Systems and Software
Verification Laboratory

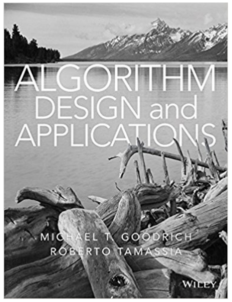
Secure C Programming: Memory Management

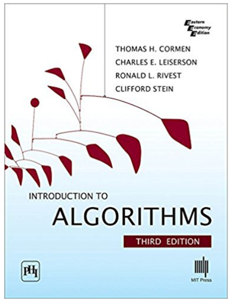
Lucas Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk

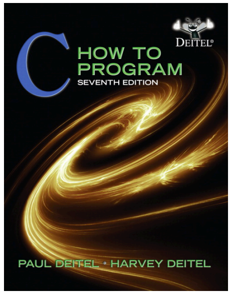
Secure C Programming

- Lucas Cordeiro (Formal Methods Group)
  - *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"

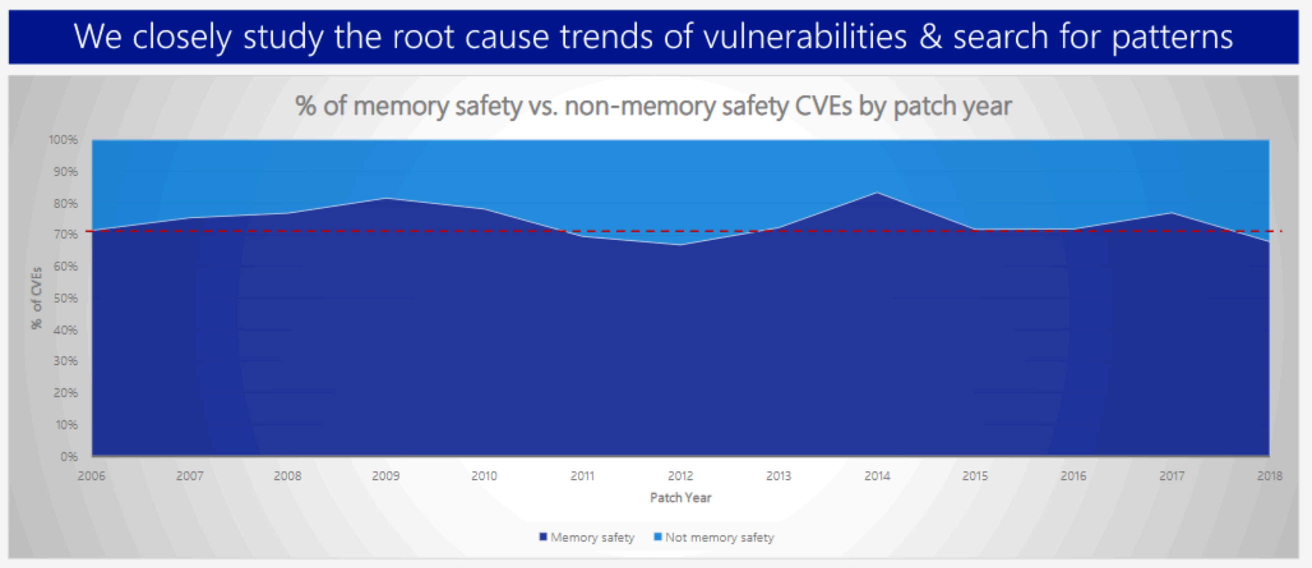






70 percent of all security bugs are memory safety issues

- *“The majority of vulnerabilities are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code. As Microsoft increases its code base and uses more Open Source Software in its code, this problem isn’t getting better, it’s getting worse.”*



We closely study the root cause trends of vulnerabilities & search for patterns

% of memory safety vs. non-memory safety CVEs by patch year

https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Dereferencing NULL pointers can allow attackers to execute code

Security

Clever attack exploits fully-patched Linux kernel

'NULL pointer' bug plagues even super max versions

By Dan Goodin 17 Jul 2009 at 22:32

76 🗨 SHARE ▼

A recently published attack exploiting newer versions of the Linux kernel is getting plenty of notice because it works even when security enhancements are running and the bug is virtually impossible to detect in source code reviews.

The exploit code was released Friday by Brad Spengler of grsecurity, a developer of applications that enhance the security of the open-source OS. While it targets Linux versions that have yet to be adopted by most vendors, the bug has captured the attention of security researchers, who say it exposes overlooked weaknesses.

https://www.theregister.co.uk/2009/07/17/linux_kernel_exploit/

Intended learning outcomes

- Understand **risk assessment** to guide **software developers**

Intended learning outcomes

- Understand **risk assessment** to guide **software developers**
- Review **dynamic data structures** (linked list)

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**

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

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

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

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	Likelly

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

Detection and Correction

- How will we cope with detection and correction?

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

Detection and Correction

- How will we cope with detection and correction?

Value	Meaning	Detection	Correction
1	High	Manual **(Code Inspection)**	Manual
2	Medium	Automatic	Manual
3	Low	Automatic	Automatic

Detection and Correction

- How will we cope with detection and correction?

Value	Meaning	Detection	Correction
1	High	Manual **(Code Inspection)**	Manual
2	Medium	Automatic **(Static and Dynamic Analysis)**	Manual
3	Low	Automatic	Automatic

Detection and Correction

- How will we cope with detection and correction?

Value	Meaning	Detection	Correction
1	High	Manual **(Code Inspection)**	Manual
2	Medium	Automatic **(Static and Dynamic Analysis)**	Manual
3	Low	Automatic **(Static and Dynamic Analysis)**	Automatic

Detection and Correction

- How will we cope with detection and correction?

Value	Meaning	Detection	Correction
1	High	Manual **(Code Inspection)**	Manual
2	Medium	Automatic **(Static and Dynamic Analysis)**	Manual
3	Low	Automatic **(Static and Dynamic Analysis)**	Automatic **(Fault Localisation and Repair)**

Risk Management

- The **three values are then multiplied** together for each rule: *severity*, *likelihood* and *remediation cost*
    - Provides a **measure** that can be used in **prioritizing the application of the rules**

Risk Management

- The **three values are then multiplied** together for each rule: *severity*, *likelihood* and *remediation cost*
  - 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**

