Secure Coding C++

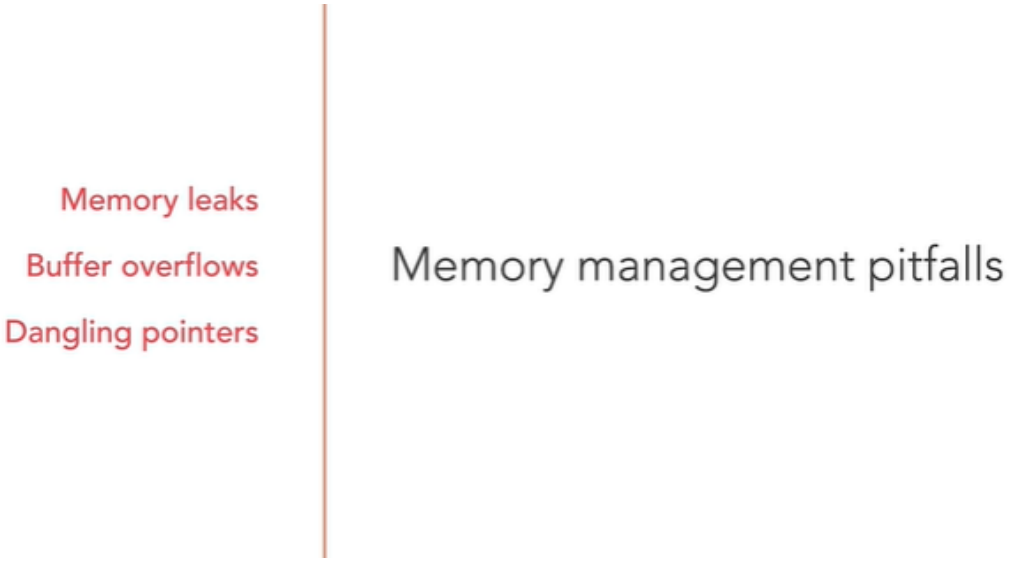
Why?
Example:- In a flight a professional game developer starts playing video games. He was too good. While playing he passed previous levels, created a new record and suddenly the whole entertainment system of flight crashed. It was due to a small vulnerability.

We need secure coding to create robust systems.

1. Balancing Freedom and Responsibility:-

1.1 The Hazards of Manual Memory Management:-

- One of the most powerful yet potentially dangerous features of C and C++ is the ability to manage memory manually.
- C and C++ do not have automatic garbage collection like some higher-level languages. So it's up to the programmer to correctly allocate and deallocate memory.
- If not handled correctly, this can lead to vulnerabilities such as
    - memory leaks,
    - buffer overflows,
    - dangling pointers.



- **Memory leaks** occur when dynamically allocated memory is not properly released, resulting in the loss of available memory.
- this can lead to
    - reduced system performance,
    - resource exhaustion,
    - even denial of service attacks in certain scenarios.

- Buffer overflows occur when a program writes more data into a buffer than it can hold, leading to memory corruption and potential code execution exploits.
    - Attackers can exploit these vulnerabilities to overwrite critical data, inject malicious code, or gain unauthorised access to systems.
- Dangling pointers occurs when a pointer is left pointing to memory that has been deallocated or freed, accessing that memory can result in undefined behaviour.
    - This can lead to crashes,
    - data, corruption and
    - potential security vulnerabilities.

- To mitigate these risks, it's crucial to follow secure coding practices when dealing with memory in C and C++.
- This includes properly allocating and deallocating memory, avoiding buffer overflows through proper bounds checking and use of safe string functions, and ensuring
  that pointers are always valid and properly managed.

SMART Pointers:-

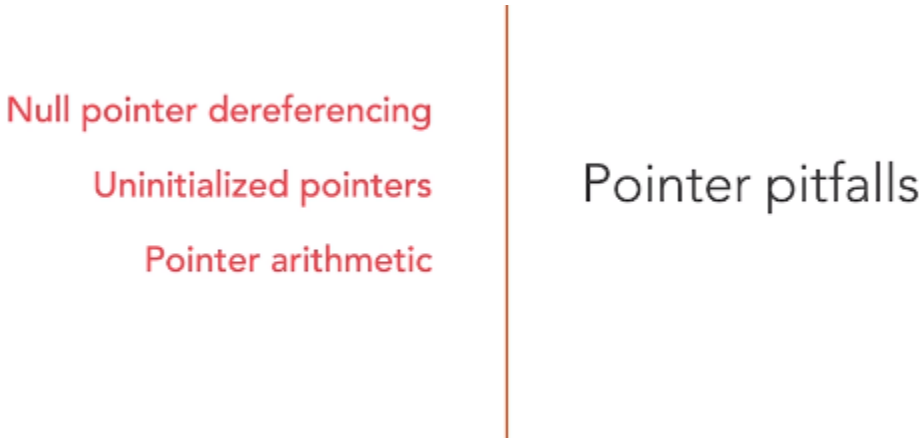
- Smart pointers in C++ can help automate memory management and reduce the risk of memory-related vulnerabilities.
- Smart pointers automatically manage memory, deallocating it when it's no longer in use.
- They help prevent memory leaks and make code safer and easier to maintain.

- However, even with smart pointers, programmers need to be cautious about how they use memory to avoid potential security issues.

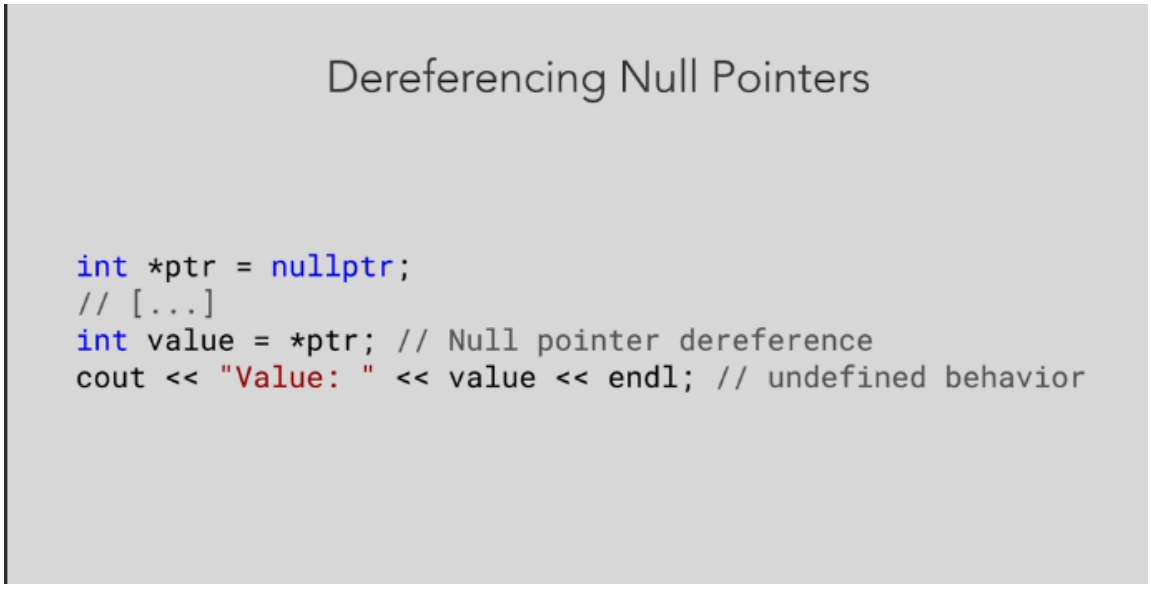
It's also important to note that these features are not available in C, making memory management more challenging in this language.

1.2 Pointer Pitfalls

- If Pointers are not used properly, pointers can lead to many problems.
    - Null or invalid pointer dereferencing
    - uninitialized or dangling pointers
    - pointer arithmetic

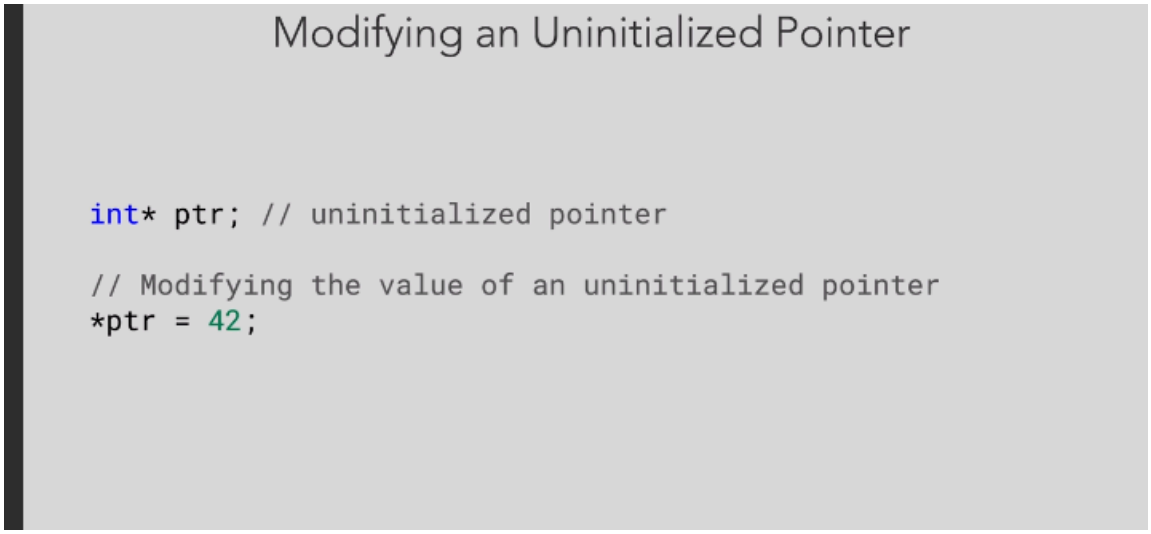


- Null pointer dereferencing occurs when a program tries to access or modify the memory pointed to by a null pointer.
- Since null pointers do not point to valid memory locations, attempting to access or modify their contents can result in crashes or system instability.
- Attackers can exploit null pointer dereferences to cause denial of service or privilege escalation vulnerabilities.

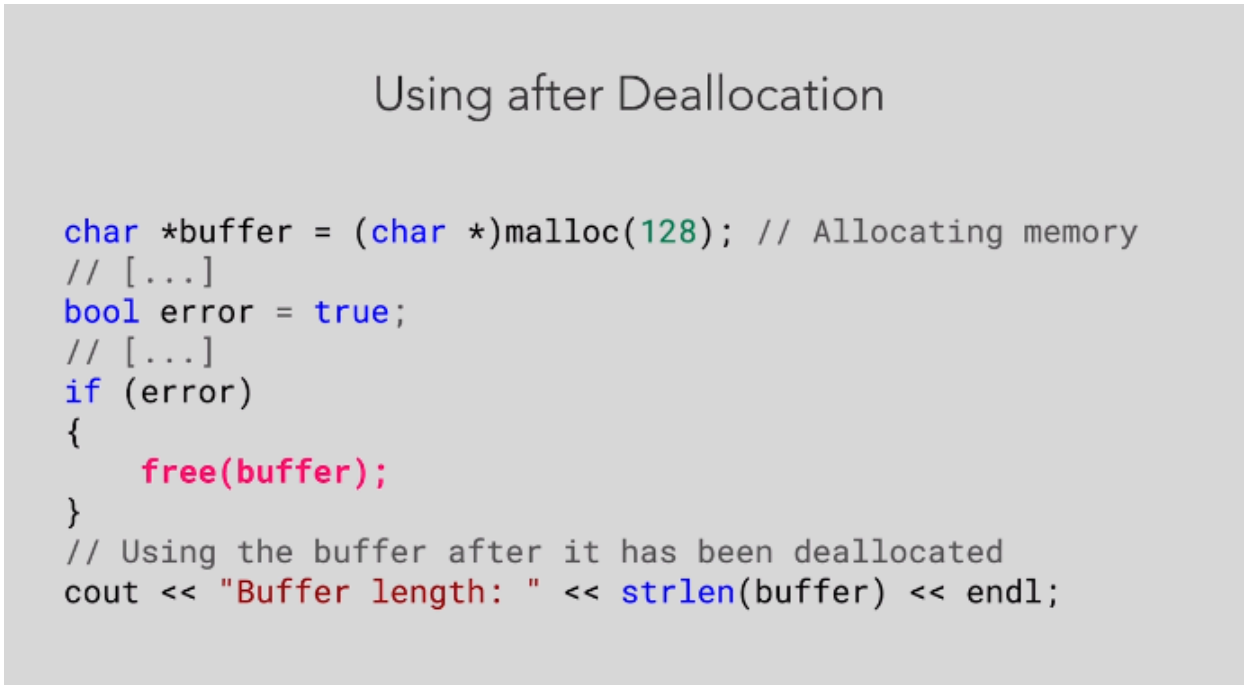


- Another issue arises when pointers are used without proper initialization.
- If a pointer is not initialised to a valid memory address, accessing or modifying its contents can lead to reading or writing data from unintended locations in memory.

- This can result in corrupted data, crashes or even security vulnerabilities if sensitive information is exposed.

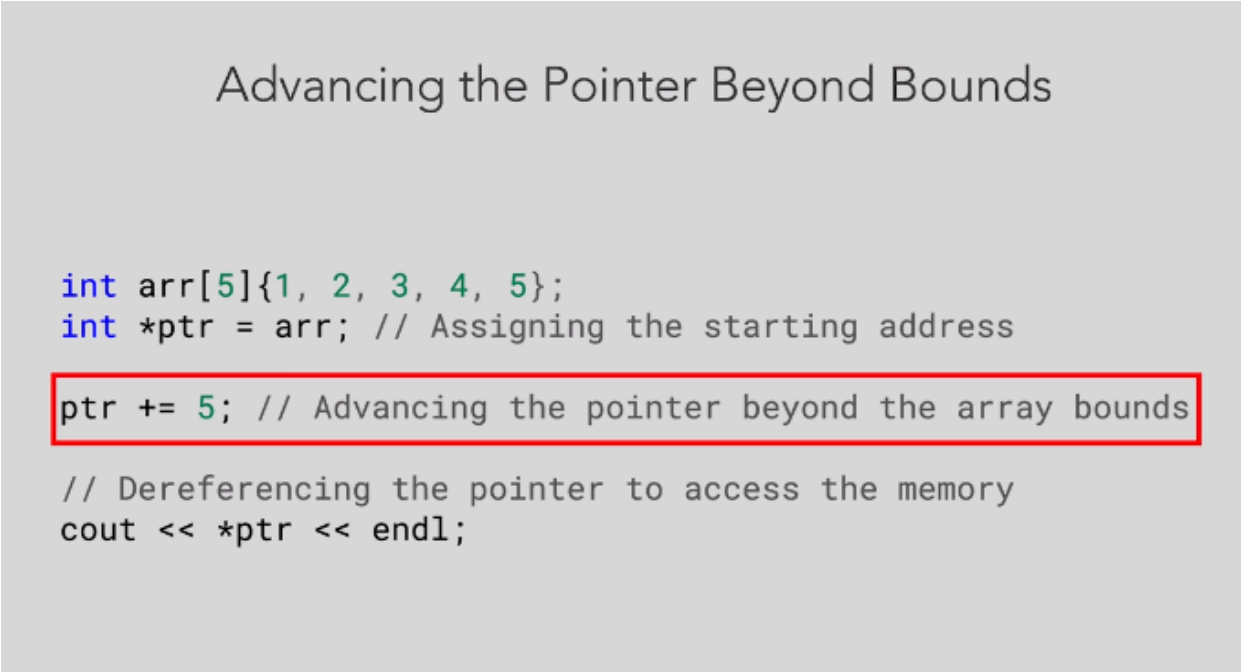


- Dangling pointers are yet another concern.
- A dangling pointer refers to a pointer that still holds a memory address, but the memory it points to has been deallocated or freed.
- Attackers can exploit dangling pointers to overwrite critical data or execute arbitrary code.



- Pointer arithmetic involves performing operations such as addition or subtraction on pointers to navigate through memory.
- While pointer arithmetic can be useful in certain scenarios, it's easy to make mistakes.

- For instance, if pointer arithmetic is used to iterate through an array or buffer, a mistake in the calculations can lead to accessing or modifying memory outside the intended boundaries.
- This can result in corrupted data, crashes, or even security vulnerabilities if an attacker can control the input and exploit the buffer overflow to overwrite critical data or execute arbitrary code.



To minimise these risks, here are some best practices for working with pointers:
- Always initialise your pointers, preferably when you declare them.

Always Initialize Pointers

```
int* ptr = new int;
// Modifying the value of an uninitialized pointer
*ptr = 42;

delete ptr;
```

- Always check pointer values before use. Before dereferencing a pointer, ensure that it's neither null nor pointing to freed memory.

Check for Null Pointers

```
if (ptr == nullptr)
{
    // Pointer is nullptr
}
```

- Be cautious with dynamic memory allocation if you're using new and delete or malloc and free in C.

