems
- Eliminate **undefined behaviours** that can lead to undefined program behaviours and **exploitable vulnerabilities**

# Do not access freed memory (MEM30-C)

- Evaluating a **pointer into memory that has been deallocated** by a memory management function is **undefined behavior**
- **Pointers to memory** that has been **deallocated** are called **dangling pointers**
  - Accessing a dangling pointer can result in **exploitable vulnerabilities**
- Using the **value of a pointer** that refers to space **deallocated** by a call to the free() or realloc() function is **undefined behavior**

# Noncompliant Code Example

- Kernighan and Ritchie shows the **incorrect technique** for **freeing the memory** associated with a linked list

```
#include <stdlib.h>
struct node {
    int value;
    struct node *next;
};
void free_list(struct node *head) {
    for (struct node *p = head; p != NULL; p
= p->next) {
        free(p);
    }
}
```

# Compliant Solution

- $p$  is freed before  $p->next$  is executed, so that  $p->next$  reads memory that has already been freed

```
#include <stdlib.h>
struct node {
    int value;
    struct node *next;
};
void free_list(struct node *head) {
    struct node *q;
    for(struct node *p=head; p!=NULL; p=q) {
        q = p->next;
        free(p);
    }
}
```

# Risk Assessment

- Reading memory that has been freed can lead to
  - abnormal program termination
  - denial-of-service attacks
- Writing memory that has already been freed can lead to the execution of arbitrary code
- Reading a pointer to deallocated memory is undefined behavior
  - the pointer value is indeterminate and might be a trap representation

Rule	Severity	Likelihood	Remediation cost	Priority	Level
MEM30-C	High	Likely	Medium	P18	L1

# **Free dynamically allocated memory when no longer needed (MEM31-C)**

- Before the lifetime of the last pointer that stores the return value of a call to a standard memory allocation function has ended, it must be matched by a call to *free()* with that pointer value

# Noncompliant Code Example

- The object allocated by the call to **malloc()** is not freed before the end of the lifetime of the last pointer *text\_buffer* referring to the object

```
#include <stdlib.h>
enum { BUFFER_SIZE = 32 };
int f(void) {
    char *text_buffer=(char
*)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }
    return 0;
}
```

# Compliant Solution

- The **pointer** must be **deallocated** with a call to **free()**:

```
#include <stdlib.h>
enum { BUFFER_SIZE = 32 };
int f(void) {
    char *text_buffer=(char
*)malloc(BUFFER_SIZE);
    if (text_buffer == NULL) {
        return -1;
    }
    free(text_buffer);
    return 0;
}
```

# Risk Assessment

- Failing to **free memory** can result in
  - **Exhaustion of system memory resources**
  - **denial-of-service attack**

Rule	Severity	Likelihood	Remediation cost	Priority	Level
MEM31-C	Medium	Probable	Medium	P8	L2

# Allocate and copy structures containing a flexible array member dynamically (MEM33-C)

- The C Standard, 6.7.2.1, paragraph 18 [ISO/IEC 9899:2011], says:

*“As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a flexible array member. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply.”*

# Noncompliant Code Example

- This example uses **automatic storage** for a structure containing a **flexible array member**

```
#include <stddef.h>
struct flex_array_struct {
    size_t num;
    int data[ ];
};

void func(void) {
    struct flex_array_struct flex_struct;
    size_t array_size = 4;
    /* Initialize structure */
    flex_struct.num = array_size;
    for (size_t i = 0; i < array_size; ++i) {
        flex_struct.data[i] = 0;
    }
    ...
}
```

# Compliant Solution

- This solution **dynamically allocates storage** for *flex\_array\_struct*:

```
#include <stdlib.h>
struct flex_array_struct {
    size_t num;
    int data[ ];
};

void func(void) {
    struct flex_array_struct *flex_struct;
    size_t array_size = 4;
    /* Dynamically allocate memory for the struct */
    flex_struct = (struct flex_array_struct
* )malloc(
        sizeof(struct flex_array_struct)
        + sizeof(int) * array_size);
```

# Compliant Solution

```
if (flex_struct == NULL) {
    /* Handle error */
}

/* Initialize structure */
flex_struct->num = array_size;

for (size_t i = 0; i < array_size; ++i) {
    flex_struct->data[i] = 0;
}
}
```

# Risk Assessment

- **Failure to use structures with flexible array members correctly can result in undefined behavior**

Rule	Severity	Likelihood	Remediation cost	Priority	Level
MEM33-C	Low	Unlikely	Low	P3	L3

# Only free memory allocated dynamically (MEM34-C)

- C Standard, Annex J [ISO/IEC 9899:2011], states that the behavior of a program is undefined when:  
*“The pointer argument to the free or realloc function does not match a pointer earlier returned by a memory management function, or the space has been deallocated by a call to free or realloc.”*
- Freeing memory that is not allocated dynamically can result in **heap corruption**
  - Do not call free() on a pointer other than one returned by a standard memory allocation function

# Noncompliant Code Example

- In this noncompliant example, the pointer parameter to *realloc()*, *buf*, does not refer to dynamically allocated memory

```
#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {
    char buf[BUFSIZE];
    char *p = (char *)realloc(buf, 2 * BUFSIZE);
    if (p == NULL) {
        /* Handle error */
    }
}
```