Risk Management

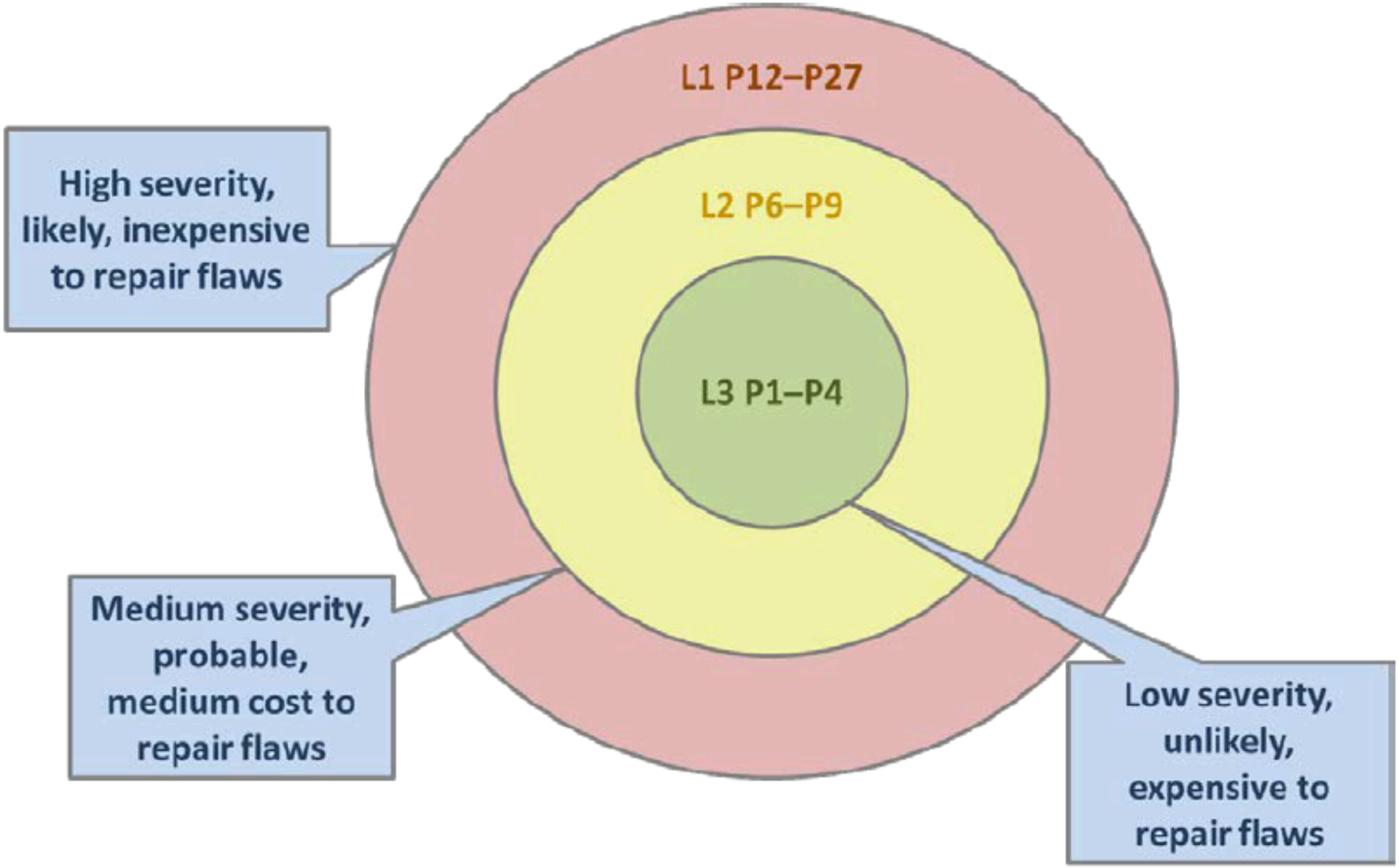
- The **three values are then multiplied** together for each rule: *severity*, *likelihood* and *remediation cost*
  - Provides a **measure** that can be used in **prioritizing the application of the rules**
- The products range from **1 to 27**, although only the following **10 distinct values** are 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



L1 P12–P27

L2 P6–P9

L3 P1–P4

High severity, likely, inexpensive to repair flaws

Medium severity, probable, medium cost to repair flaws

Low severity, unlikely, expensive to repair flaws

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()

https://wiki.sei.cmu.edu/confluence/display/c

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

Risk Assessment Summary

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	High (3)	Likely (3)	Medium (2)	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

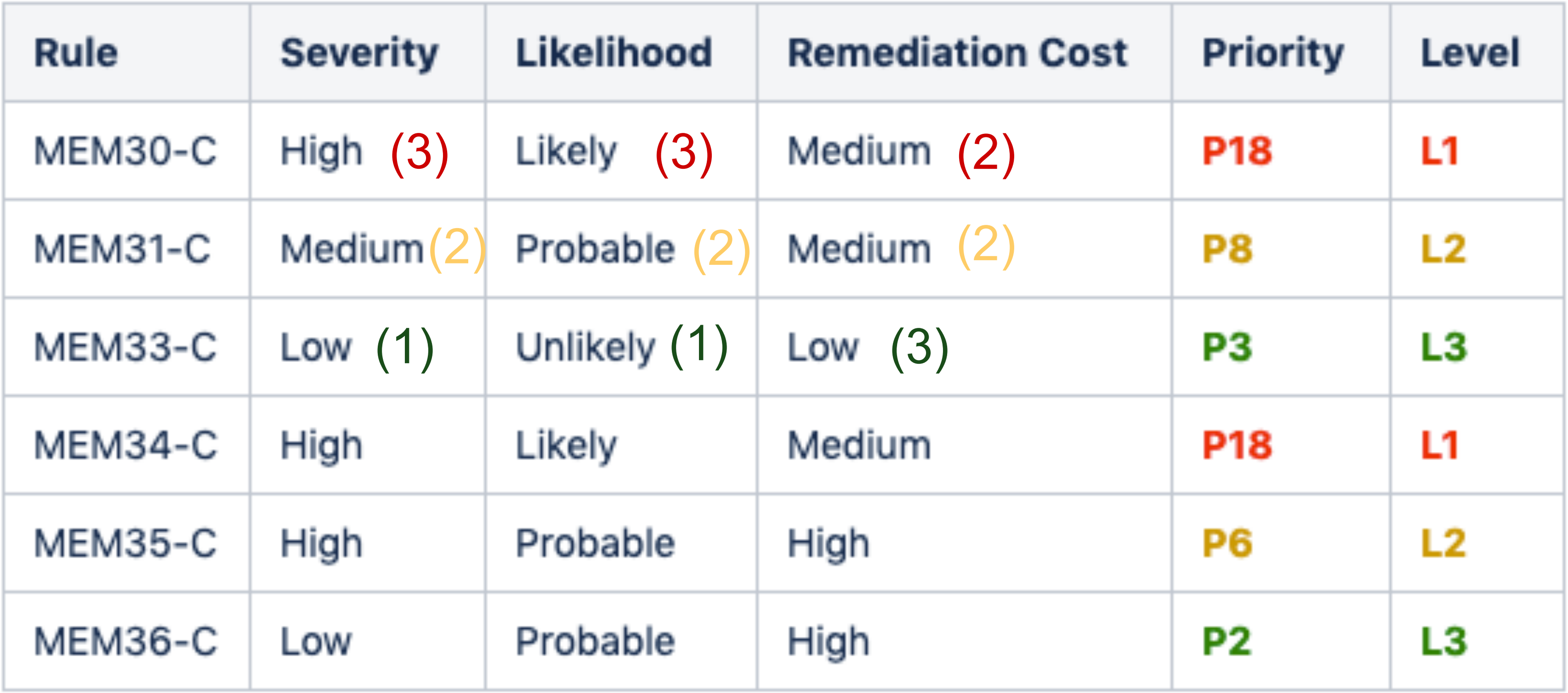
https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard

Risk Assessment Summary

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	High (3)	Likely (3)	Medium (2)	**P18**	**L1**
MEM31-C	Medium (2)	Probable (2)	Medium (2)	**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

Risk Assessment Summary



Rule	Severity	Likelihood	Remediation Cost	Priority	Level
MEM30-C	High (3)	Likely (3)	Medium (2)	P18	L1
MEM31-C	Medium (2)	Probable (2)	Medium (2)	P8	L2
MEM33-C	Low (1)	Unlikely (1)	Low (3)	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

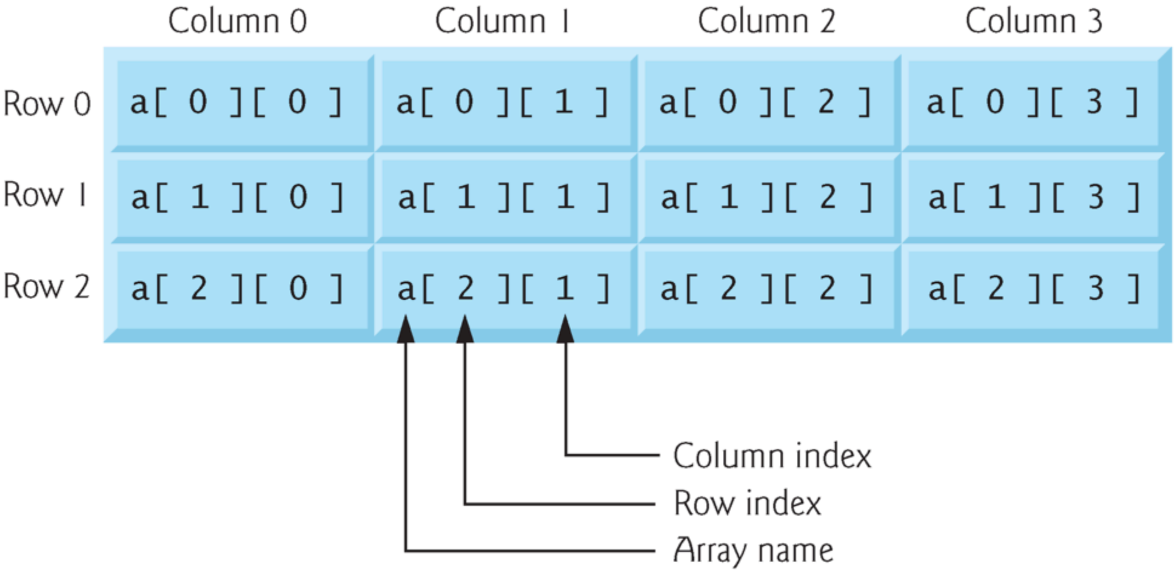
- Understand **risk assessment** to guide **software developers**
- Review **dynamic data structures** (linked list)
- Provide rules for **secure coding** in the **C programming language**
- Develop **safe**, **reliable**, and **secure** systems
- Eliminate **undefined behaviours** that can lead to undefined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

Dynamic data structures

- Examples of fixed-size data structures include **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 acount;
  return 0;
}
```



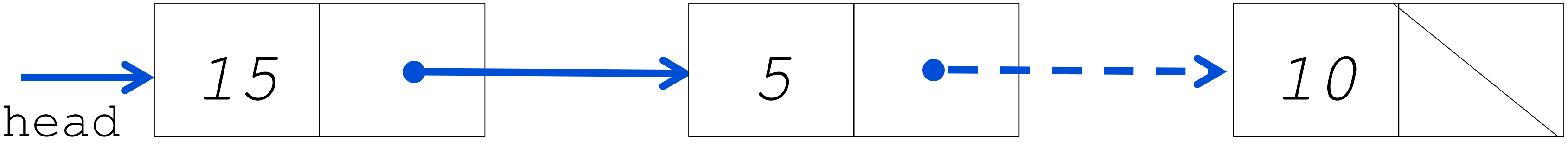
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Dynamic data structures

- Examples of fixed-size data structures include **single-subscripted arrays**, **double-subscripted arrays** and `structs`
- Dynamic data structures
  - They can **grow** and **shrink** during execution

Dynamic data structures

- Examples of fixed-size data structures include **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

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;
```

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`
    - Points to an object of type node
    - Referred to as a link
    - Ties one **node** to another **node**

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`
    - Points to an object of type node
    - Referred to as a link
    - 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

Dynamic memory allocation

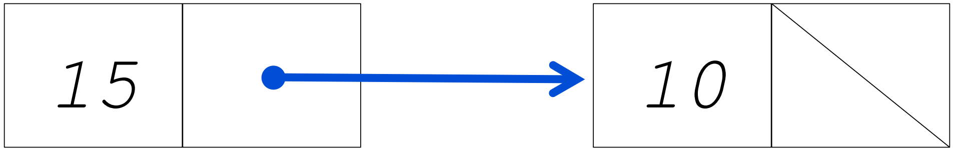
- Dynamic memory allocation
  - **Obtain** and **release** memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - If no memory available,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

Dynamic memory allocation

- Dynamic memory allocation
  - **Obtain** and **release** memory during execution
- `malloc`
  - Takes number of bytes to allocate
    - Use `sizeof` to determine the size of an object
  - Returns pointer of type `void *`
    - A `void *` pointer may be assigned to any pointer
    - 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
    - `free (newPtr);`

Dynamic memory allocation

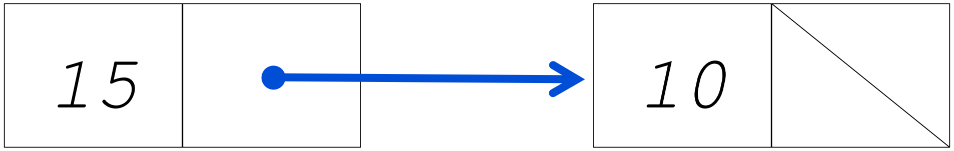
Two self-referential structures linked together



```
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;
}
```