Avoid Pointer Arithmetic

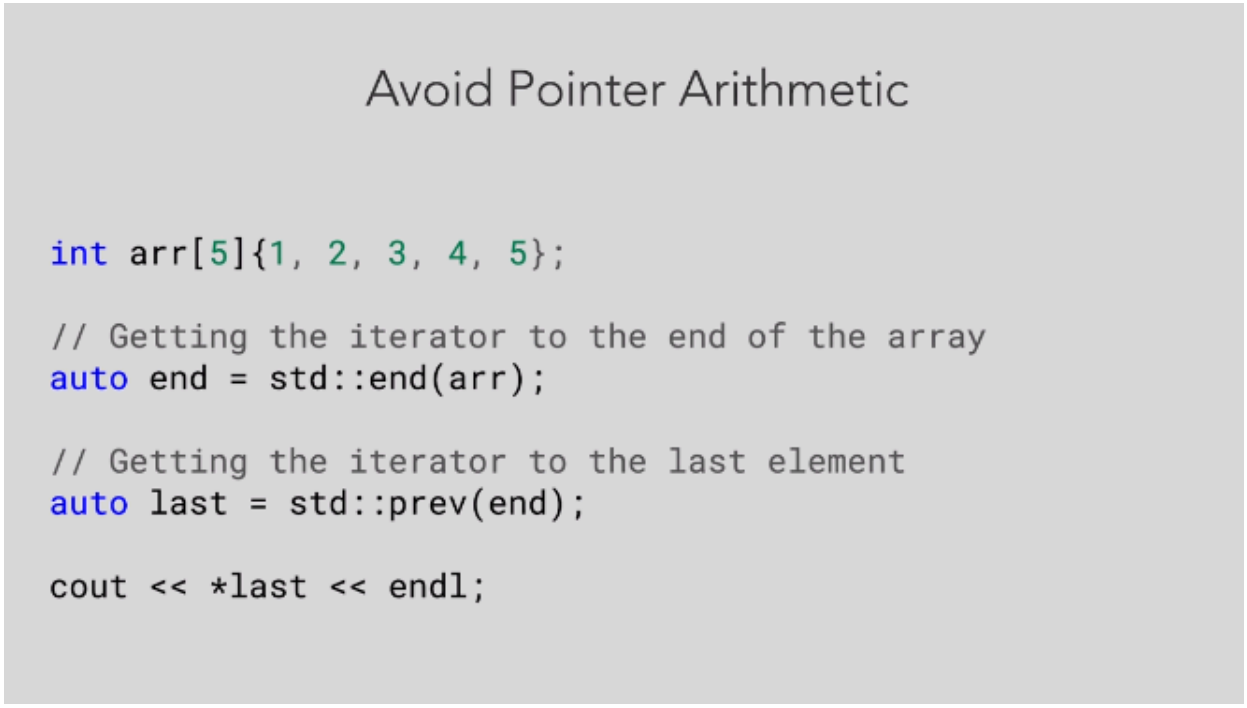
```
int arr[5]{1, 2, 3, 4, 5};
int *ptr = arr; // Assigning the starting address

ptr += 5; // Advancing the pointer beyond the array bounds

// Dereferencing the pointer to access the memory
cout << *ptr << endl;
```

- Make sure that every allocation with new or malloc is paired with the corresponding call to delete or free.

Release Allocated Memory

```
char *buffer = (char *)malloc(128); // Allocating memory
// [...]
free(buffer);

int* ptr = new int;
// [...]
delete ptr;
```

- Finally, avoid using pointer arithmetic whenever possible.



Instead, prefer higher level abstractions or container classes that handle memory management and boundary checking for you.

1.3 The Double-Edged Sword: Low-Level System Access

- C and C++ provide low-level access to system resources, which is part of why they are so powerful and flexible.
- Yet this low-level access can also lead to security issues if not properly managed.
- For instance, hackers could gain access to privileged system resources. This vulnerability is commonly known as a **privilege escalation attack.**
- To mitigate these risks, it is important to adhere to certain principles when working with low level access in C and C++.

- Two key principles that can guide developers in managing low level access effectively
  and securely:
  - The principle of least privilege and
  - The principle of fail-safe defaults.

- The principle of least privilege suggests that a process should be granted only those privileges that are essential to its function.

- In the context of programming, it means that code should have the minimum privileges necessary to perform its intended tasks.

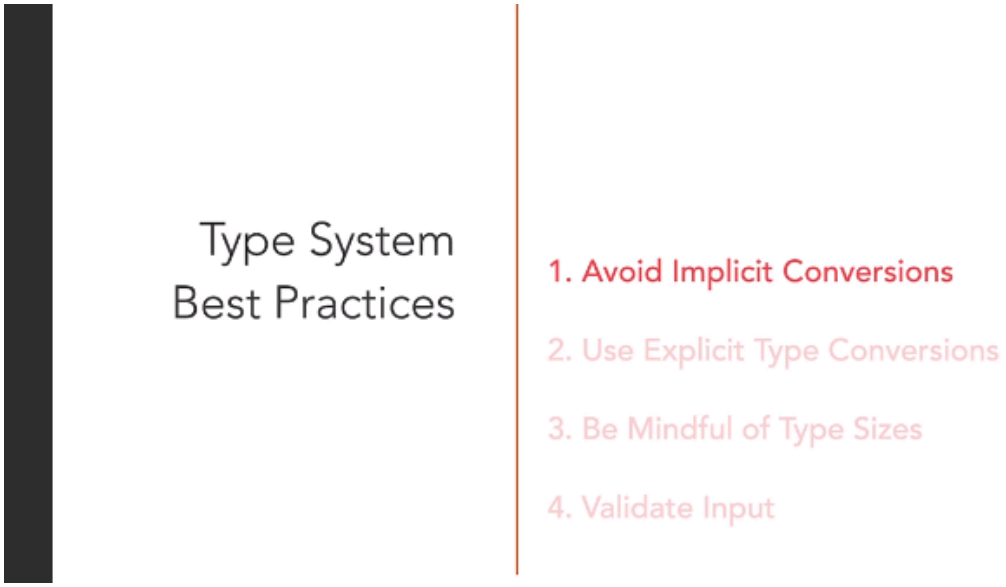
- The principle of fail-safe default states that code should not have access to system resources unless such access is explicitly granted and justified. By defaulting to a restricted access model,access to system resources is denied unless explicitly authorised.

- As a consequence, the chances of unauthorised access or privilege escalation are minimised.
- Developers can maintain a balance between functionality and security by following the principle of least privilege and the principle of fail-safe defaults.

1.4 Risky Type Conversions

- Even though C and C++ use a static type system, it still allows risky operations such as implicit conversions and casting between incompatible types.
- Incorrect use of these features can lead to undefined behaviour, which, as we already know, is a synonym for security risk.
- To minimise these possible hazards, it's important to follow best practices when working with the type system.
- Here are some guidelines to keep in mind.

  Avoid implicit conversions.



- Implicit conversions, where the compiler automatically converts one type to another, can lead to unexpected behavior and vulnerabilities.

- Be explicit in your code and avoid relying on implicit conversions.

This helps ensure that the types are compatible and prevents unintended consequences.

Now, in this particular case, an implicit transformation happens from a double type to an integer

type, leading to the loss of precision.

implicit-conversion.cpp

```
3
4   int main()
5   {
6       double x = 3.14;
7       int y = x; // Implicit conversion from double to int
8
9       cout << "Value of x: " << x << " y: " << y << endl;
10  }
11
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL C/C++: clang++ build active file

```
*  Executing task: C/C++: clang++ build active file

Starting build...
/usr/bin/clang++ -std=c++20 -fcolor-diagnostics -fansi-escape-codes -g "/Users/knyisztor/! Projects/2. Courses/Udemy/02. Sec
ure Coding in C:C++/src/Ch02-Balancing-freedom-and-responsibility/implicit-conversion.cpp" -o "/Users/knyisztor/! Projects/2
. Courses/Udemy/02. Secure Coding in C:C++/src/Ch02-Balancing-freedom-and-responsibility/implicit-conversion"

Build finished successfully.
*  Terminal will be reused by tasks, press any key to close it.
```

Use explicit type conversions.

Type System Best Practices

1. Avoid Implicit Conversions
2. Use Explicit Type Conversions
3. Be Mindful of Type Sizes
4. Validate Input

- When type conversions are necessary, it's crucial to use explicit type conversion operators or functions.
- Avoid using generic c-style casts as they can bypass compiler checks and lead to type related vulnerabilities.

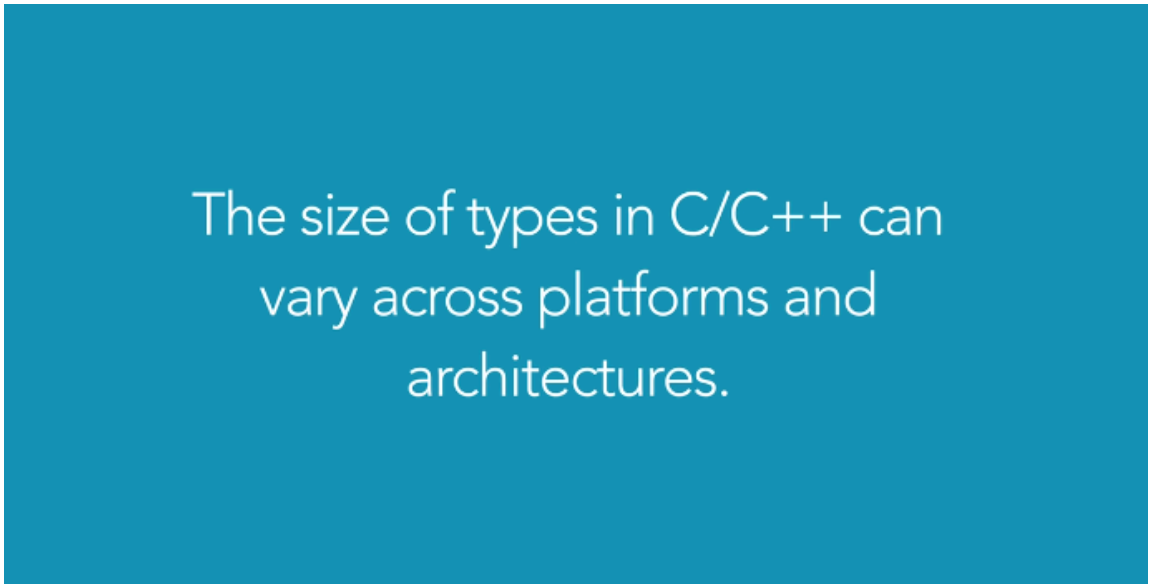
- Use designated cast operators such as static_cast, reinterpret_cast or dynamic_cast, depending on the specific conversion needs.

implicit-conversion.cpp

```
3
4   int main()
5   {
6       double x = 3.14;
7       int y = static_cast<int>(x); // explicit type c
8
9       cout << "Value of x: " << x << " y: " << y << endl;
10  }
11
```

- **Be mindful of type sizes.**

- The size of types in C and C++ can vary across different platforms and architectures.



- This can lead to buffer overflows, data truncation, and memory corruption if not handled properly.
- Always ensure that you allocate sufficient memory and properly handle type sizes to avoid potential security vulnerabilities.

In this example, we have a large number that exceeds the range of an int16_t type, which can hold values
from -32,768 to 32,767.

type-sizes.cpp

```cpp
#include <iostream>
#include <cstdint>
using namespace std;

int main()
{
    int32_t largeNumber = 1000000000; // A large number that doesn't fit in an int

    int16_t smallNumber = static_cast<int16_t>(largeNumber); // Narrowing conversion

    cout << "Large Number: " << largeNumber << endl;
    cout << "Small Number: " << smallNumber << endl;

    return 0;
}
```

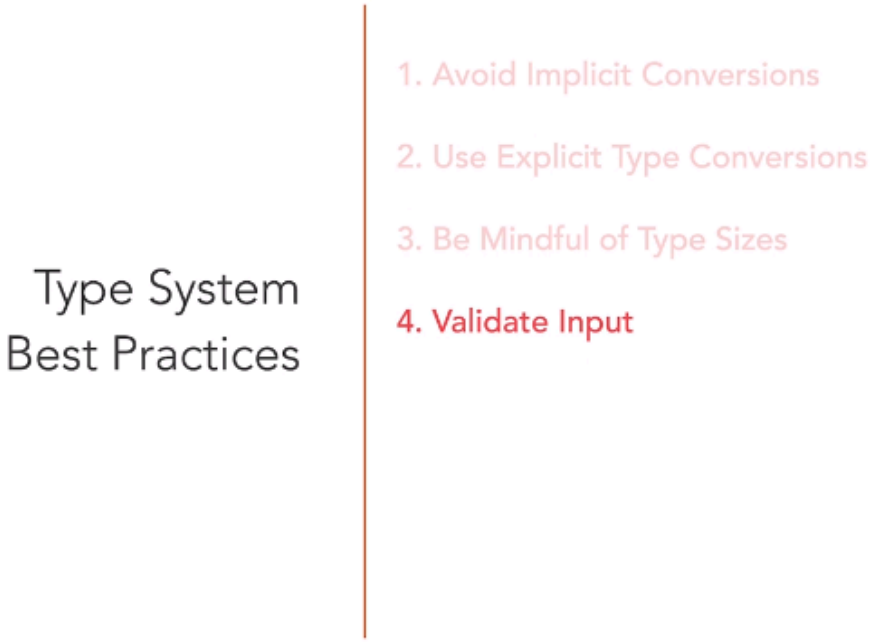
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  Filter (e.g. text, !exclude)
Loaded '/usr/lib/system/libxpc.dylib'. Symbols loaded.
Loaded '/usr/lib/libobjc.A.dylib'. Symbols loaded.
Loaded '/usr/lib/liboah.dylib'. Symbols loaded.
Large Number: 1000000000
Small Number: -13824
The program '/Users/knyisztor/! Projects/2. Courses/Udemy/02. Secure Coding in C:C++/src/Ch02-Balancing-freedom-and-responsibility/type-sizes' has exited with code 0 (0x00000000).
```

We attempt to perform a narrowing conversion by assigning largeNumber to smallNumber, explicitly casting it to int16_t using the static_cast operator. However, due to the difference in type sizes, the value of largeNumber cannot be accurately represented in an int16_t.

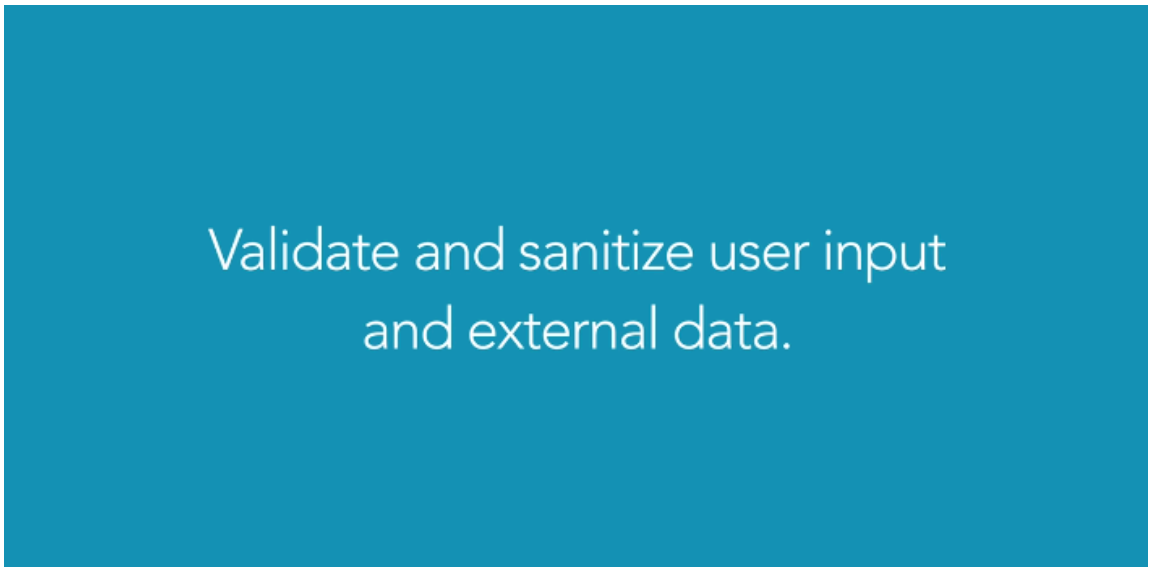
As a result, the conversion will cause truncation or overflow, leading to potential loss of data and incorrect results.

As you can see, it's essential to choose appropriate data types that can accommodate the range of values you expect to handle.

Validate input when working with user input or external data.



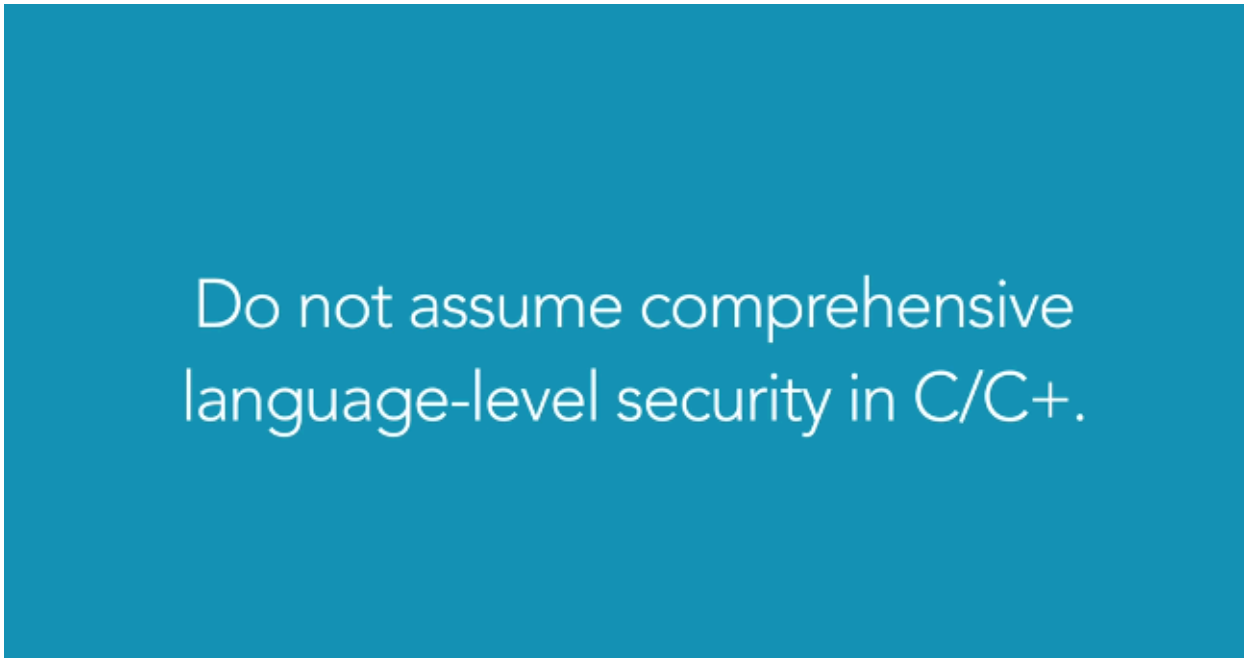
- Validate and sanitise the input to prevent unexpected type related issues.