# Compliant Solution

- In this compliant solution, *buf* refers to dynamically allocated memory:

```
#include <stdlib.h>

enum { BUFSIZE = 256 };

void f(void) {
    char *buf = (char *)malloc(BUFSIZE *
sizeof(char));
    char *p = (char *)realloc(buf, 2 * BUFSIZE);
    if (p == NULL) {
        /* Handle error */
    }
}
```

# Risk Assessment

- The consequences of this error **depend on the implementation**
  - they range from **nothing** to **arbitrary code execution** if that memory is reused by *malloc()*

Rule	Severity	Likelihood	Remediation cost	Priority	Level
MEM34-C	High	Likely	Medium	P18	L1

# Allocate sufficient memory for an object (MEM35-C)

- The **types of integer expressions** used as size arguments to *malloc()*, *calloc()*, *realloc()*, or *aligned\_alloc()* must have **sufficient range** to represent the size of the objects to be stored
  - If **size** arguments are **incorrect** or can be **manipulated by an attacker**, then a **buffer overflow** may occur
  - Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer
- Typically, the amount of memory to allocate will be the **size of the type of object** to allocate

# Noncompliant Code Example

- An **insufficient amount of memory** can be allocated where `sizeof(long)` is larger than `sizeof(int)`, which can cause a **heap buffer overflow**

```
#include <stdint.h>
#include <stdlib.h>
void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(long)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(int));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}
```

# Compliant Solution

- This compliant solution uses `sizeof(long)` to correctly size the memory allocation:

```
#include <stdint.h>
#include <stdlib.h>
void function(size_t len) {
    long *p;
    if (len == 0 || len > SIZE_MAX / sizeof(long)) {
        /* Handle overflow */
    }
    p = (long *)malloc(len * sizeof(long));
    if (p == NULL) {
        /* Handle error */
    }
    free(p);
}
```

# Risk Assessment

- Providing **invalid size arguments** to memory allocation functions can lead to
  - **buffer overflows**
  - the **execution of arbitrary code** with the permissions of the vulnerable process

Rule	Severity	Likelihood	Remediation cost	Priority	Level
MEM35-C	High	Probable	High	P6	L2

# Do not modify the alignment of objects by calling `realloc()` **(MEM36-C)**

- Do not invoke `realloc()` to modify the size of allocated objects that have stricter alignment requirements than those guaranteed by `malloc()`
- Storage allocated by a call to the standard `aligned_alloc()` function, e.g., can have stricter than normal alignment requirements
- The C standard requires only that a pointer returned by `realloc()` be suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement

# Noncompliant Code Example

- This code example returns a pointer to allocated memory that has been aligned to a **4096-byte boundary**

```
#include <stdlib.h>
void func(void) {
    size_t resize = 1024;
    size_t alignment = 1 << 12;
    int *ptr;
    int *ptr1;
    if (NULL == (ptr = (int *)aligned_alloc(alignment,
sizeof(int))))) {
        /* Handle error */
    }
    if (NULL == (ptr1 = (int *)realloc(ptr, resize))) {
        /* Handle error */
    }
    ...
}
```

# Noncompliant Code Example

- When compiled with GCC 4.1.2 and run on the x86\_64 Red Hat Linux platform, the following code produces the following output:

```
memory aligned to 4096 bytes
ptr = 0x1621b000
```

```
After realloc():
ptr1 = 0x1621a010
```

ptr1 is no longer aligned to 4096 bytes

# Compliant Solution

```
...
void func(void) {
    size_t resize = 1024, alignment = 1 << 12;
    int *ptr, *ptr1;
    if (NULL == (ptr = (int *)aligned_alloc(alignment,
                                                sizeof(int)))) {
        /* Handle error */
    }
    if (NULL == (ptr1 = (int *)aligned_alloc(alignment,
                                                resize))) {
        /* Handle error */
    }
    if (NULL == (memcpy(ptr1, ptr, sizeof(int)))) {
        /* Handle error */
    }
    free(ptr);
}
```

# Risk Assessment

- **Improper alignment** can lead to **arbitrary memory locations** being accessed and written to

Rule	Severity	Likelihood	Remediation cost	Priority	Level
MEM36-C	Low	Probable	High	P2	L3

# Summary (Secure C Programming)

## ***Chapter 8 of the CERT Secure C Coding Standard***

- Chapter 8 of the CERT Secure C Coding Standard is dedicated to memory-management recommendations and rules—many apply to the uses of pointers and dynamic-memory allocation presented in this chapter.
- For more information, visit [www.securecoding.cert.org](http://www.securecoding.cert.org).

# Summary (Secure C Programming)

- Pointers should not be left uninitialized
- They should be assigned either NULL or the address of a valid item in memory
- When you use free to deallocate dynamically allocated memory, the pointer passed to free is not assigned a new value, so it still points to the memory location where the dynamically allocated memory used to be

# Summary (Secure C Programming)

- Using a pointer that's been freed can lead to program crashes and security vulnerabilities
- When you free dynamically allocated memory, you should immediately assign the pointer either NULL or a valid address
- We chose not to do this for local pointer variables that immediately go out of scope after a call to free

# Summary (Secure C Programming)

- Undefined behavior occurs when you attempt to use free to deallocate dynamic memory that was already deallocated—this is known as a “double free vulnerability”
- To ensure that you don’t attempt to deallocate the same memory more than once, immediately set a pointer to NULL after the call to free—attempting to free a NULL pointer has no effect

# Summary (Secure C Programming)

- Function malloc returns NULL if it's unable to allocate the requested memory
- You should always ensure that malloc did not return NULL before attempting to use the pointer that stores malloc's return value