Dynamic memory allocation

Two self-referential structures linked together

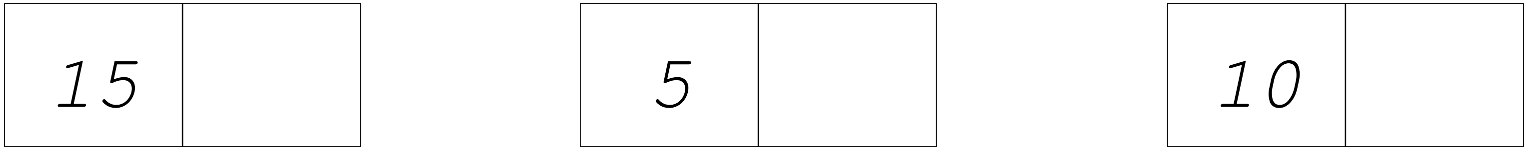


```
int main() {
  // allocates memory
  nodet *node1 = (nodet *)malloc(sizeof(nodet));
  nodet *node2 = (nodet *)malloc(sizeof(nodet));
  node1->data = 15;
  node2->data = 10;
  // link node1 to node2
  node1->nextPtr = node2;
  node2->nextPtr = NULL;
  // Deallocates memory allocated by malloc
  free(node1);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

If there exists no memory available, then `malloc` returns `NULL`

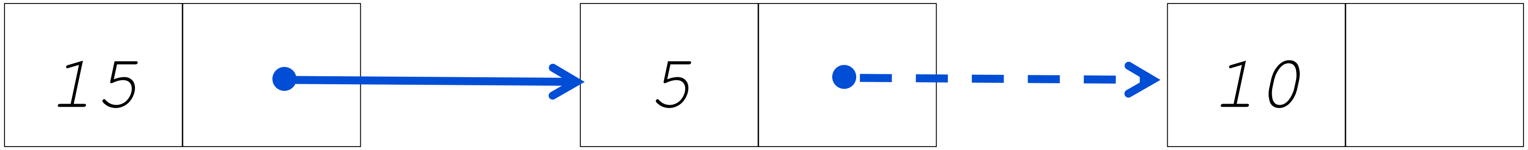
Linked lists properties

- Linked list
  - Linear collection of self-referential class objects, called nodes



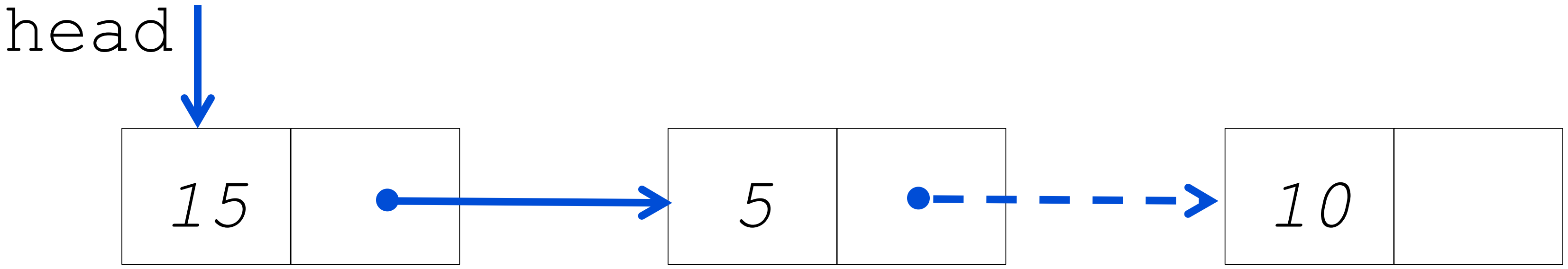
Linked lists properties

- Linked list
  - Linear collection of self-referential class objects, called nodes
  - Connected by pointer links



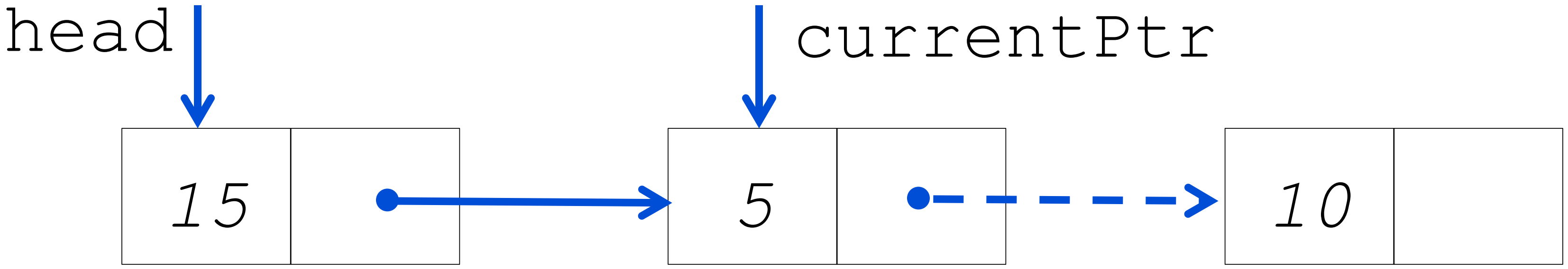
Linked lists properties

- 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



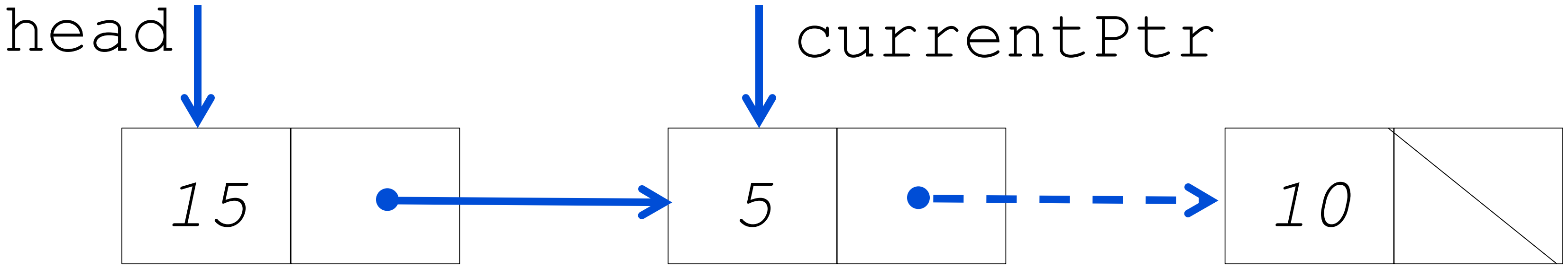
Linked lists properties

- 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



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

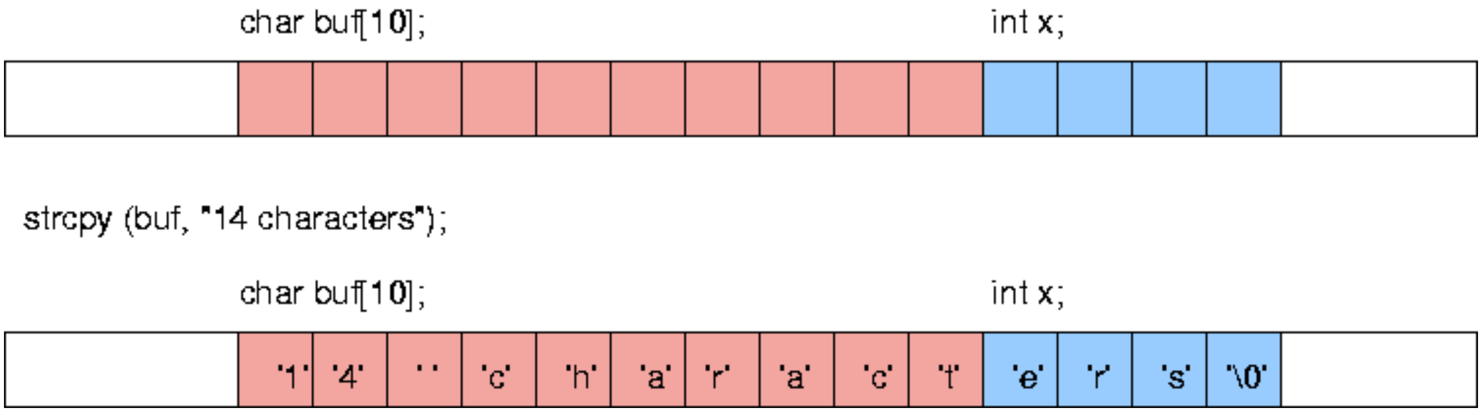
- Linked lists are **dynamic**, so the length of a list can **increase** or **decrease** as necessary

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?

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