- Do not assume the input will always conform to the expected type and perform appropriate checks and validation to ensure the integrity and safety of the program.

1.5 Library Landmines: Dangerous Functions

- C and C ++ come with extensive standard libraries that offer a wide range of functionality, making them powerful languages for software development.
- Developers often assume that these library functions, as well as the C and C++ programming languages themselves, protect them by handling a lot of security related tasks behind the scenes.

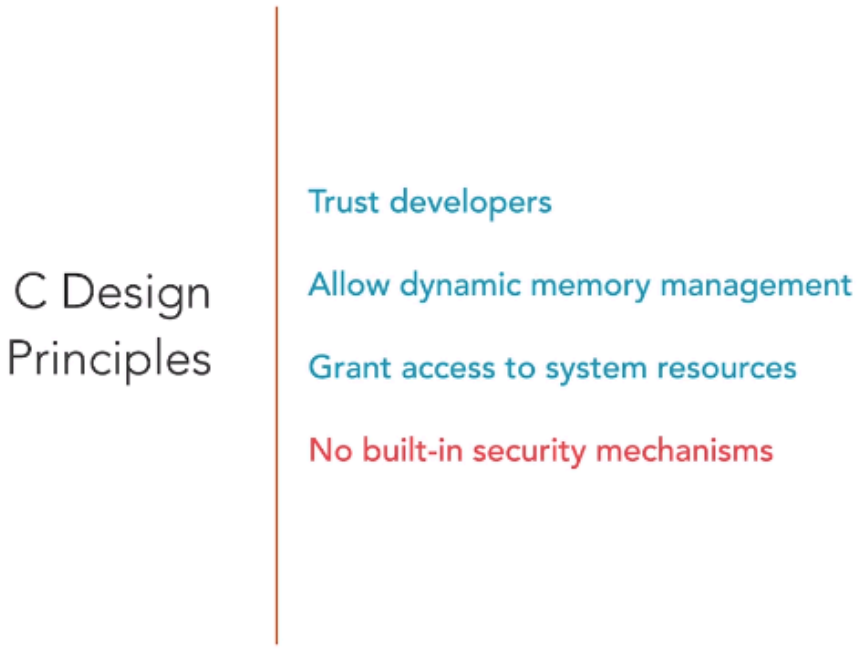


- Yet this assumption is not always true, particularly in the case of C. The C language was designed to be lightweight and simple.

- The philosophy behind it is based on the trust in the developers abilities, giving them greater control and flexibility over memory management and system resources.

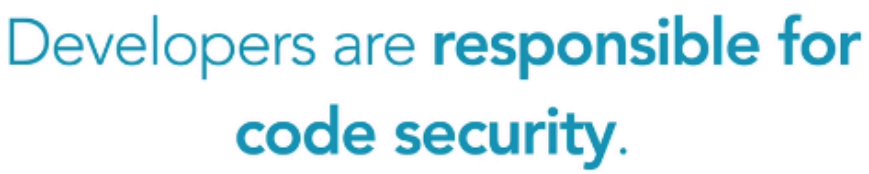
- While this level of control can be advantageous, it also means that developers bear the responsibility of implementing their own security measures.

- That's why certain library functions often do not perform bounds checking or input validation by default.

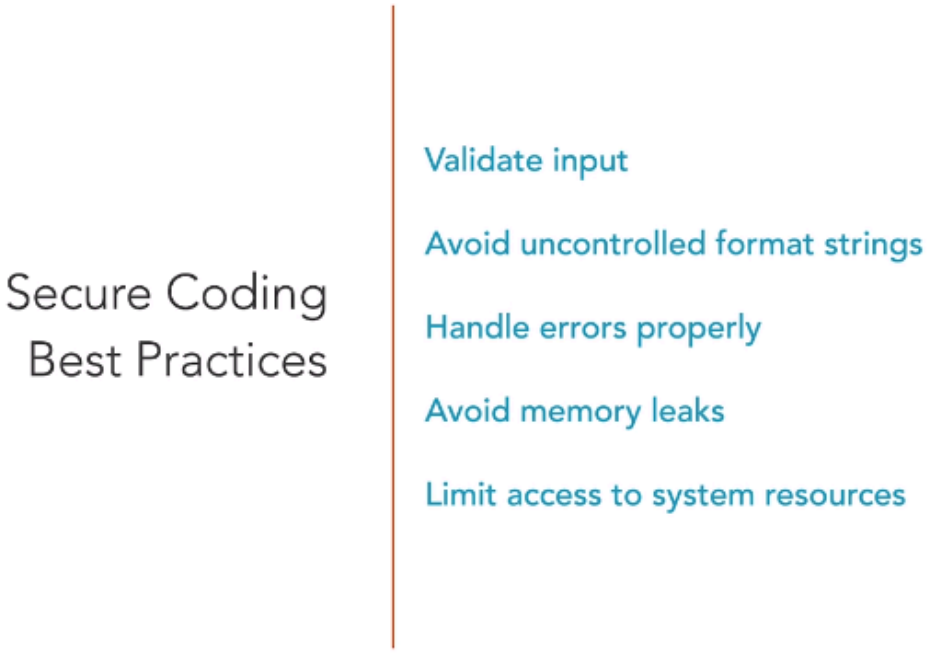


- This lack of built-in safety mechanisms leaves code vulnerable to buffer overflows and other security issues if developers do not implement additional logic to prevent them.

- In other words, developers are responsible for security, so secure coding practices are essential.



- This includes practices such as input, validation, safe string handling and other defensive programming techniques.



1.5 Tread with Caution: Legacy Code

- The longevity and widespread usage of C and C++ have resulted in a significant amount of legacy code.
- These systems, developed years or even decades ago, are still in use, requiring ongoing maintenance and occasional updates.
- The security aspect of this legacy code is particularly important.
- Here are a few things to consider:

  - Outdated, non-existent security practices.

Legacy Code:
Security Aspects

Outdated Security Practices

Many of these code bases were written before.

- Many of today's best practices for secure coding were developed or widely adopted.

- This may leave them vulnerable to attacks that were not well known or well understood at the time of their creation.

- Powering critical systems.

Legacy Code: Security Aspects

Outdated Security Practices

Used in Critical Systems

- Legacy systems often operate in critical areas such as infrastructure, telecommunications and industrial systems, where an attack could have serious implications.

- Changing threat landscape.

Legacy Code: Security Aspects

Outdated Security Practices

Used in Critical Systems

Evolving Threat Landscape

- The threat landscape evolves rapidly with new attack techniques and vulnerabilities emerging regularly.

- Without proper security considerations and updates, legacy systems become prime targets for attackers seeking to exploit known vulnerabilities.

- Lack of documentation and knowledge.

Legacy Code: Security Aspects

- Outdated Security Practices
- Used in Critical Systems
- Evolving Threat Landscape
- Lack of Documentation/Knowledge

Modifying legacy code can be challenging due to lack of documentation,

outdated style or conventions, or use of deprecated or obsolete language features.

Developers who initially worked on the code base may have moved on, resulting in a lack of information about the system.

This lack of understanding makes it challenging to identify and address security vulnerabilities effectively.

2. Common Security Flaws in C and C++ Programming

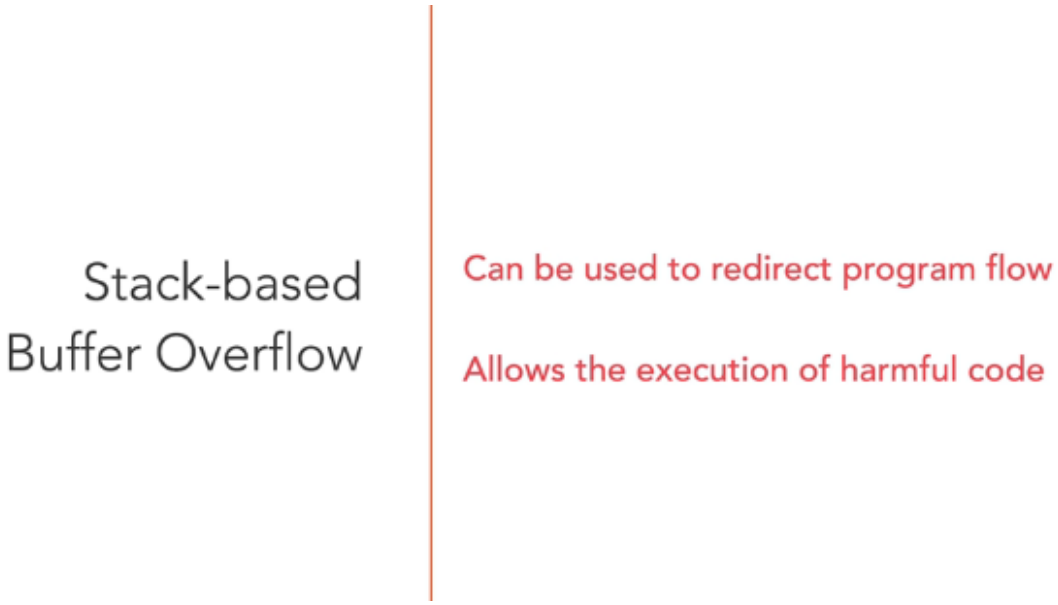
2.1 Unleashing Chaos with Stack and Heap Overflows

- A buffer overflow occurs when a program writes more data to a buffer than it was designed to hold, which can lead to crashes or exploitable security vulnerabilities.

Buffer overflow

Occurs when writing data outside of the boundaries of the allocated memory.

- Buffer overflows can happen on the stack, heap, or in a memory-mapped file.
- Stack based buffer overflows can be exploited to overwrite the return address on the stack.

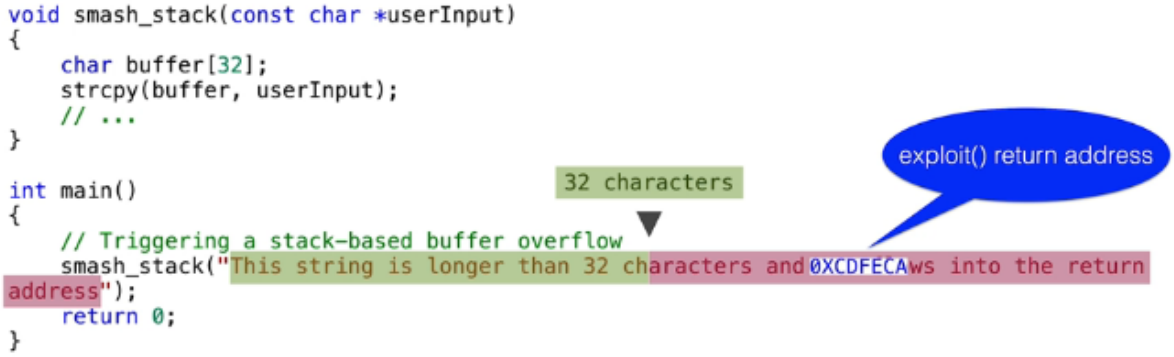


- This would enable an attacker to redirect the program flow to execute their own malicious code.
- Here's a simple example of a function vulnerable to a buffer overflow attack.

Code Vulnerable to Buffer Overflow Attacks

```
void smash_stack(const char *userInput)
{
    char buffer[32];
    strcpy(buffer, userInput);
    // ...
}
```

- Suppose an attacker provides an input string that's longer than 32 characters.
- This would override the return address stored on the stack.
- If attackers carefully craft the input string, they could override the return address with the address of their malicious code, effectively redirecting the program execution flow.



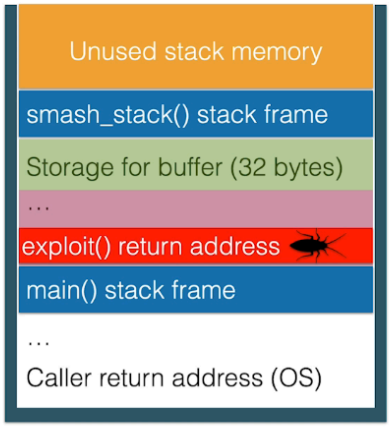
- When a function is called, a new stack frame is created on the stack.
- The stack frame contains local variables of the function and the return address of the caller.
- When the function completes its tasks, it uses the stored return address to jump back to where it was called. In our example, the smash_stack() function takes an input string and copies it into a local character array buffer of length 32.

Stack-based Buffer Overflow

Code execution

```
void smash_stack(const char *userInput)
{
    char buffer[32];
    strcpy(buffer, userInput);
    // ...
}

int main()
{
    // Triggering a stack-based buffer overflow
    smash_stack("This string is longer than 32
characters and 0XCDFECAws into the return
address");
    return 0;
}
```

Stack

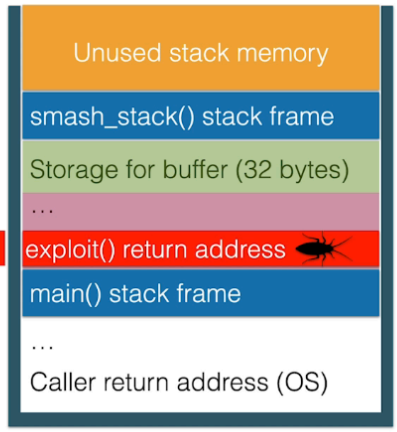


- The vulnerability here is that strcpy() does not check if the source string userInput is larger than the destination buffer.
- If user input is larger than the buffer, strcpy will keep copying data past the end of the buffer.
- This can overwrite the local variables and the return address stored in the stack frame.
- Now, here's where an attacker could exploit this vulnerability by calling the function with an input string that is larger than the buffer.

Code execution

```
void smash_stack(const char *userInput)
{
    char buffer[32];
    strcpy(buffer, userInput);
    // ...
}

int main()
{
    // Triggering a stack-based buffer overflow
    // ...acters and overflows into the return
    // ...ddress");
    return 0;
}

void exploit()
{
    // All your base are belong to us!
```

Stack



- The first 32 characters will fill the buffer and the remaining characters will overwrite the return address stored on the stack.

- If the attacker knows the program's memory layout, he could replace the return address with the address of some malicious code.
- When the function finishes execution, instead of jumping back to the main function, it will jump to the code provided by the attacker.
- Thus, by exploiting this buffer overflow vulnerability, an attacker could hijack the control flow of the program and make it execute arbitrary code.
- I'd like to add that modern operating systems have various security mechanisms in place like address space layout randomization (ASLR) and non-executable stacks to make this kind of attack more difficult.

- However, buffer overflows are still a serious issue, and we should follow safe coding practices to avoid these types of vulnerabilities.

Heap overflows are similar to stack overflows, but they occur in dynamically allocated memory on the heap.

- Executing custom code is not usually possible with a heap overflow, but an attacker could still exploit this vulnerability by overwriting the data used by the application.

Heap-based Buffer Overflow

Similar to stack overflows

Happens in dynamically allocated memory

Can be exploited to overwrite application data

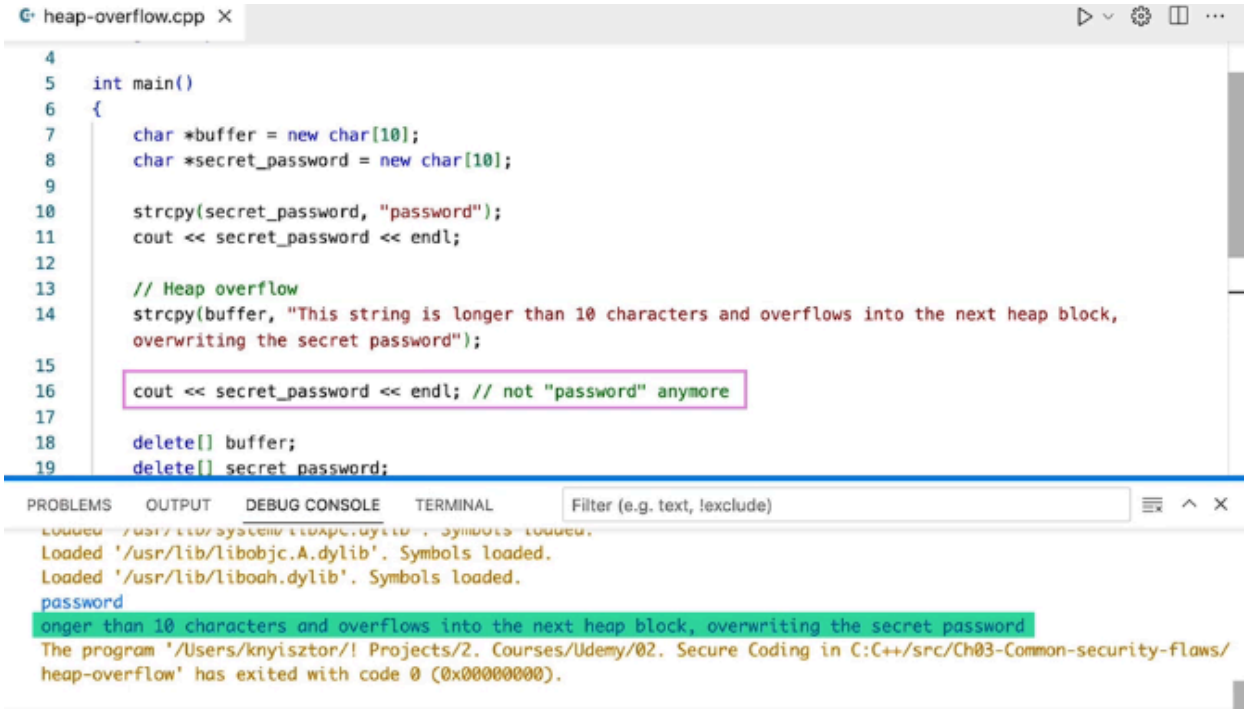
- In the following example, we've got two dynamically allocated blocks on the heap, a buffer and another block that stores a secret password.

```
4
5   int main()
6   {
7       char *buffer = new char[10];
8       char *secret_password = new char[10];
9
10      strcpy(secret_password, "password");
11      cout << secret_password << endl;
12
13      // Heap overflow
14      strcpy(buffer, "This string is longer than 10 characters and overflows into the next heap block,
        overwriting the secret password");
15
16      cout << secret_password << endl; // not "password" anymore
17
18      delete[] buffer;
19      delete[] secret_password;
20
21      return 0;
22  }
23
```

- We first copy the password string into the secret_password block.

Now, storing passwords as plain text in a program is a terrible idea, but bear with me as I'm using
this for illustration purposes only. Then we copy an overly long string into the buffer. The string is much longer than the buffer capacity, causing a heap overflow.

The excess characters spill over into the next block in the heap, which in this case, is our secret_password block.
If attackers can control this overflow and have knowledge or make an educated guess about the layout of the heap, they can override secret_password with a value of their choice.



- The line cout << secret_password << endl; would then not output 'password' as expected.
- Instead, it prints the value an attacker managed to overflow into that memory block, which can lead to serious security risks, depending on how this overwritten data is used elsewhere in our program.
- The biggest takeaway from this lesson is that we must ensure we're not copying more data than the destination buffer can hold.

2.2 Overstepping Limits: Integer Overflows

- Another common security flaw is integer overflow.

- This happens when a signed integer operation results in a value that cannot be represented using the assigned type.

```
2   #include <iostream>
3   using namespace std;
4
5   int main()
6   {
7       int c = INT_MAX;
8       printf("c = %d\n", c);
9       // ...
10      ++c;
11      printf("c = %d\n", c);
12
13      return 0;
14  }
15
```

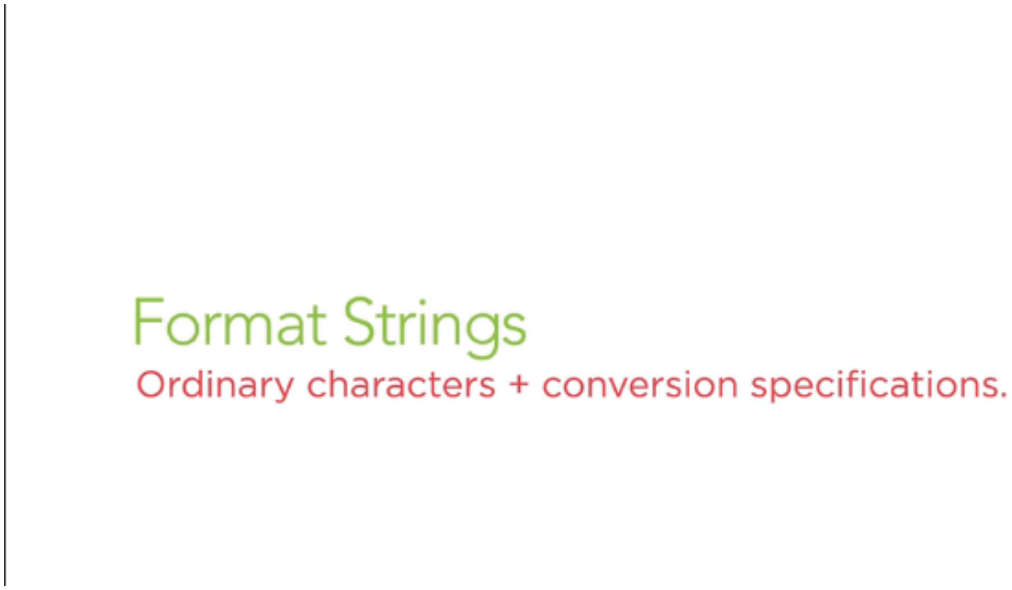
- Integer overflows can result in undefined behaviour and may lead to a security vulnerability if the result is used to control looping, specify the size for memory allocation or as an array index.

2.3 Uncontrolled Format Strings: When Small Mistakes Cause Big Problems

Another common issue that can compromise security is the uncontrolled format string.

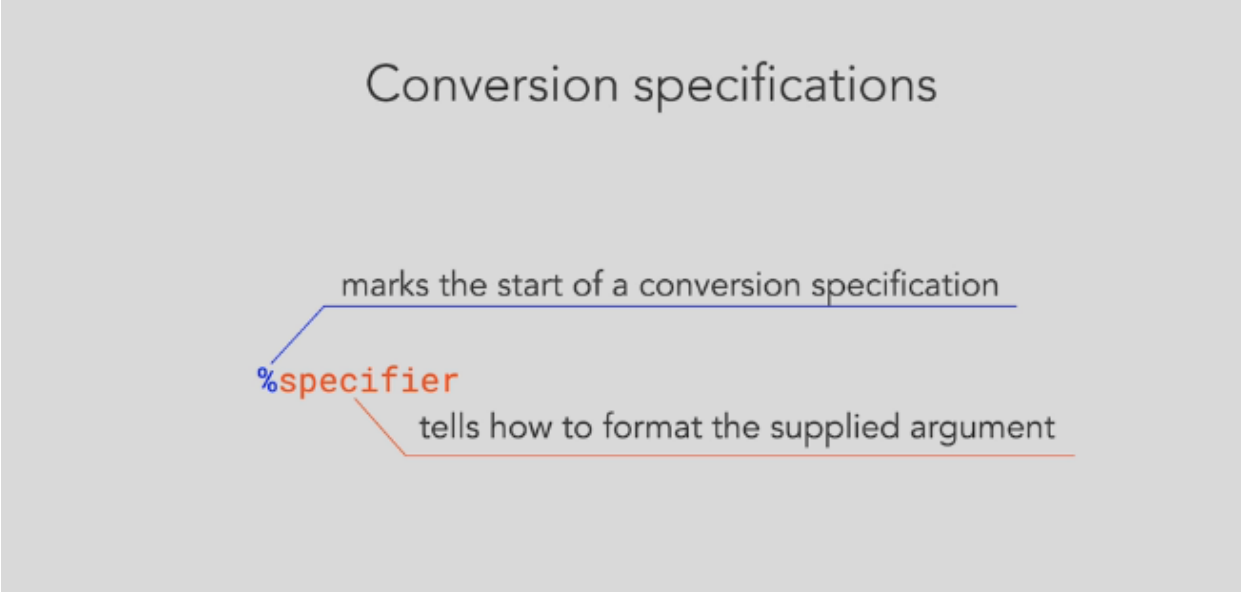
This occurs when a program uses an unchecked format string as an argument to the print command or similar functions.

Format strings consist of sequences of ordinary characters and conversion specifications.



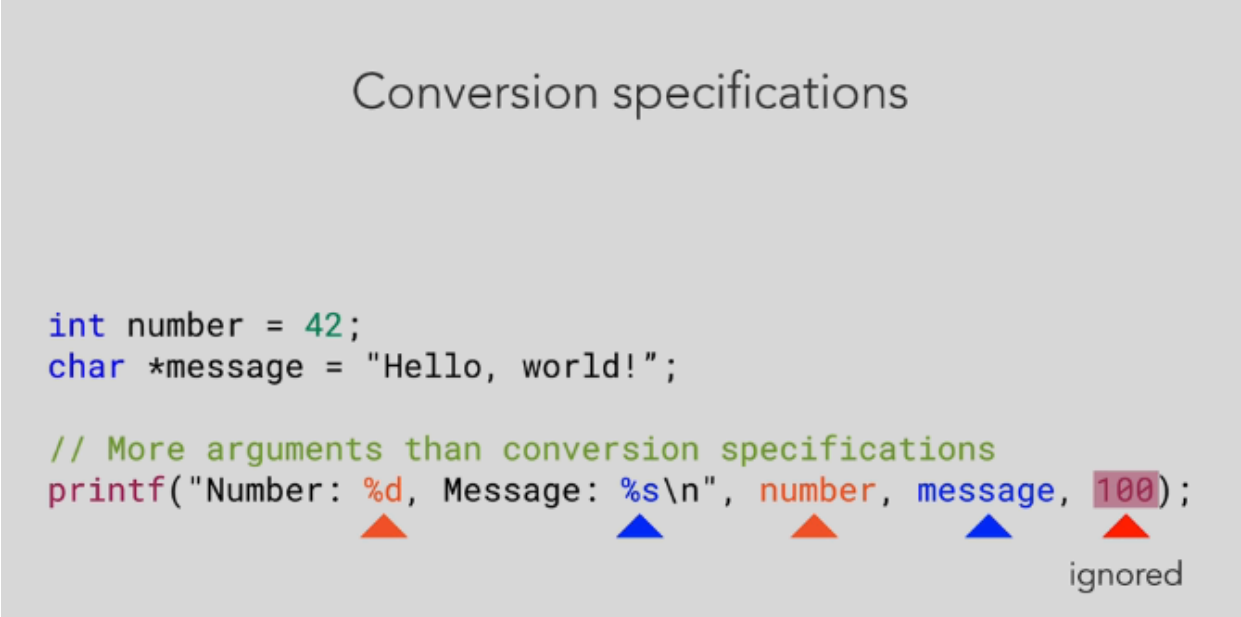
The ordinary characters get copied to the output stream without modifications.

Conversion specifications--in their simplest form--begin with a percent sign, followed by a conversion specifier, which is a character that tells how to format the supplied argument before copying it to the output stream.

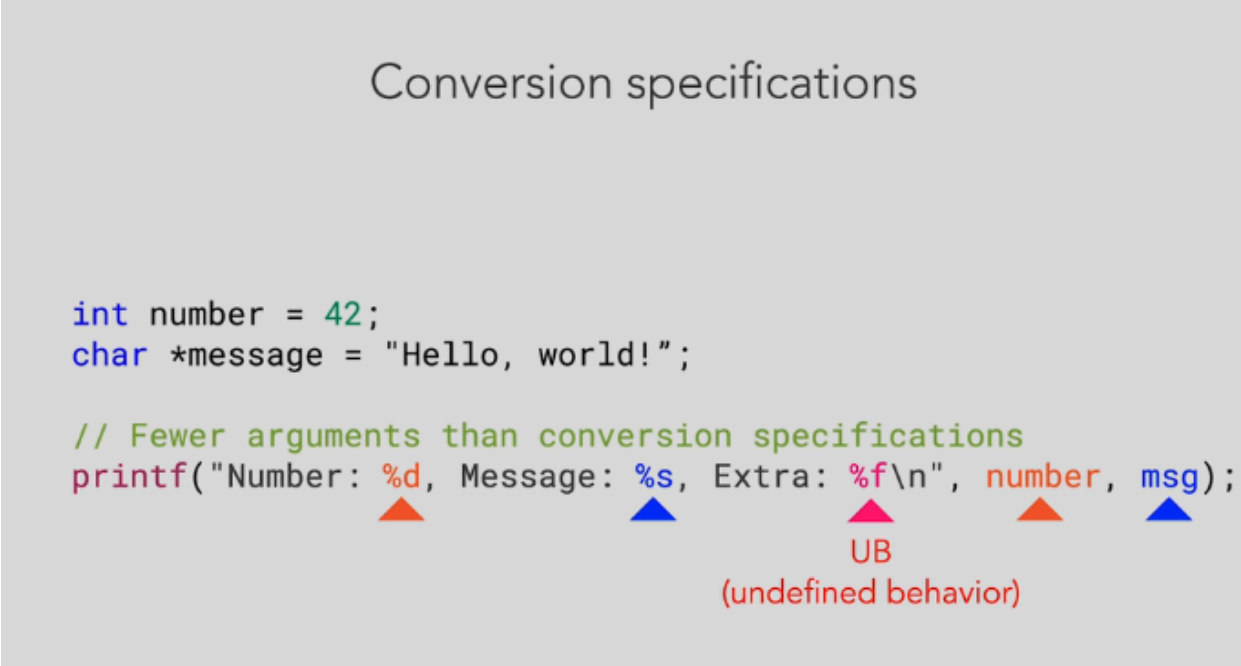


Developers must ensure that the number of arguments matches the specified format.

If there are more arguments than conversion specifications, the extra arguments will be ignored.



However, if the conversion specifications exceed the number of arguments, this will lead to undefined behaviour.



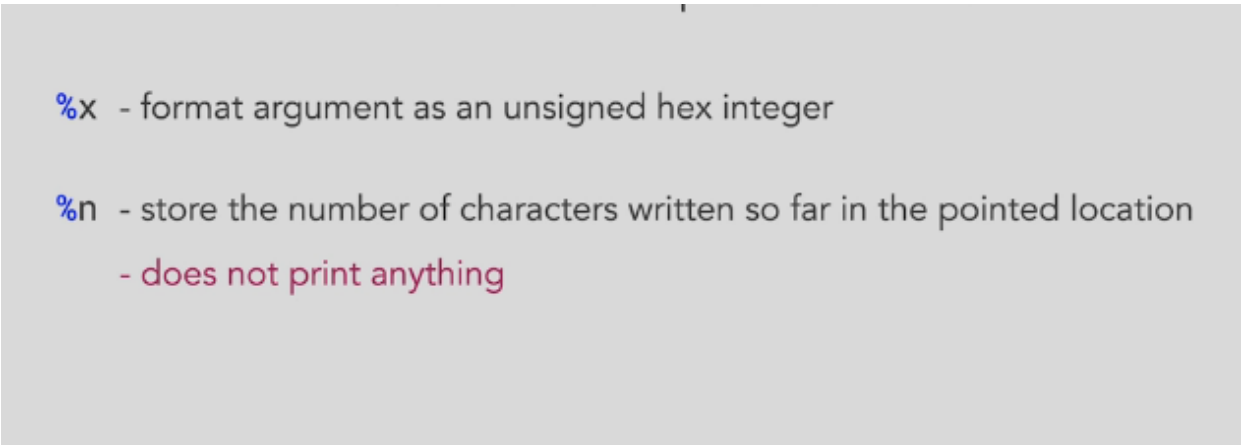
An attacker could use maliciously crafted format strings as arguments for print(), potentially revealing sensitive information stored in memory and even executing arbitrary code.

format-string.cpp

```cpp
using namespace std;

void vulnerable_function(const char *userInput)
{
    printf(userInput); // Uncontrolled format string vulnerability
}

int main()
{
    char userInput[100];

    // The user enters a string with format specifiers
    cout << "Enter a string: ";
    cin.getline(userInput, 100);

    // Prints out the user's input
    cout << "You entered: ";
    vulnerable_function(userInput);

    return 0;
}
```

This code accepts a string from the user and passes it to vulnerable_function() which uses printf() to print the string.

If the user enters a string with format specifiers like %s or %d, printf() will try to use additional arguments that aren't there, resulting in undefined behaviour. And using the %x and %s format
specifiers will cause printf() to print out values from the stack, potentially revealing sensitive information.



Even more dangerous is the %n specifier, which commands printf() to write the number of characters successfully written so far to an address pointed to by the corresponding argument.

As there is no corresponding argument, printf() will write to whatever location is at the top of the
stack, leading to a potential security vulnerability.

```c
int count = 0;

printf("Welcome! %n", &count);

printf("%d characters printed so far.\n", count);

// Output: 9 characters printed so far.
```

Risky %n

```c
// Prints "9" to a random memory location
printf("Welcome! %n");
```

Actually, %n has been deemed so risky that it has been banned in many coding guidelines and some compilers or runtime environments may even disable it by default.

Still, if encountered and exploited, it could lead to arbitrary code execution or allow an attacker to override data in memory.

Therefore, it's imperative to validate all user inputs and never use unchecked user input as a format string to mitigate such vulnerabilities.

Now, we can quickly fix the issue in this particular case.

Replacing printf(userInput) with printf("%s", userInput) prevents any format specifiers in userInput from being interpreted by printf().

```
2   using namespace std;
3
4   void vulnerable_function(const char *userInput)
5   {
6       //printf(userInput); // Uncontrolled format string vulnerability
7       printf("%s", userInput);
8   }
9
10  int main()
11  {
12      char userInput[100];
13
14      // The user enters a string with format specifiers
15      cout << "Enter a string: ";
16      cin.getline(userInput, 100);
17
18      // Prints out the user's input
19      cout << "You entered: ";
20      vulnerable_function(userInput);
21
22      return 0;
23  }
24
```

Alternatively, we could use cout instead of printf because cout does not interpret format specifiers.

```
using namespace std;

void vulnerable_function(const char *userInput)
{
    //printf(userInput); // Uncontrolled format string vulnerability
    cout << userInput << endl;
}

int main()
{
    char userInput[100];

    // The user enters a string with format specifiers
    cout << "Enter a string: ";
    cin.getline(userInput, 100);

    // Prints out the user's input
    cout << "You entered: ";
    vulnerable_function(userInput);

    return 0;
}

```

Secure coding may sound like a daunting task.

Still, more often than not, it boils down to being aware of the potential risks and utilising the safer features and techniques available within the language.