```
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 usually are **not stored contiguously in memory**

Linked lists properties

- Linked-list nodes usually are **not stored contiguously in memory**
    - How are arrays stored in memory? What would be the advantage here?

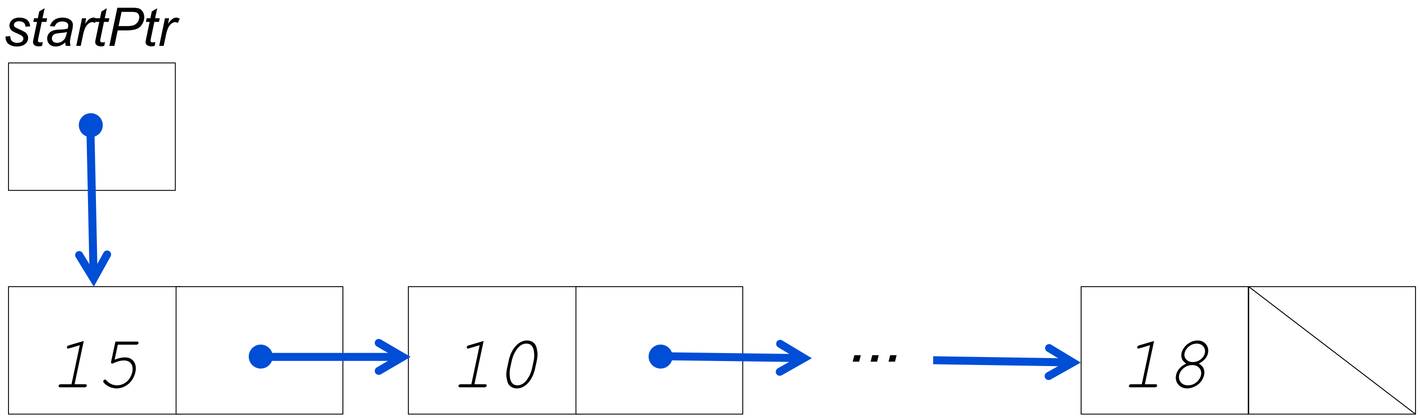
Linked lists properties

- Linked-list nodes usually are **not stored contiguously in memory**
    - How are arrays 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

Linked lists properties

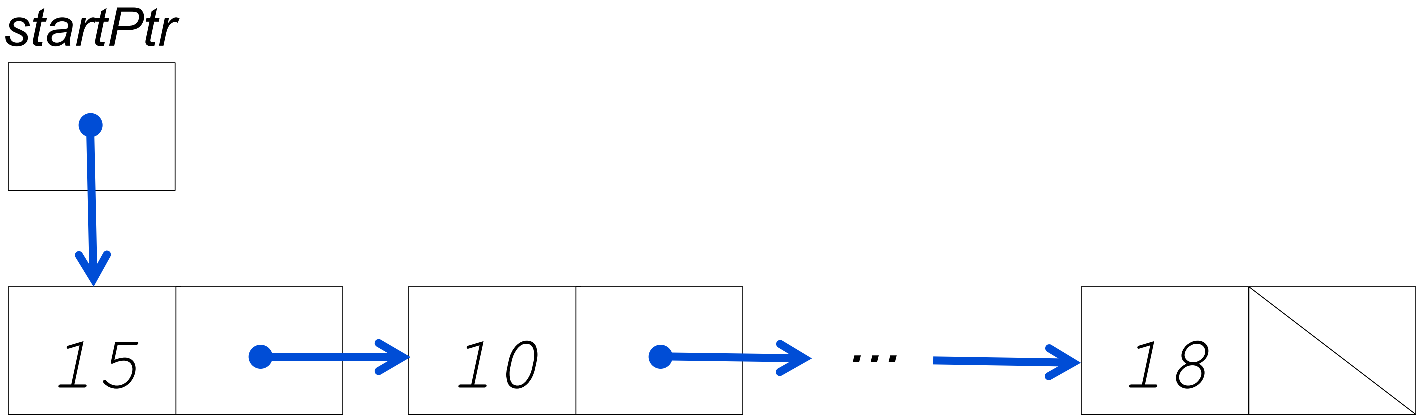
- Linked-list nodes usually are **not stored contiguously in memory**
    - How are arrays stored in memory? What would be the advantage here?
        - This allows **immediate access** since the address of any element can be calculated directly based on its position relative to the beginning of the 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;** 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;
}
```

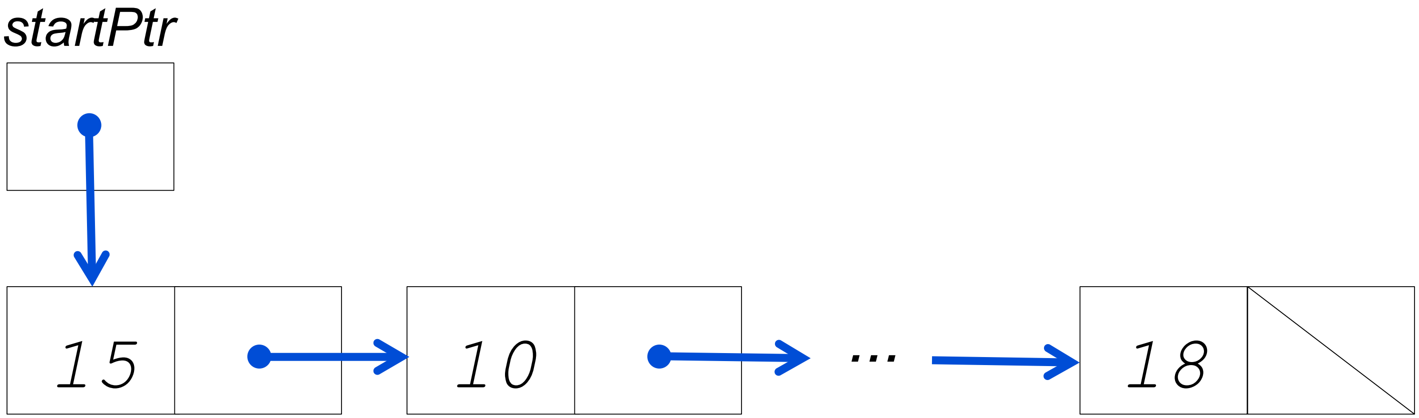
A graphical representation of a linked list