2.4 The Dangers of Improper Error Handling

Proper error handling is a critical aspect of writing secure and robust code.

Failing to handle errors appropriately can leave your code vulnerable to various security risks and impact your program's reliability and stability.

One common mistake in error handling is simply ignoring errors or failing to check the return values of functions that can fail.

This can lead to unexpected behaviour and potential security vulnerabilities.

Consider the following code snippet.

Ignored Errors

```
char buffer[256];
FILE *file = fopen("data.txt", "r");

fscanf(file, "%s", buffer);
cout << "First word: " << buffer << endl;
fclose(file);
```

In this example, the return value of fopen() is not checked. If the file fails to open due to permission issues or file not found, the subsequent calls to fscanf() will read data from an uninitialized or incorrect file stream, leading to undefined behaviour.

To reduce the chance of potential risks, always check the return values of functions that can fail
and handle errors appropriately. In this case, we could add code to handle the failed file opening scenario and take necessary actions, such as logging the error, notifying the user or terminating the program.

Fix for Ignored Errors

```
char buffer[256];
FILE *file = fopen("data.txt", "r");

if (!file)
{
    printf("Failed to open the file.\n");
    return;
}

fscanf(file, "%s", buffer);
cout << "First word: " << buffer << endl;
```

Incomplete or incorrect error messages.

Providing clear and useful error messages is really important for fixing and resolving issues.

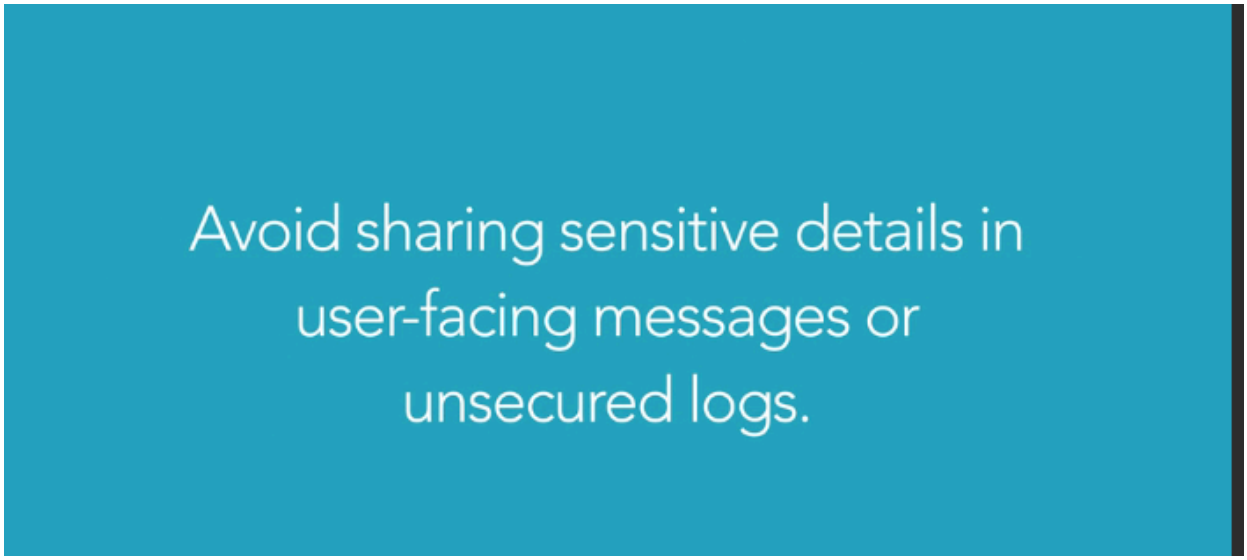
At the same time, keeping sensitive information secure is just as crucial.

Error messages can accidentally leak internal details about the system or provide valuable information to potential attackers.

In the following example, the error message reveals the internal error code, which an attacker could exploit to gain insight into the system's vulnerabilities.

To address this issue, it's important to carefully consider what we put into our log messages, especially if those messages can easily be accessed or intercepted.

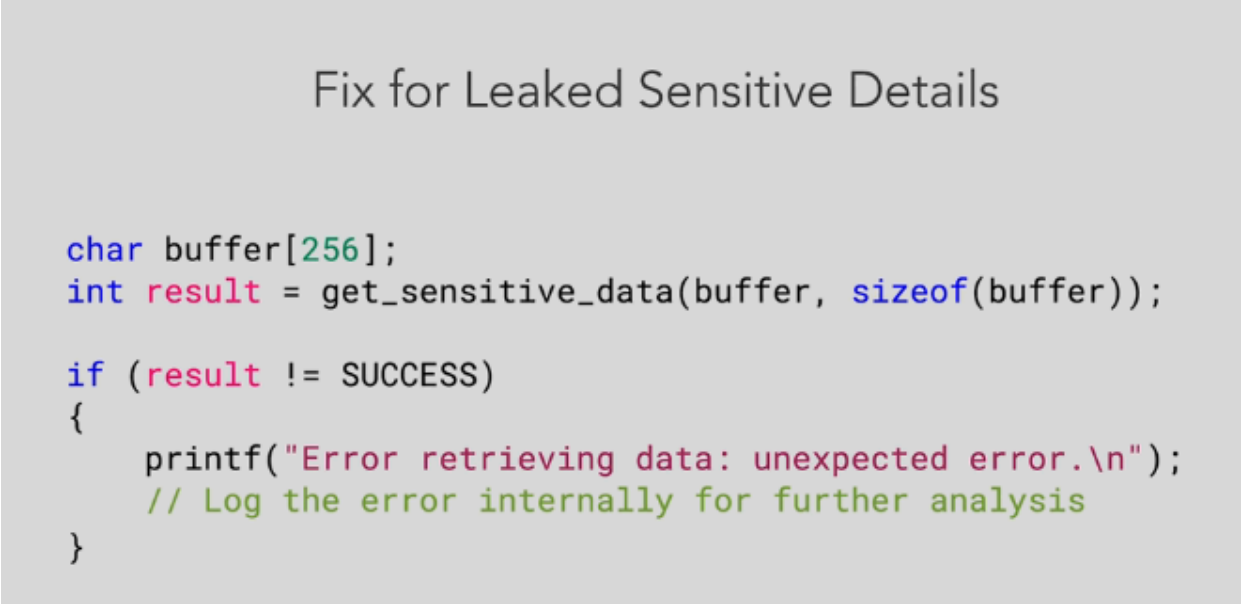
You should avoid displaying detailed error messages in production environments, as this can aid attackers in identifying potential weaknesses.



If identifying and fixing issues requires specific details, it is best to implement secure logging
techniques that limit access to sensitive data.

This includes error codes and access system resources that could be targeted by potential attackers.

Here's an improved version of the code.



The error message is not specific and doesn't disclose any confidential data.

The actual error code obtained from the get_sensitive_data() function should be securely logged if it's useful for troubleshooting.

Resource leaks.

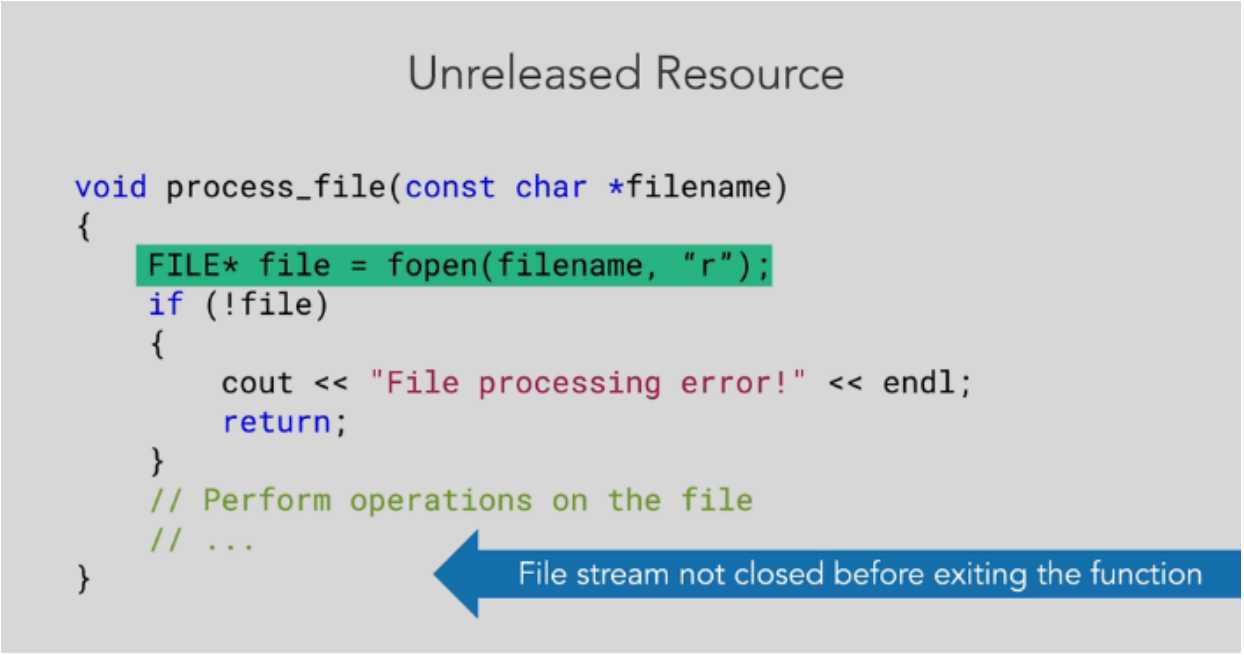
Improper handling of resources such as memory or file handles can result in resource leaks.

Improper Error Handling

1. Ignoring errors
2. Incorrect error messages
3. Resource leaks

Such leaks occur when resources are not properly released or freed, leading to a waste of system resources and potential denial-of-service vulnerabilities.

In the following example, the file handle is not released after use.

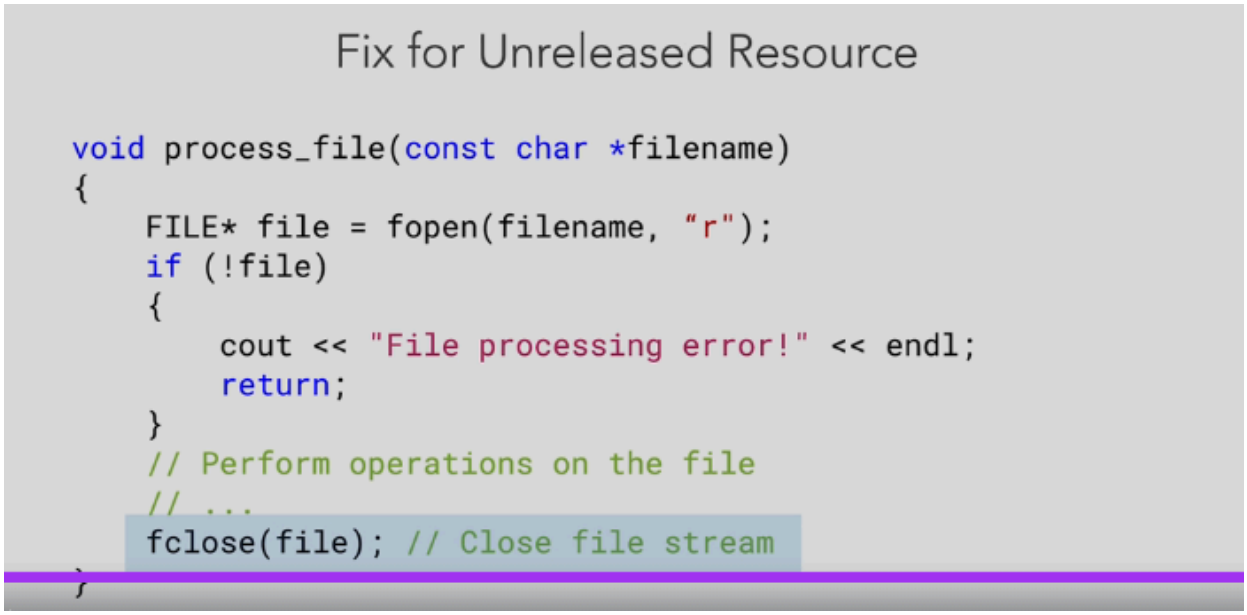


If this code is executed repeatedly, it can exhaust the available file handles and cause a denial of service condition.

To prevent resource leaks,

always ensure that resources are properly released when they are no longer needed.

By explicitly releasing the file handle with fclose(), you prevent resource leaks and ensure efficient resource utilisation.



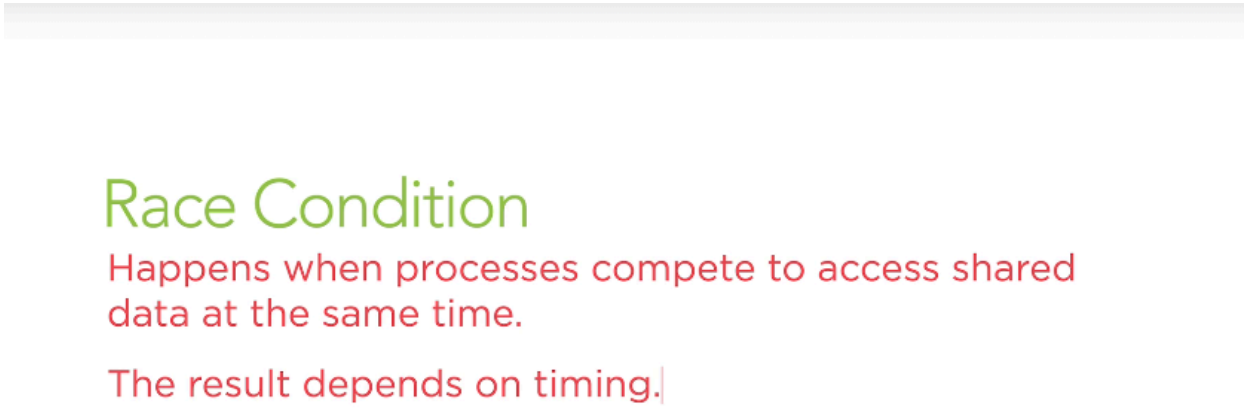
So, in summary, improving error handling, ensuring appropriate information in logs, and releasing system resources when not in use can enhance the security and reliability of your code.

2.5 When Code Collides: Race Conditions

Concurrency is a powerful concept in programming that allows multiple tasks or threads to execute simultaneously, improving performance and responsiveness.

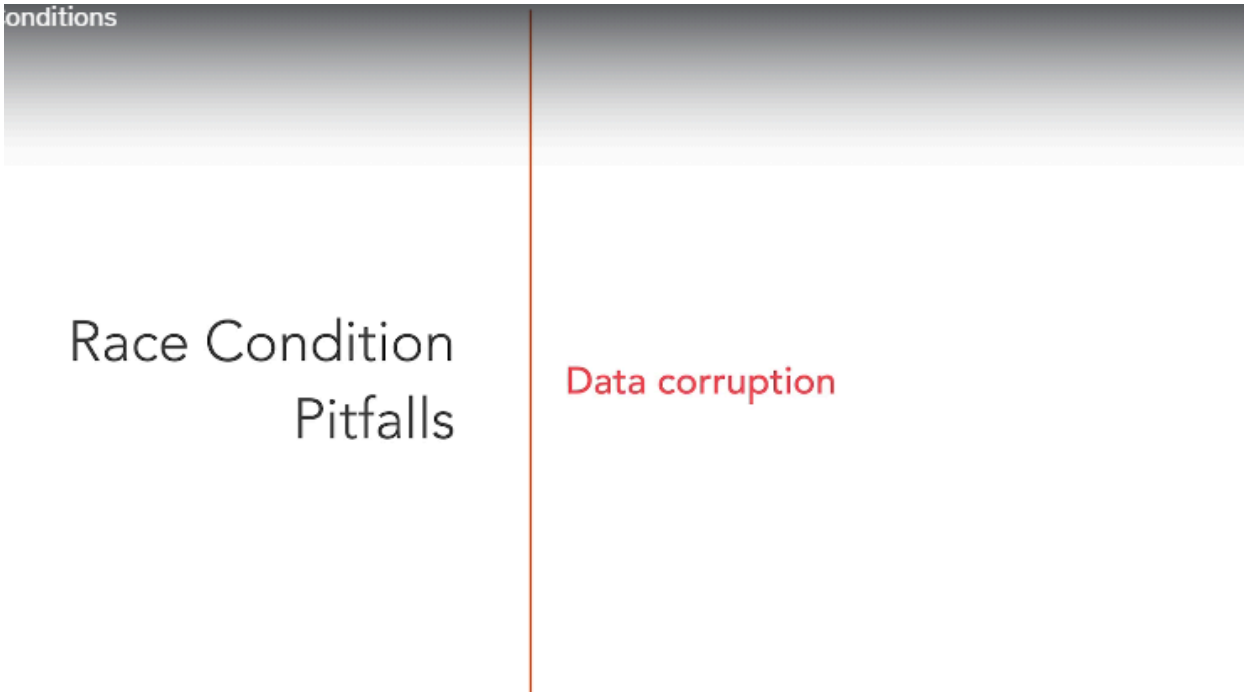
However, it introduces new challenges, including the potential for race conditions.

A race condition refers to a situation where two or more processes are racing to access or change shared data concurrently, and the outcome depends on the specific timing or how the processes are executed.



Race conditions can lead to sporadic, hard to detect bugs and security vulnerabilities with severe consequences.

Let's explore how race conditions can pose security risks and why attention to detail is crucial in handling concurrency.



Let's have a look at the following program, where two threads concurrently increment a shared variable.

```cpp
#include <iostream>
#include <thread>
using namespace std;

int counter = 0;

void increment_counter()
{
    for (int i = 0; i < 10000; ++i)
    {
        counter++;
    }
}

int main()
{
    thread t1(increment_counter);
    thread t2(increment_counter);

    t1.join();
    t2.join();

    cout << "Counter value: " << counter << endl;

    return 0;
```

Race condition occurs if both threads access and modify the counter variable simultaneously without proper synchronisation. As a result, the final value of the counter may not be the expected value of 20,000.

```cpp
using namespace std;

int counter = 0;

void increment_counter()
{
    for (int i = 0; i < 10000; ++i)
    {
        counter++;
    }
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  Filter (e.g. text, !exclude)
Loaded '/usr/lib/system/libsystem_kernel.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libsystem_platform.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libsystem_pthread.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libsystem_symptoms.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libsystem_trace.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libunwind.dylib'. Symbols loaded.
Loaded '/usr/lib/system/libxpc.dylib'. Symbols loaded.
Loaded '/usr/lib/libobjc.A.dylib'. Symbols loaded.
Loaded '/usr/lib/liboah.dylib'. Symbols loaded.
Counter value: 14060
The program '/Users/knyisztor/! Projects/2. Courses/Udemy/02. Secure Coding in C:C++/src/Ch03-Comm
```

The data corruption caused by the race condition compromises the integrity of the counter variable, potentially leading to incorrect program behaviour and security vulnerabilities.

By introducing a mutex (counterMutex) and using std::lock_guard to lock and unlock the mutex during the critical section, we ensure that only one thread can access and modify the counter variable at a time, preventing data corruption caused by a race condition.

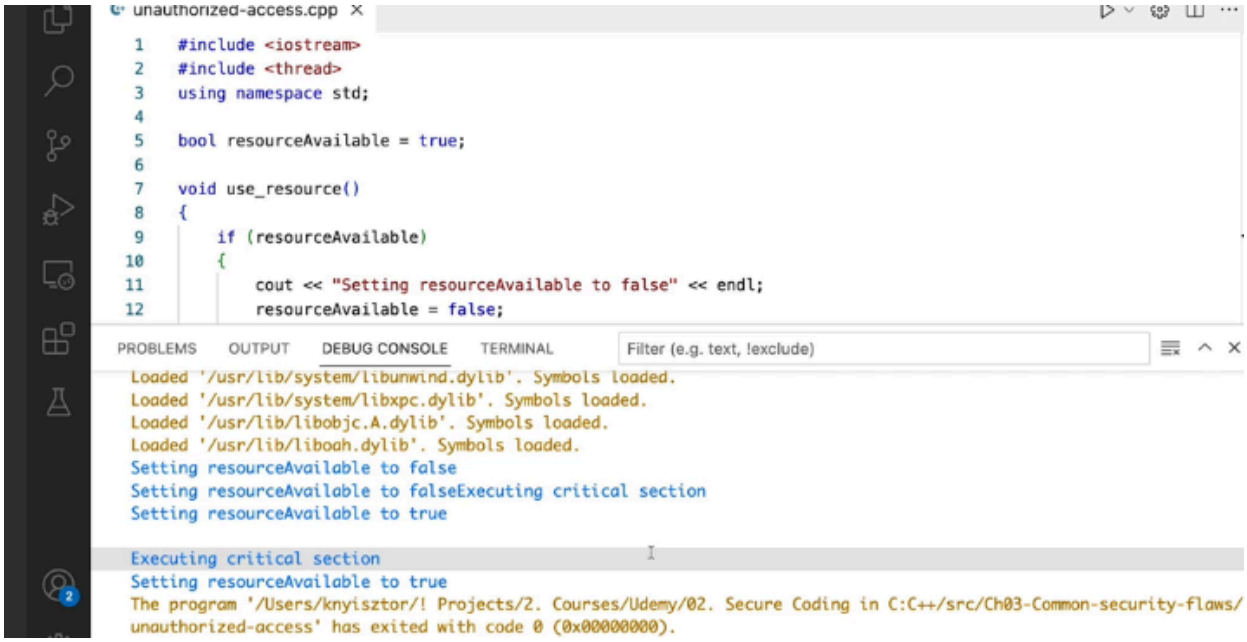
Unauthorised access.

Race conditions can also result in unauthorised access to protected resources.

Consider the following scenario where a resource is accessed based on a check.

```cpp
#include <iostream>
#include <thread>
using namespace std;

bool resourceAvailable = true;

void use_resource()
{
    if (resourceAvailable)
    {
        cout << "Setting resourceAvailable to false" << endl;
        resourceAvailable = false;
        cout << "Executing critical section" << endl;
    }
    cout << "Setting resourceAvailable to true" << endl;
    resourceAvailable = true;
}

int main()
{
    std::thread t1(use_resource);
    std::thread t2(use_resource);

    t1.join();
    t2.join();
```

In this case, both threads check the resourceAvailable flag before accessing the resource. However, due to the lack of proper synchronisation, a race condition occurs. This race condition allows both threads to enter the critical section simultaneously, bypassing the intended access control mechanism.



Unauthorised access to the protected resource can compromise security and lead to undefined behaviour.

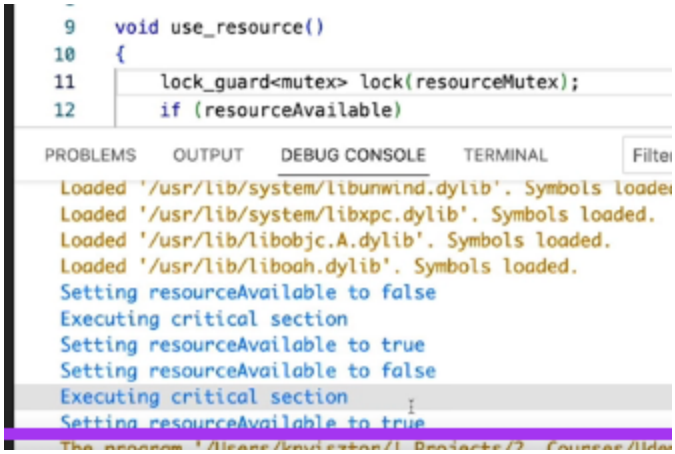
By introducing a mutex (resourceMutex) and using std::lock_guard to protect the critical section, we ensure that only one thread can access and modify the resourceAvailable flag at a time, preventing unauthorised access to the protected resource.

unauthorized-access.cpp 1

```cpp
#include <iostream>
#include <thread>
#include <mutex>
using namespace std;

bool resourceAvailable = true;
mutex resourceMutex;

void use_resource()
{
    lock_guard<mutex> lock(resourceMutex);
    if (resourceAvailable)
    {
        cout << "Setting resourceAvailable to false" << endl;
        resourceAvailable = false;
        cout << "Executing critical section" << endl;
    }
    cout << "Setting resourceAvailable to true" << endl;
    resourceAvailable = true;
}
```

To prevent race conditions, it's essential to use appropriate synchronisation mechanisms such as locks, mutexes, or atomic operations.

This ensures the proper ordering and synchronisation of concurrent operations.



Additionally, designing thread-safe algorithms and data structures can minimise the occurrence of race conditions.

Even these overly simplified code examples demonstrate that concurrent programming is a complex and challenging topic.

3. Principles of Secure C and C++ Programming.

3.1 Minimise Attack Surface Area

Let's delve into some key principles that can help us improve our C and C++ coding habits, resulting in code that's more robust and secure.

One of the fundamental principles of secure programming is to minimise the attack surface area.

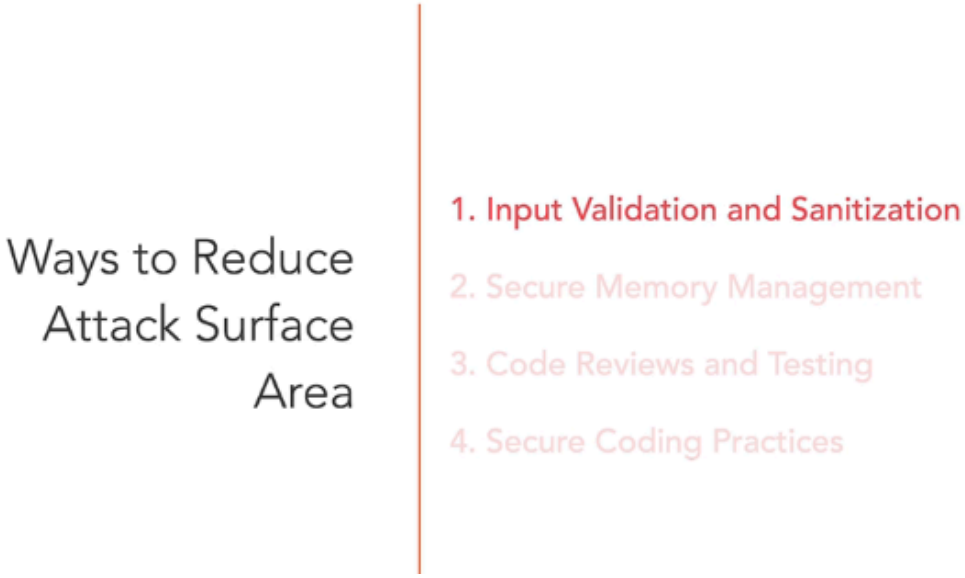
The attack surface area refers to the sum of all points in your code where an attacker could potentially exploit vulnerabilities.

By reducing this surface area, we can lower the chances of successful attacks and enhance the overall security of our software.

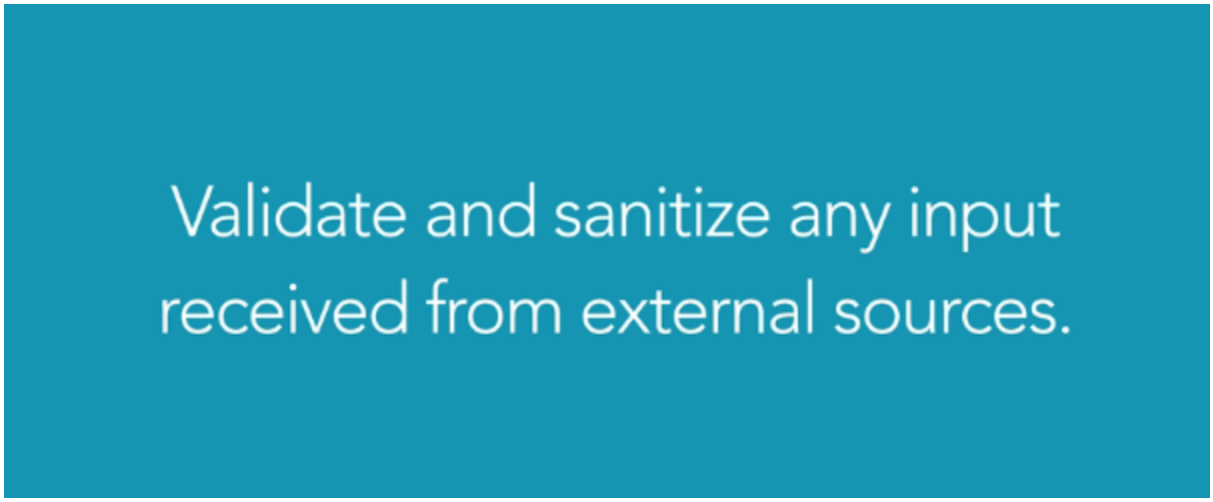
Think of it this way: the smaller the target, the more challenging it becomes to hit it.

There are several techniques and best practices that can help minimise the attack surface area in C and C++ programs.

Input validation and sanitization.



The goal is to thoroughly validate and sanitise all input from external sources such as user input or network data.



Input validation helps ensure that the program only processes expected and valid data, preventing various forms of attacks, including buffer overflows, injection attacks and command execution vulnerabilities.

Here is an example of input validation.

input-validation.cpp

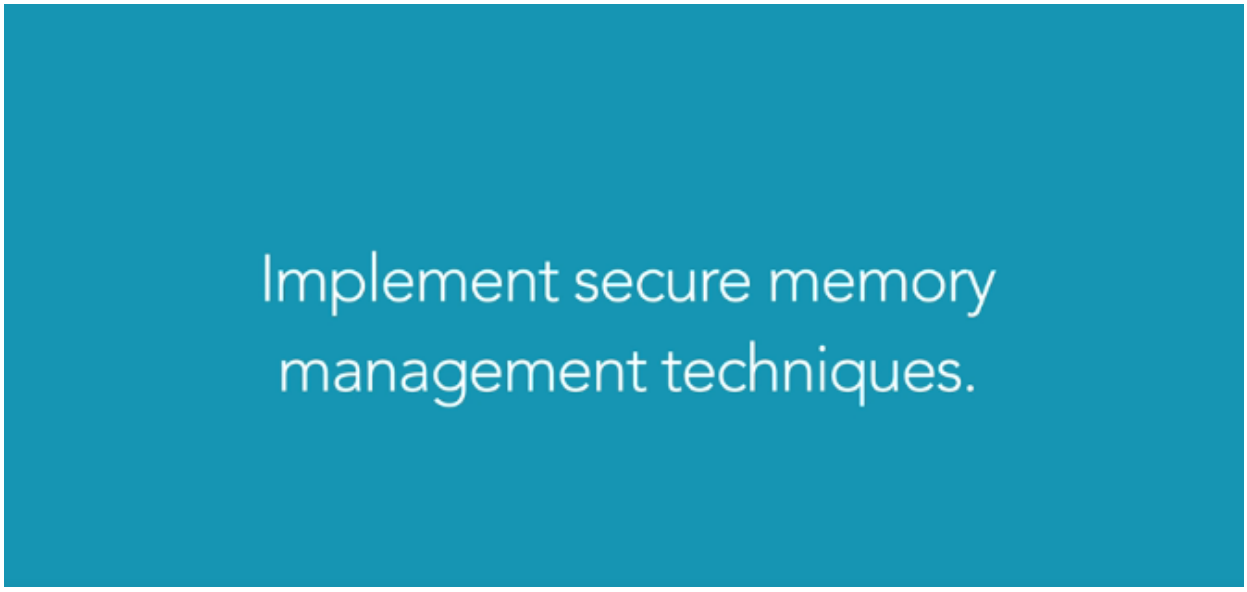
```cpp
#include <iostream>
using namespace std;

void process_input(const char *userInput)
{
    // Check for format string vulnerabilities
    if (strpbrk(userInput, "%"))
    {
        cout << "Invalid input. Format string vulnerability detected." << endl;
        return;
    }

    // Process the input safely
    printf("User Input: %s\n", userInput);
}

int main()
{

    char userInput[100];
    cout << "Enter your input: ";
    cin.getline(userInput, sizeof(userInput));

    process_input(userInput);

    return 0;
}
```

In the main() function we ask for user input. Then the array containing the characters entered by the user gets passed to the process_input() function. The process_input() function first checks for the presence of a percent character using a built-in function. If a percent character is found, it indicates a potential format string vulnerability and an error

message gets displayed. By including the format string vulnerability check within the process_input() function, we ensure that the input is validated and processed securely.

Secure memory management.

Use secure memory management techniques such as utilising smart pointers in C++ to automate memory allocation and deallocation, reducing the risk of memory leaks and buffer overflows.



Smart pointers handle memory management automatically, preventing common programming mistakes that can lead to vulnerabilities.

Code reviews and testing.

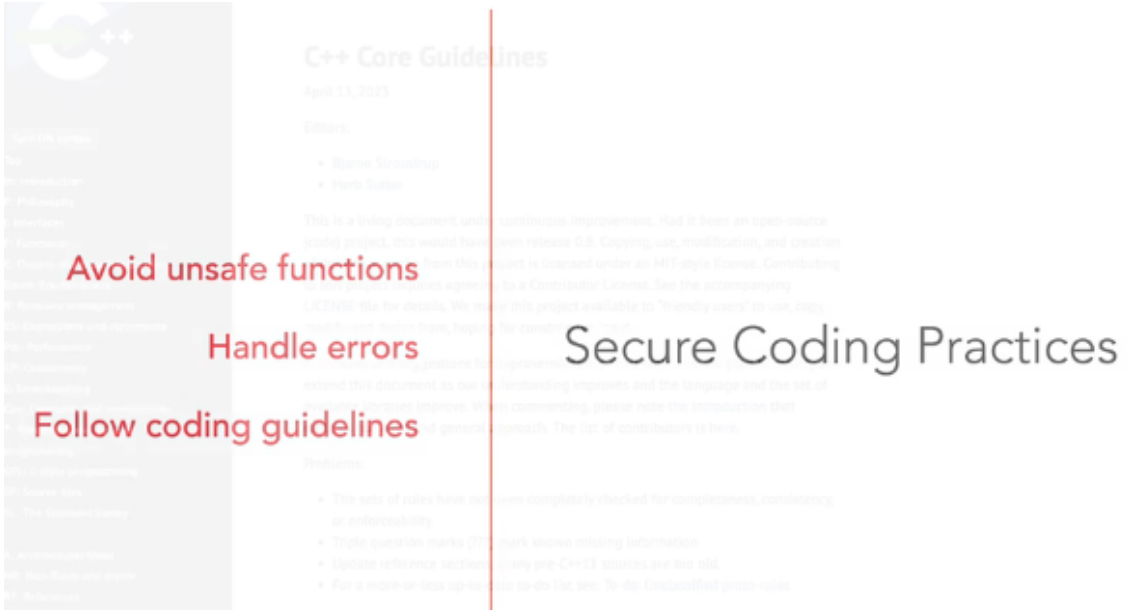
Conduct thorough code reviews and testing to identify potential security risks.

Peer reviews and security focused testing can help uncover issues that may have been overlooked during development.

Consider using static code analysis tools and security testing frameworks to enhance the robustness of your code.

Secure coding practices.