```
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 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`?

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

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`?

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)**

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`)

```c
// Fig. 12.3: fig12_03.c
// Inserting and deleting nodes in a list
#include <stdio.h>
#include <stdlib.h>

// self-referential structure
struct listNode {
   char data; // each listNode contains a character
   struct listNode *nextPtr; // pointer to next node
};

typedef struct listNode ListNode; // synonym for struct listNode
typedef ListNode *ListNodePtr; // synonym for ListNode*

// prototypes
void insert(ListNodePtr *sPtr, char value);
char delete(ListNodePtr *sPtr, char value);
int isEmpty(ListNodePtr sPtr);
void printList(ListNodePtr currentPtr);
void instructions(void);

int main(void)
{
```

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

```
   ListNodePtr startPtr = NULL; // initially there are no nodes
   char item; // char entered by user

   instructions(); // display the menu
   printf("%s", "? ");
   unsigned int choice; // user's choice
   scanf("%u", &choice);

   // loop while user does not choose 3
   while (choice != 3) {

      switch (choice) {
         case 1:
            printf("%s", "Enter a character: ");
            scanf("\n%c", &item);
            insert(&startPtr, item); // insert item in list
            printList(startPtr);
            break;
         case 2: // delete an element
            // if list is not empty
            if (!isEmpty(startPtr)) {
               printf("%s", "Enter character to be deleted: ");
               scanf("\n%c", &item);

```

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

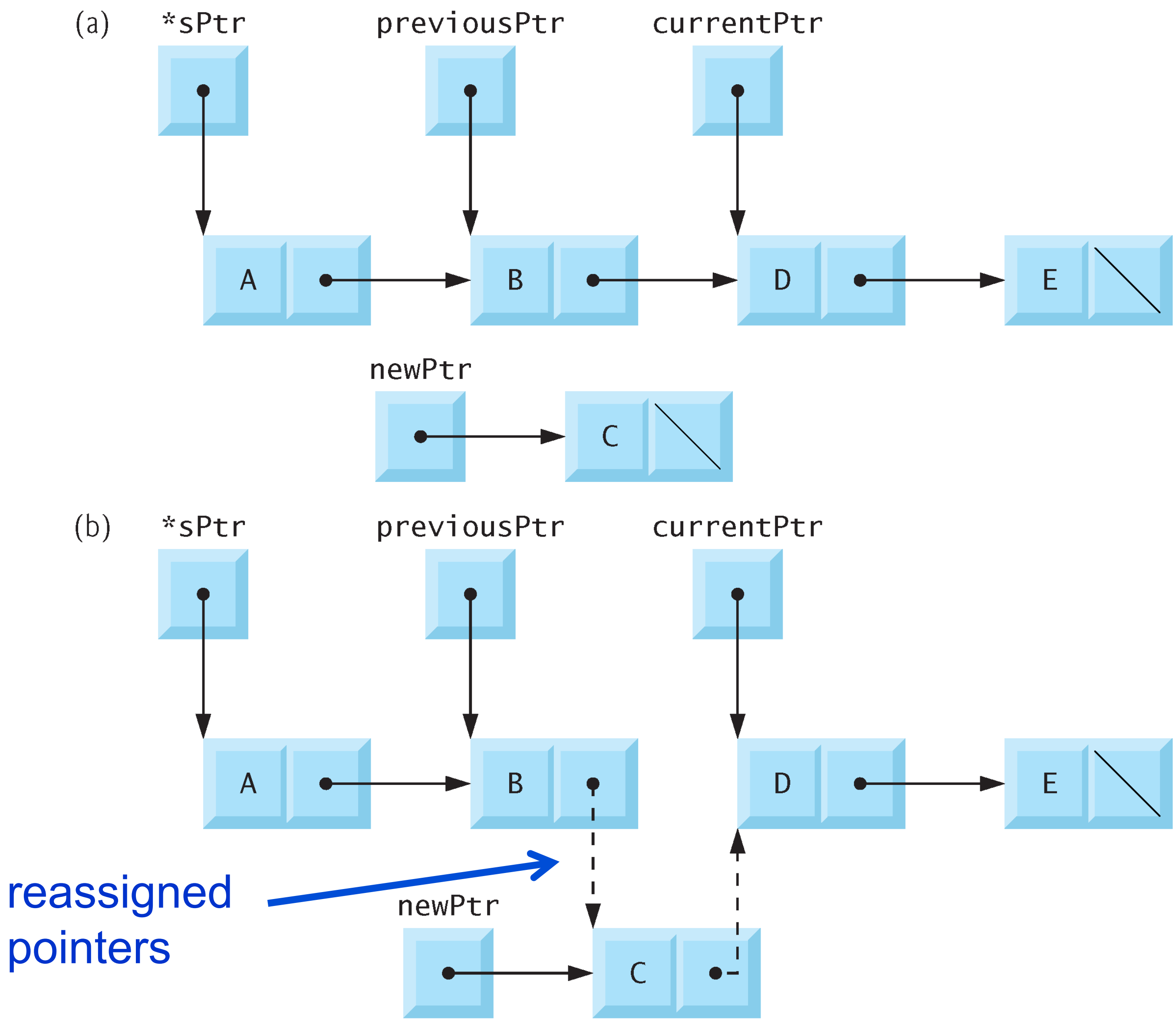
```
               // if character is found, remove it
               if (delete(&startPtr, item)) { // remove item
                  printf("%c deleted.\n", item);
                  printList(startPtr);
               }
               else {
                  printf("%c not found.\n\n", item);
               }
            }
            else {
               puts("List is empty.\n");
            }

            break;
         default:
            puts("Invalid choice.\n");
            instructions();
            break;
      }

      printf("%s", "? ");
      scanf("%u", &choice);
   }