Adhere to secure coding practices, such as avoiding the use of deprecated or unsafe functions, correctly handling error conditions and following established coding standards and guidelines.



Discover the latest security features by familiarising yourself with the most recent C and C++ standards, and then incorporate them into your code by including these techniques and best practices in your C and C++ programming practices, you can effectively minimise the attack surface area of your code, making it more resistant to potential attacks.

3.2 Principle of Least Privilege

The Principle of Least Privilege is a fundamental principle in secure programming that advocates granting the minimum necessary privileges to perform a specific task. By restricting access rights and privileges to only what is required, the potential damage an attacker or a compromised component can cause is significantly minimised.

In C and C++ programming, the Principle of Least Privilege can be applied in various ways.

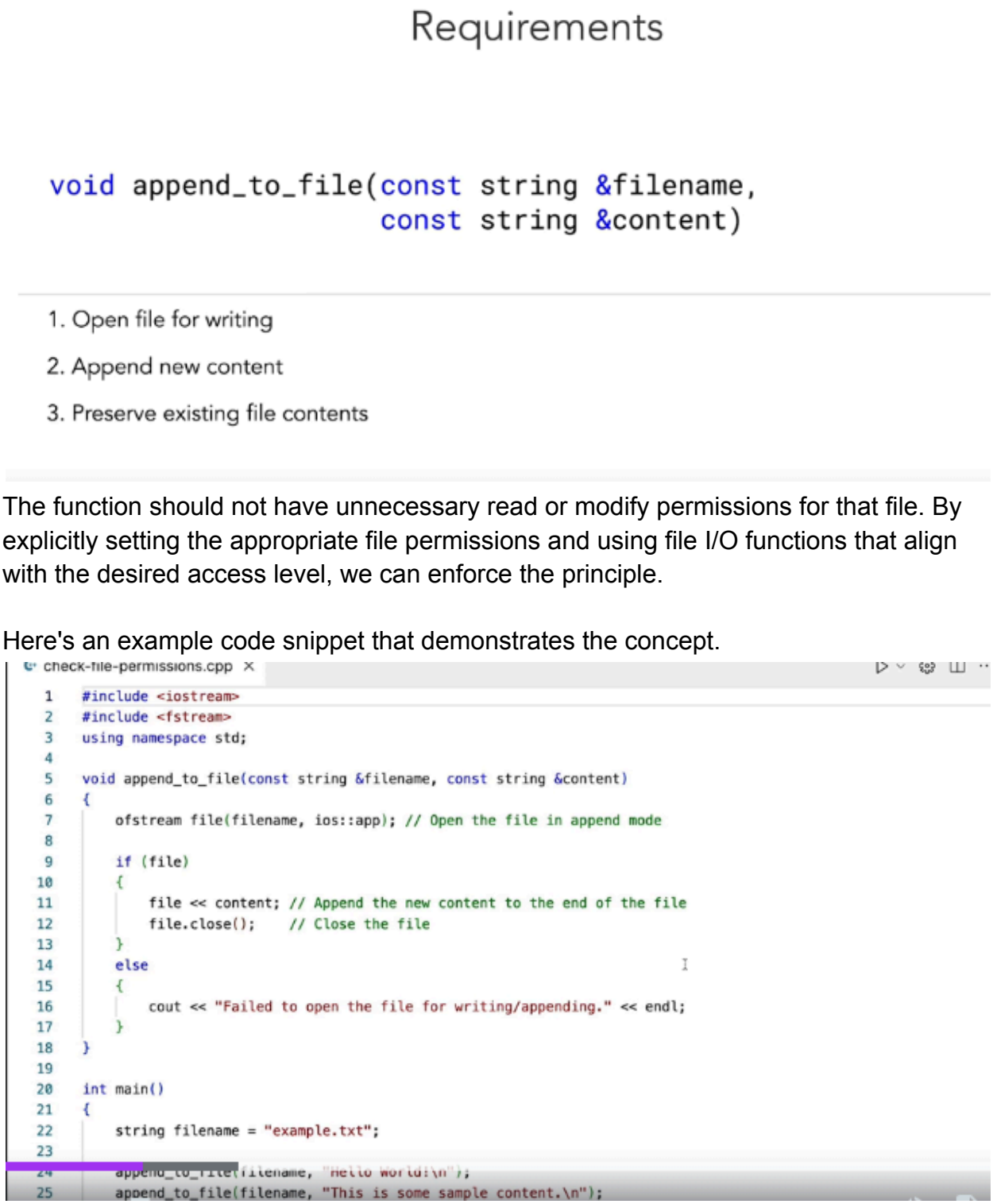
Let's explore a few examples.

Limiting code permissions

In the context of applying the principle of least privilege to C and C++ coding, one crucial aspect is limiting code permissions to the necessary minimum.

By doing so, we ensure that code only has access to the required resources and operations, reducing the potential attack surface and vulnerabilities.

For example, consider a function that needs to open a file for writing and append new content to the end of the file while preserving the existing file contents.



The function should not have unnecessary read or modify permissions for that file. By explicitly setting the appropriate file permissions and using file I/O functions that align with the desired access level, we can enforce the principle.

Here's an example code snippet that demonstrates the concept.

check-file-permissions.cpp

```cpp
#include <iostream>
#include <fstream>
using namespace std;

void append_to_file(const string &filename, const string &content)
{
    ofstream file(filename, ios::app); // Open the file in append mode

    if (file)
    {
        file << content; // Append the new content to the end of the file
        file.close();    // Close the file
    }
    else
    {
        cout << "Failed to open the file for writing/appending." << endl;
    }
}

int main()
{
    string filename = "example.txt";

    append_to_file(filename, "Hello World!\n");
    append_to_file(filename, "This is some sample content.\n");
```

The append_to_file() function opens the file in append mode using the std::ios::app flag, which allows writing to the end of the file while preserving the existing content. By using append mode, the function is limited to the necessary permissions for appending to a file without being able to read or modify it. This approach demonstrates the concept of limiting code permissions by granting the function only the necessary capabilities to perform its intended task, In line with the Principle of Least Privilege.

Controlling system calls.

In certain scenarios, your code may need to interact with the operating system through system calls.

However, it's crucial to carefully validate and sanitise the input to prevent unintended or malicious actions.

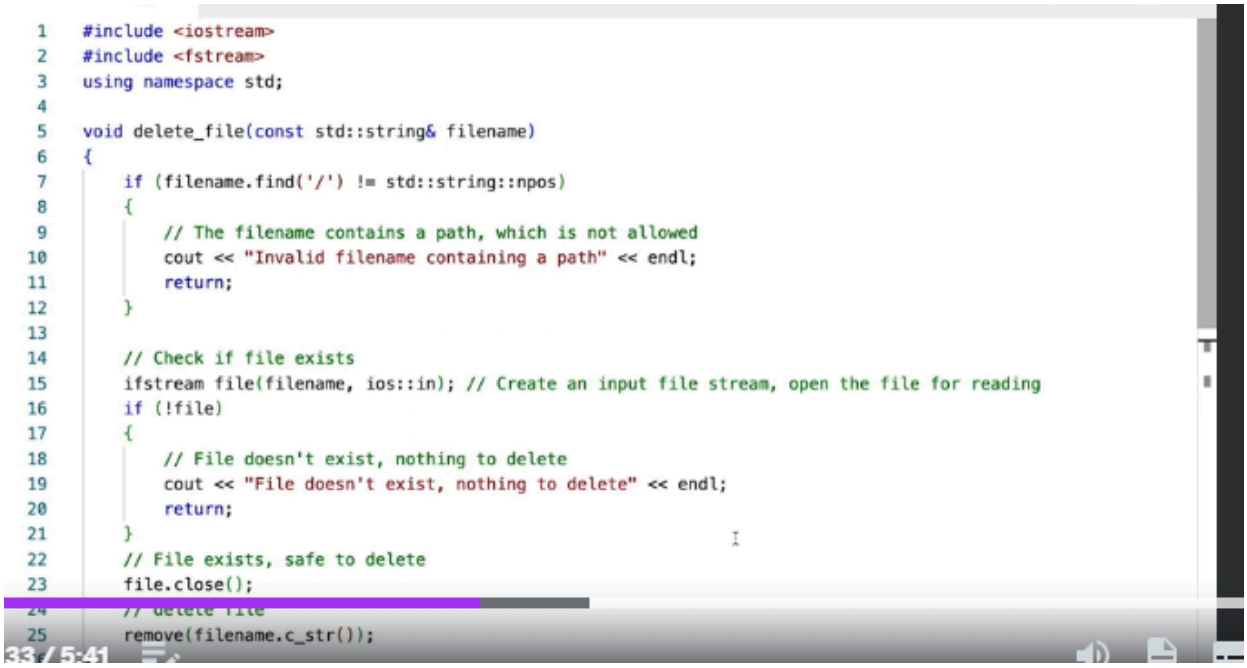
Controlling System Calls

Validate and sanitize the input

Prevent unauthorized access & unexpected behavior

By implementing these measures, you can significantly reduce the risk of unauthorised access or unexpected system behaviour.

In this program, the delete_file() function is designed to securely remove a file with a specified file



name from the current working directory.

The function incorporates several measures to ensure the deletion is performed safely and within the intended scope.

First, the function checks if the file name contains a forward slash character, which is commonly
used to specify a directory path in Unix-based systems. This check is essential to prevent the accidental deletion of files beyond the intended directory. Next, the function verifies if the file exists by creating an input file stream (std::ifstream) and attempting to open the file for reading.

If the file does not exist, the function returns early, ensuring that no actions are taken on a non-existent file. If the file exists, the input file stream is closed. Finally, we call the remove() function to delete the file from the filesystem. By performing these checks and taking appropriate actions, the delete_file() function adheres to the Principle of Least Privilege by limiting its capabilities to securely delete files within the current working directory and preventing unintended file deletions outside of the intended scope.

Applying Least Privilege in user permissions.

When developing applications that require access control, it's crucial to adhere to the Principle of

Least Privilege when assigning privileges or roles to different user accounts.

By following this principle, users are granted only the specific permissions necessary for their intended actions, effectively minimising the potential impact of compromised accounts.

The following code demonstrates a basic implementation of permission checks using an enumeration called

UserRole.

The perform_restricted_action function takes a userRole parameter and performs different actions based

on the user's role.

The output messages indicate the level of access granted to each user role.

By utilising this approach, the code ensures that guest users have restricted access, regular users have the necessary access privileges, and admin users possess full access rights.

This, in turn, aligns with the Principle of Least Privilege by providing users with the minimum privileges required to fulfil their intended tasks, preventing unauthorised access and potential misuse of system resources.

user-permissions.cpp

```cpp
#include <iostream>
using namespace std;

enum class UserRole
{
    Guest,
    User,
    Admin
};

void perform_restricted_action(UserRole userRole)
{
    switch (userRole)
    {
    case UserRole::Guest:
        cout << "Guest user: Restricted access." << endl;
        break;
    case UserRole::User:
        cout << "Regular user: Access granted." << endl;
        break;
    case UserRole::Admin:
        cout << "Admin user: Full access." << endl;
        break;
    }
```

Applying the Principle of Least Privilege in access control reduces the attack surface area and limits the potential impact of security vulnerabilities.

It helps to enforce the Principle of Separation of Privileges and ensures that users can only perform actions within their designated roles, enhancing the overall security of the system.

3.3 Fail-safe Defaults

The Principle of Fail-safe Defaults revolves around the idea that access to resources and sensitive operations should be denied by default, with explicit permission required for access.

This approach minimises the risk of unauthorised access and potential exploits.

To understand why this is the right strategy, consider the opposite scenario where access is granted by default, but turned off if certain conditions are met.

In this case, there is a higher likelihood of overlooking critical conditions or failing to promptly revoke access, leading to potential security breaches.

Imagine a corporate office with restricted access areas, such as the server room or the executive suite.

In a fail-safe approach, access to these areas would be denied, requiring employees to explicitly request access through proper authorization channels.

This ensures that only authorised personnel can enter these sensitive spaces.

However, if access is not explicitly denied, employees who were initially granted access might enter restricted areas without proper authorization.

Let's see a practical application of the Principle of Fail-safe Defaults.

The following code illustrates the process of initialising application settings from a configuration file.

fail-safe-defaults.cpp

```cpp
#include <iostream>
#include <fstream>
using namespace std;

struct Settings
{
    bool debugModeEnabled;
    int logLevel;
    int maxDebugMessages;
};

void initialize_settings(const string& configFile, Settings& settings)
{
    // Check if the configuration file exists
    ifstream file(configFile, ios::in);
    if (!file)
    {
        // Configuration file does not exist, use fail-safe defaults
        settings.debugModeEnabled = false;
        settings.logLevel = 2;
        settings.maxDebugMessages = 500;
    }
    else
    {
        // Read settings from the configuration file
        file >> settings.debugModeEnabled;
        file >> settings.logLevel;
```

```cpp
void initialize_settings(const string& configFile, Settings& settings)
{
    // Check if the configuration file exists
    ifstream file(configFile, ios::in);
    if (!file)
    {
        // Configuration file does not exist, use fail-safe defaults
        settings.debugModeEnabled = false;
        settings.logLevel = 2;
        settings.maxDebugMessages = 500;
    }
    else
    {
        // Read settings from the configuration file
        file >> settings.debugModeEnabled;
        file >> settings.logLevel;
        file >> settings.maxDebugMessages;

        // Validate and sanitize the settings if necessary
        // ...
    }

    // Close the configuration file
    file.close();
}

int main()
```

```cpp
}

int main()
{
    Settings settings;
    initialize_settings("config.txt", settings);

    // Use the initialized settings in the application
    cout << "DebugModeEnabled: " << (settings.debugModeEnabled ? "enabled" : "disabled") << endl;
    cout << "Log Level: " << settings.logLevel << endl;
    cout << "Max Debug Message Count: " << settings.maxDebugMessages << endl;

    return 0;
}
```

We first define a structure that encapsulates various application settings.

To initialise these settings, we have a function called initialize_settings(), which takes the name of the configuration file and the Settings reference as parameters. Within the function, we check if the configuration file exists by attempting to open it using an input file stream. If the file doesn't exist, we fall back to fail-safe defaults, setting debugModeEnabled to false, logLevel to two, and maxDebugMessages to 500.

If the configuration file is found, we read the settings from the file using the stream extraction operator into the corresponding properties of the Settings structure.

Our application remains functional even if the configuration file is missing because we have incorporated default values directly into the code.

Note that the provided settings need to be the most secure and the safest to comply with the Fail-safe Defaults principle.

3.3 Defence in Depth

When it comes to securing your C and C++ applications.

Relying on a single security mechanism is often not enough.

A comprehensive security approach involves employing multiple layers of defence to protect your system.

This strategy is known as Defense in Depth.

The idea is to design your system in a way that even if one layer of defence is breached, there are still other layers in place to prevent or lessen the impact of an attack.

This approach makes it harder for attackers to achieve their goal, as they will have to overcome several security measures, increasing the difficulty and effort required to compromise the system.

Now let's have a look at the following code.

In this example, the code incorporates three security mechanisms: user authentication, access control,

check, and biometric verification.

Step 1: Authenticate User.

The sensitive_operation() function first calls authenticate_user() to perform user authentication.

This step typically involves validating the username and password combination or other authentication

mechanisms.

If this step fails, the operation is denied.

Step 2: Verify Access Permissions.

After successful authentication, the function checks the user's access permissions using the

verify_access_permissions() function.

This step ensures that only authorized users with the necessary access rights can proceed.

If the access permissions are not sufficient, the function displays a warning message and exits.

Step 3: Verify Biometrics.

After confirming that the user has appropriate access permissions, the function moves on to authenticate

the user's biometrics by calling the verify_biometrics() function.

This step could involve fingerprint matching, face recognition or other biometric verification methods.

If the biometric verification fails, the function displays an access denied message and returns without

proceeding further.

If all the security checks pass, the function proceeds to read an encrypted file.

If the file is successfully opened, the function continues with reading the encrypted file,

decrypting the sensitive data and performing additional processing.

This code example demonstrates the Defense in Depth approach.

It prepares for the possibility of the first layer being breached, where the attacker bypasses user

authentication.

Without additional security measures, the attacker could access confidential information easily.

However, the attacker must also overcome the access

permission check.

Even if he succeeds, the third layer of defense, biometric verification, presents a significant obstacle.

Defense in Depth can also involve implementing fallback mechanisms to prevent total system compromise.

Instead of allowing an attacker to gain complete control or access, we can instruct the system to crash

or enter a secure, fail-safe mode.

By employing these various techniques, you create a layered security approach that increases the difficulty

for attackers to compromise your C and C++ applications.