```

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



reassigned pointers

Inserting a node in order in a list

```
   puts("End of run.");
}

// display program instructions to user
void instructions(void)
{
   puts("Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.");
}

// insert a new value into the list in sorted order
void insert(ListNodePtr *sPtr, char value)
{
   ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node

   if (newPtr != NULL) { // is space available?
      newPtr->data = value; // place value in node
      newPtr->nextPtr = NULL; // node does not link to another node

      ListNodePtr previousPtr = NULL;
      ListNodePtr currentPtr = *sPtr;

```

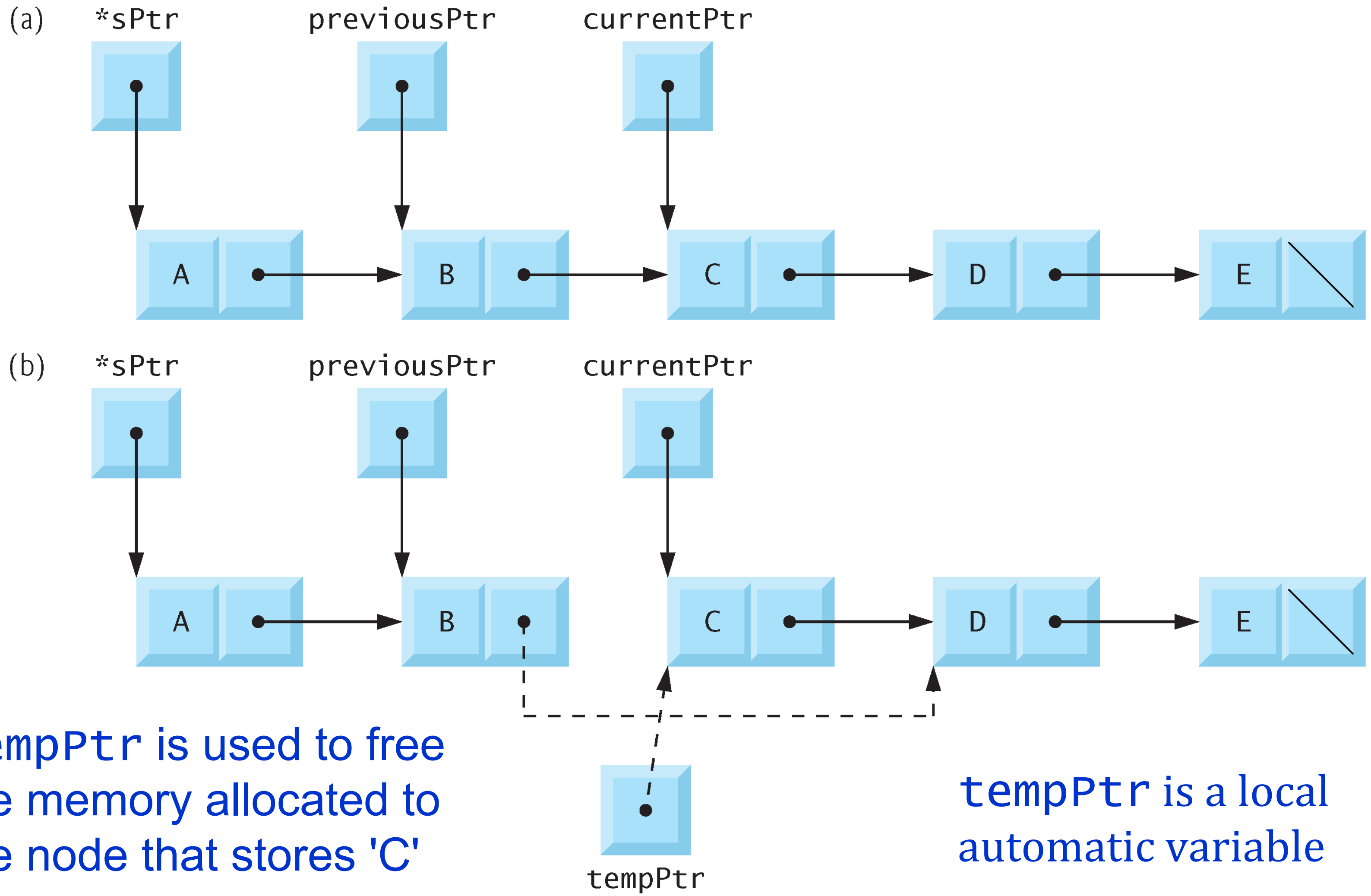
Inserting and deleting nodes in a list (Part 4 of 8)

```
      // loop to find the correct location in the list
      while (currentPtr != NULL && value > currentPtr->data) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }

      // insert new node at beginning of list
      if (previousPtr == NULL) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else { // insert new node between previousPtr and currentPtr
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else {
      printf("%c not inserted. No memory available.\n", value);
   }
}

```

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



`tempPtr` is used to free the memory allocated to the node that stores 'C'

`tempPtr` is a local automatic variable

Deleting a node from a list

```c
// delete a list element
char delete(ListNodePtr *sPtr, char value)
{
   // delete first node if a match is found
   if (value == (*sPtr)->data) {
      ListNodePtr tempPtr = *sPtr; // hold onto node being removed
      *sPtr = (*sPtr)->nextPtr; // de-thread the node
      free(tempPtr); // free the de-threaded node
      return value;
   }
   else {
      ListNodePtr previousPtr = *sPtr;
      ListNodePtr currentPtr = (*sPtr)->nextPtr;

      // loop to find the correct location in the list
      while (currentPtr != NULL && currentPtr->data != value) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }

```

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

```
      // delete node at currentPtr
      if (currentPtr != NULL) {
         ListNodePtr tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free(tempPtr);
         return value;
      }
   }

   return '\0';
}

// return 1 if the list is empty, 0 otherwise
int isEmpty(ListNodePtr sPtr)
{
   return sPtr == NULL;
}

```

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

```c
// print the list
void printList(ListNodePtr currentPtr)
{
   // if list is empty
   if (isEmpty(currentPtr)) {
      puts("List is empty.\n");
   }
   else {
      puts("The list is:");

      // while not the end of the list
      while (currentPtr != NULL) {
         printf("%c --> ", currentPtr->data);
         currentPtr = currentPtr->nextPtr;
      }

      puts("NULL\n");
   }
}
```

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	RUNTIME (Big-O)
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	

```c
   puts("End of run.");
}

// display program instructions to user
void instructions(void)
{
   puts("Enter your choice:\n"
      "   1 to insert an element into the list.\n"
      "   2 to delete an element from the list.\n"
      "   3 to end.");
}

// insert a new value into the list in sorted order
void insert(ListNodePtr *sPtr, char value)
{
   ListNodePtr newPtr = malloc(sizeof(ListNode)); // create node

   if (newPtr != NULL) { // is space available?
      newPtr->data = value; // place value in node
      newPtr->nextPtr = NULL; // node does not link to another node

      ListNodePtr previousPtr = NULL;
      ListNodePtr currentPtr = *sPtr;

```

O(1)

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

```
      // loop to find the correct location in the list
      while (currentPtr != NULL && value > currentPtr->data) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }

      // insert new node at beginning of list
      if (previousPtr == NULL) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else { // insert new node between previousPtr and currentPtr
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
   else {
      printf("%c not inserted. No memory available.\n", value);
   }
}
```

O(n)

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

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

```c
      // loop to find the correct location in the list
      while (currentPtr != NULL && value > currentPtr->data) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }
```

O(n)

```c

      // insert new node at beginning of list
      if (previousPtr == NULL) {
         newPtr->nextPtr = *sPtr;
         *sPtr = newPtr;
      }
      else { // insert new node between previousPtr and currentPtr
         previousPtr->nextPtr = newPtr;
         newPtr->nextPtr = currentPtr;
      }
   }
```

O(1)

```c
   else {
      printf("%c not inserted. No memory available.\n", value);
   }
}

```

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

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

```
// delete a list element
char delete(ListNodePtr *sPtr, char value)
{
   // delete first node if a match is found
   if (value == (*sPtr)->data) {
      ListNodePtr tempPtr = *sPtr; // hold onto node being removed
      *sPtr = (*sPtr)->nextPtr; // de-thread the node
      free(tempPtr); // free the de-threaded node
      return value;
   }
   else {
      ListNodePtr previousPtr = *sPtr;
      ListNodePtr currentPtr = (*sPtr)->nextPtr;

      // loop to find the correct location in the list
      while (currentPtr != NULL && currentPtr->data != value) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }

```

O(1)

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

```c
// delete a list element
char delete(ListNodePtr *sPtr, char value)
{
   // delete first node if a match is found
   if (value == (*sPtr)->data) {
      ListNodePtr tempPtr = *sPtr; // hold onto node being removed
      *sPtr = (*sPtr)->nextPtr; // de-thread the node
      free(tempPtr); // free the de-threaded node
      return value;
   }
   else {
      ListNodePtr previousPtr = *sPtr;
      ListNodePtr currentPtr = (*sPtr)->nextPtr;

      // loop to find the correct location in the list
      while (currentPtr != NULL && currentPtr->data != value) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }

```

O(1)

O(1)

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

```c
// delete a list element
char delete(ListNodePtr *sPtr, char value)
{
   // delete first node if a match is found
   if (value == (*sPtr)->data) {
      ListNodePtr tempPtr = *sPtr; // hold onto node being removed
      *sPtr = (*sPtr)->nextPtr; // de-thread the node
      free(tempPtr); // free the de-threaded node
      return value;
   }
   else {
      ListNodePtr previousPtr = *sPtr;
      ListNodePtr currentPtr = (*sPtr)->nextPtr;

      // loop to find the correct location in the list
      while (currentPtr != NULL && currentPtr->data != value) {
         previousPtr = currentPtr; // walk to ...
         currentPtr = currentPtr->nextPtr; // ... next node
      }

```

O(1)

O(1)

O(n)

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

```c
      // delete node at currentPtr
      if (currentPtr != NULL) {
         ListNodePtr tempPtr = currentPtr;
         previousPtr->nextPtr = currentPtr->nextPtr;
         free(tempPtr);
         return value;
      }
   }

   return '\0';
}

// return 1 if the list is empty, 0 otherwise
int isEmpty(ListNodePtr sPtr)
{
   return sPtr == NULL;
}

```

O(1)

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

OPERATION	RUNTIME (Big-O)
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 behaviour**

Do not access freed memory (MEM30-C)

- Evaluating a **pointer** into memory that has been **deallocated** by a memory management function is **undefined behaviour**
- **Pointers to memory** that has been **deallocated** are called **dangling pointers**
  - Accessing a dangling pointer can result in **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 behaviour**
- **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 behaviour**

Noncompliant Code Example

- Illustrates 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**

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**

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 behaviour**
  - the pointer value is indeterminate and might be a trap representation

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 behaviour**
  - 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**

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**

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.”

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()*

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

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

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
  - Inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer

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
  - 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

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

Data structure alignment

- **Data structure alignment** is concerned with the approach data is arranged and accessed in computer memory
  - **data alignment**, **data structure padding**, and **packing**
- Modern hardware **reads** and **writes** to memory most efficiently if the **data is aligned**
  - The data's memory address must be a multiple of the data size
    - In a 32-bit architecture, the data may be aligned if the data is stored in four consecutive bytes, and the first byte lies on a 4-byte boundary

Example of Data structure alignment

- `MixedData` is a structure with members of various types, totaling 8 bytes before compilation:

```
struct MixedData
{
   char Var1;
   short Var2;
   int Var3;
   char Var4;
};
```

After compilation in 32-bit x86 machine

```
struct MixedData
{
   char Var1;
   char Padding1[1];
   short Var2;
   int Var3;
   char Var4;
   char Padding2[3];
};
```

- `MixedData` is supplemented with **padding bytes** to ensure a **proper alignment** of its members

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()*

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 more stringent than typical alignment requirements**

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 more stringent than typical alignment requirements**
- The C standard requires that a pointer returned by *realloc()* be suitably aligned so that it may be assigned to a pointer to any 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

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

realloc() may not preserve the stricter alignment of the original object.

```
#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

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.

Summary

(Secure C Programming)

- Pointers should not be left uninitialized

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

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

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

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”

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

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