Now, remember that Defense in Depth is not a one-size-fits-all solution, and its implementation depends

on the specific requirements and contexts of your application.

layered-authentication.cpp

```cpp
    if (!file)
    {
        cout << "Failed to open the encrypted file. Operation aborted." << endl;
        return;
    }

    // Decrypt sensitive data
    decrypt_sensitive_data();

    // Process the decrypted data
    // ...

    // Close the file
    file.close();
}

int main()
{
    // ...

    sensitive_operation();

    // ...

    return 0;
}
```

1. Authenticate User
2. Check Permissions
3. Verify Biometrics

Security Defense Layers


```
    cout << "Biometric verification failed. Access denied." << endl;
    return;
}
cout << "Biometric verification successful." << endl;

// Read encrypted file
ifstream file("encrypted_data.txt", ios::in | ios::binary);

if (!file)
{
    cout << "Failed to open the encrypted file. Operation aborted." << endl;
    return;
}

// Decrypt sensitive data
decrypt_sensitive_data();

// Process the decrypted data
// ...

// Close the file
file.close();
}
```

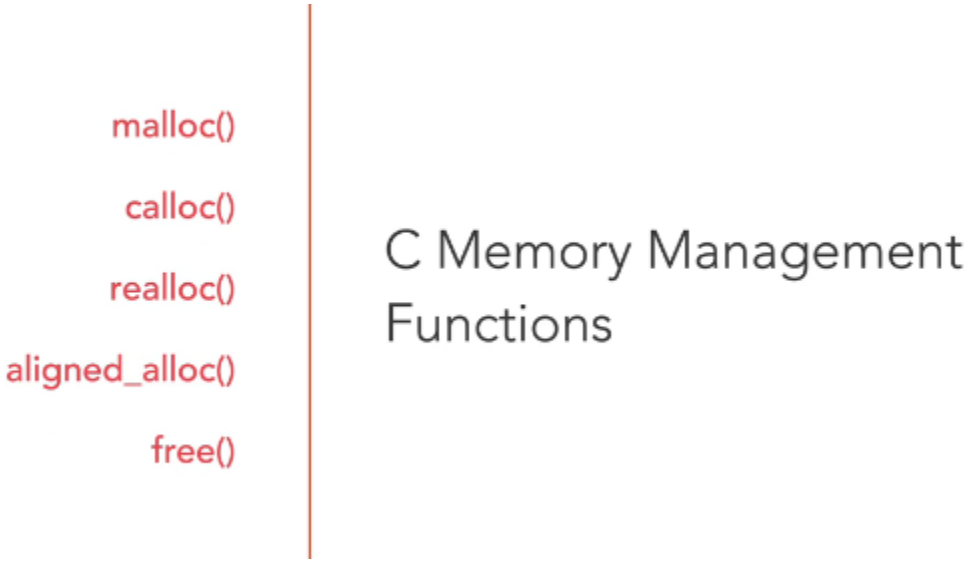
4. Secure Memory Management in C and C++

4.1 C Memory Management

Memory management is a fundamental aspect of programming, and understanding it is crucial for writing secure and reliable code.

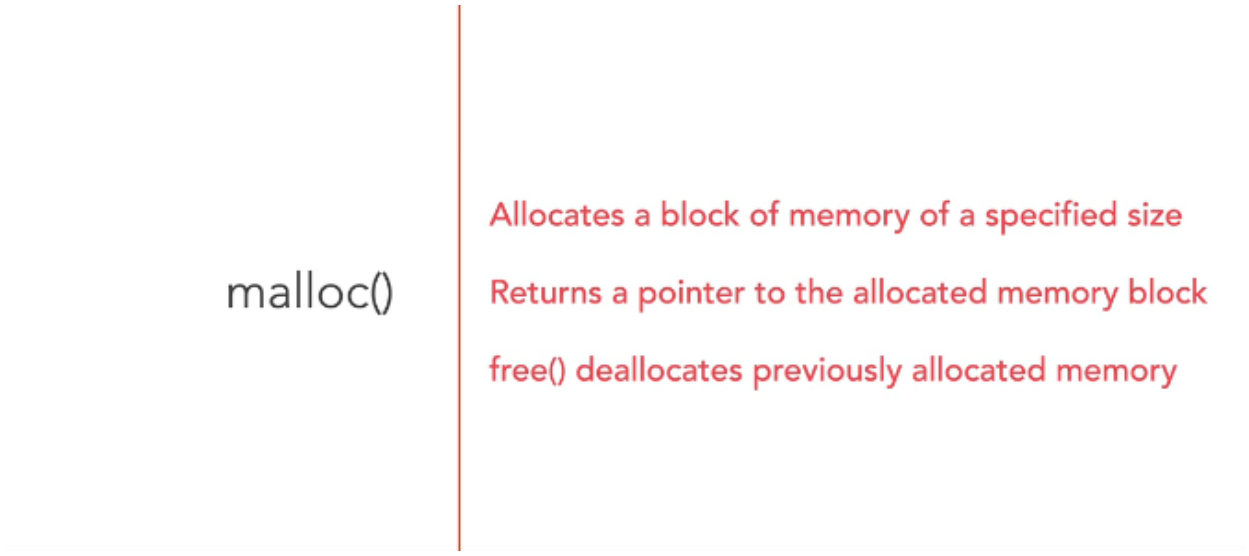
Today we'll focus on the basics of memory management in the C programming language.

Before we dive into the details, let's briefly discuss some of the standard functions used for memory management in C. These functions provide a way to dynamically allocate and deallocate memory during program execution.



The most commonly used functions include malloc(), calloc(), realloc(), aligned_alloc() and free().

The malloc() function is used to allocate a block of memory of a specified size in bytes.



It returns a pointer to the allocated memory block which can be used to store data.

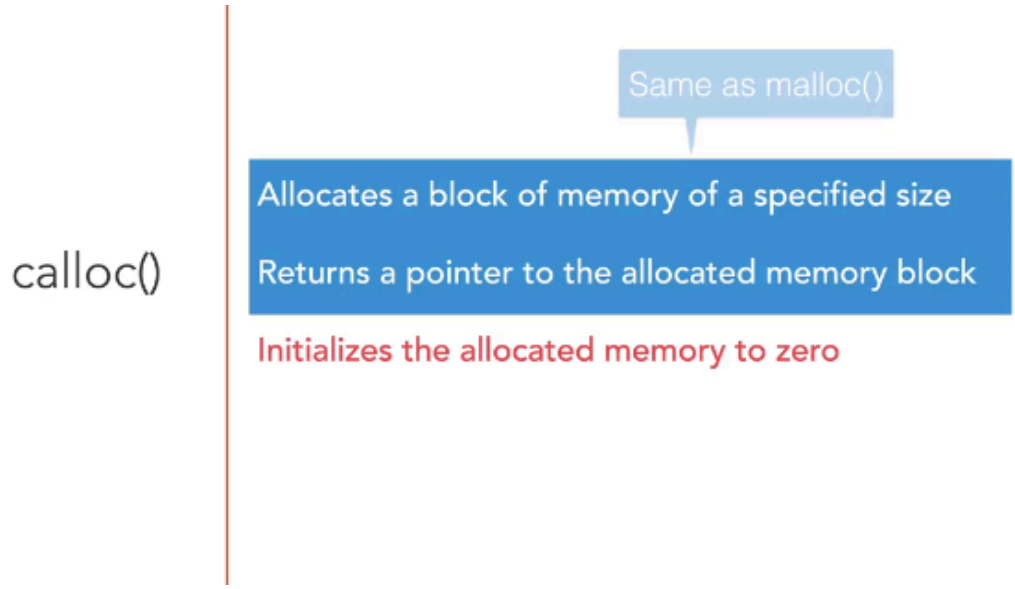
The free() function is used to deallocate memory that was previously allocated.

c-memory-management.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    // Allocating memory for an array of 5 integers using malloc
    int *numbers = (int *)malloc(5 * sizeof(*numbers));
    if (numbers != NULL)
    {
        // Memory allocation successful
        // Use the allocated memory for operations
        //...
        // Deallocating the memory block
        free(numbers);
    }
```

It's important to release memory that is no longer needed to avoid memory leaks.

The calloc() function is similar to malloc(), but it also initialises the allocated memory to zero.



This can be useful when working with arrays or data structures that need to be initialised.

```c
// Allocating and initializing memory for a character buffer using calloc
char *buffer = (char *)calloc(100, sizeof(*buffer));
if (buffer != NULL)
{
    // Memory allocation successful
    // Use the allocated memory for operations
    //...
    // Deallocating the memory block
    free(buffer);
}
```

The realloc() function is used to resize an existing memory block.

realloc()

Resizes an existing memory block

Takes a pointer and the new size in bytes

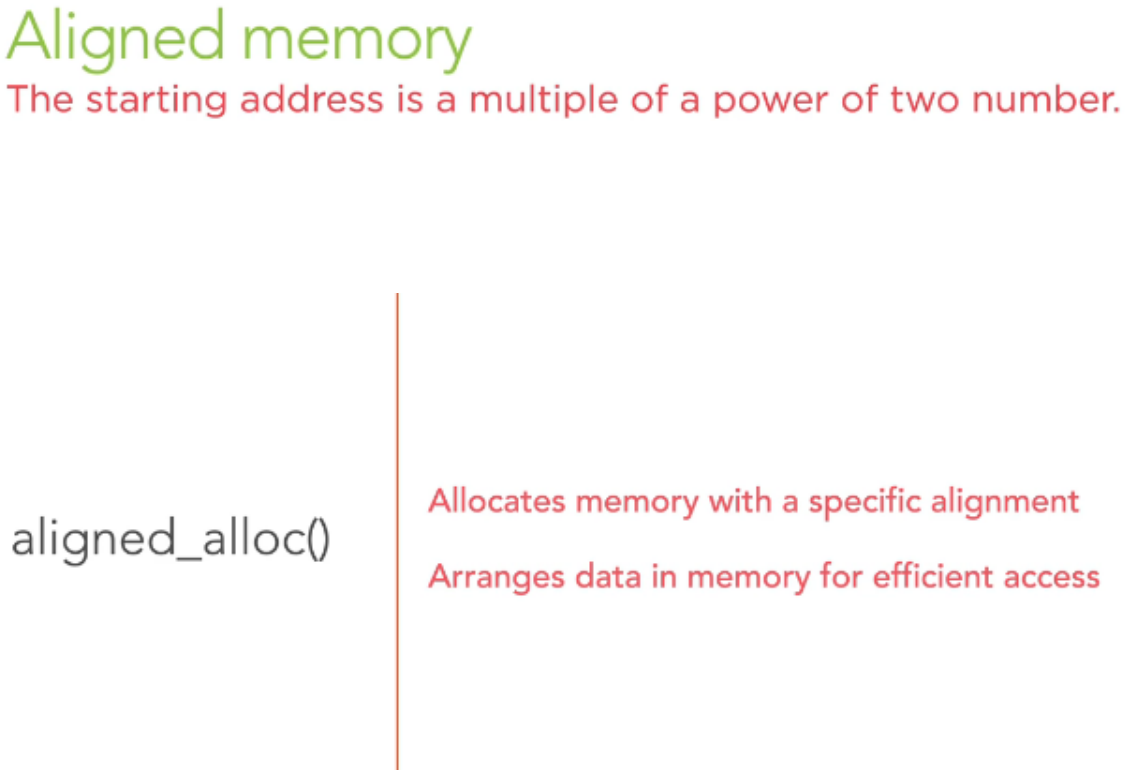
Can adjust the size of a data structure dynamically

It takes a pointer to the original block, the new size in bytes, and returns a pointer to the resized block.

```c
// Allocating memory for an array of 5 integers
int *data = (int *)malloc(5 * sizeof(*data));
if (data != NULL)
{
    // Memory allocation successful
    // ...

    // Resizing the array to accommodate 10 integers
    int *temp = (int *)realloc(data, 10 * sizeof(*data));
    if (temp == NULL)
    {
        // Memory reallocation failed
        printf("realloc failed with errno = %d\n", errno);
        free(data); // Free the original memory
    }
    else
    {
        // Memory reallocation successful
        data = temp; // Assign the new memory to data
        // Use the resized memory for operations
        //...
        // Deallocating the memory block
        free(data);
    }
```

This function can be helpful when you need to dynamically adjust the size of an array or data structure.

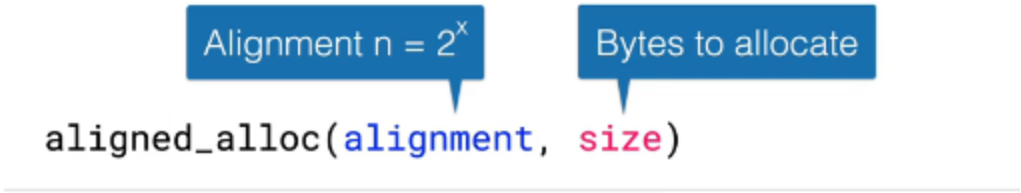
The aligned_alloc function is useful when you need to allocate memory with a specific alignment.



This can be important in scenarios where certain hardware or platform specific requirements demand data to be aligned in memory for efficient access.

Aligned memory refers to the concept of allocating memory such that the starting address of the allocated block is a multiple of a particular value n, where n is a power of two.

The aligned_alloc() function has two parameters: alignment and size.



'alignment' specifies the alignment and must be a power of two.

'size' represents the number of bytes to allocate. It should be an integral multiple of 'alignment'.

Here's an example of how to use aligned_alloc().

```
    // Allocating an 8-byte aligned memory block of 64 bytes
    void *hardwareBuffer = aligned_alloc(8, 64);
    if (hardwareBuffer != NULL)
    {
        // Memory allocation successful
        printf("8-byte aligned addr: %p (%ld decimal)\n", hardwareBuffer, (long)hardwareBuffer);
        // Use the aligned memory for operations
        //...

        // Deallocating the memory block
        free(hardwareBuffer);
    }
    else
    {
        // Memory allocation failed
        printf("aligned_alloc failed with errno = %d\n", errno);
        return -1;
    }

    return 0;
}
```

In this example, we use the aligned_alloc() function to allocate 64 bytes of memory with 8-byte alignment.

If the memory allocation succeeds, it returns a pointer to the beginning of the allocated memory.

This address will then be printed to the terminal in both hexadecimal and decimal format.

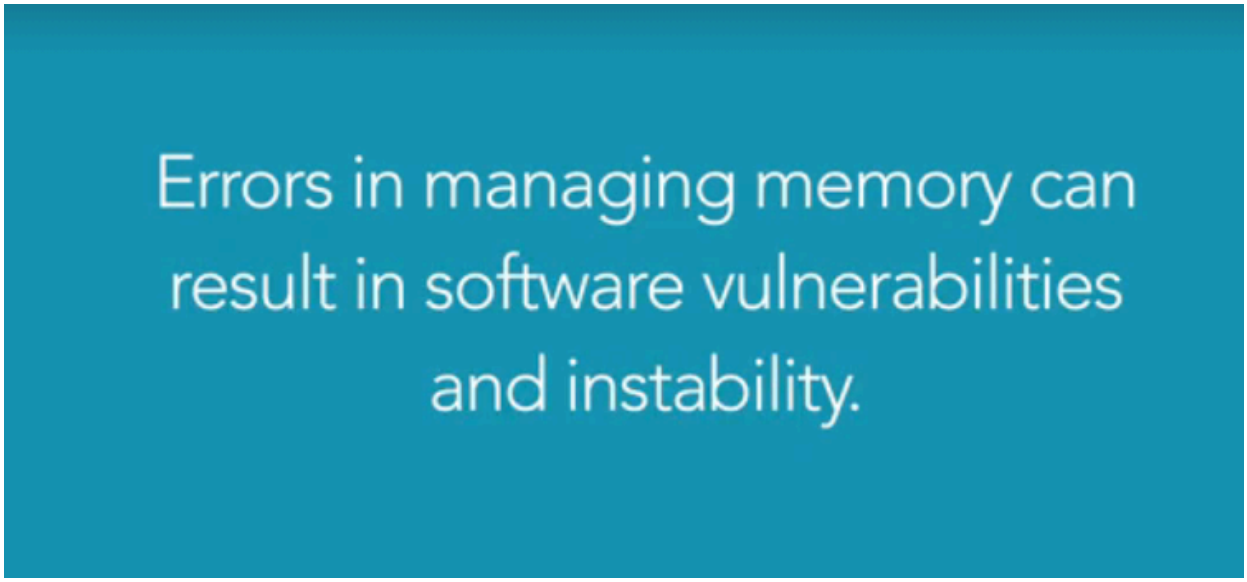
This will allow us to confirm whether it's an 8-byte aligned memory address.

4.2 C Memory Management Mistakes - Part 1

Now that we've covered the standard memory management functions in C, let's discuss some of the common errors that can occur during memory management.

These errors can lead to vulnerabilities and instability in our programs.

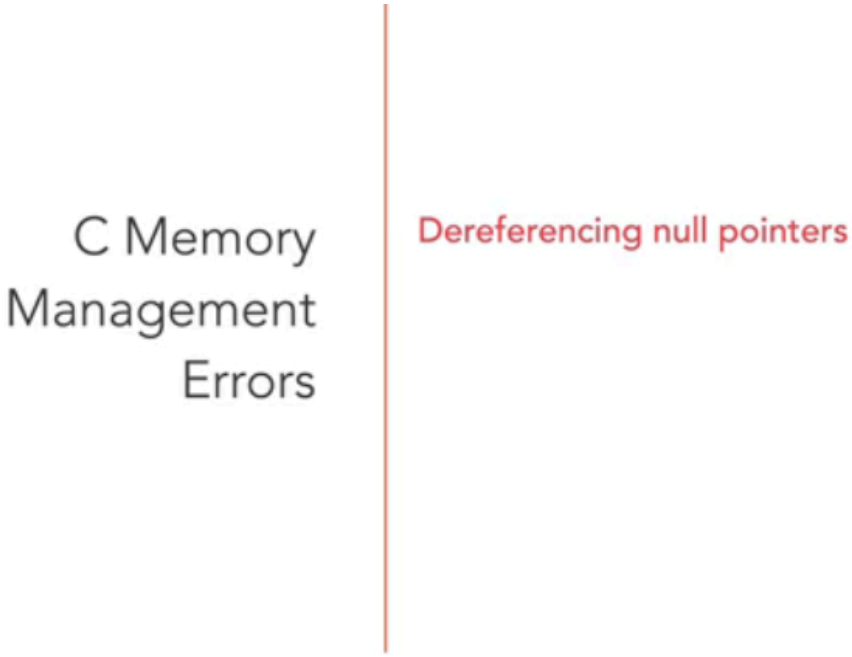
It's important to be aware of these pitfalls and follow best practices to ensure secure memory management.



We'll start with the more basic memory management errors and gradually uncover the more sophisticated ones.

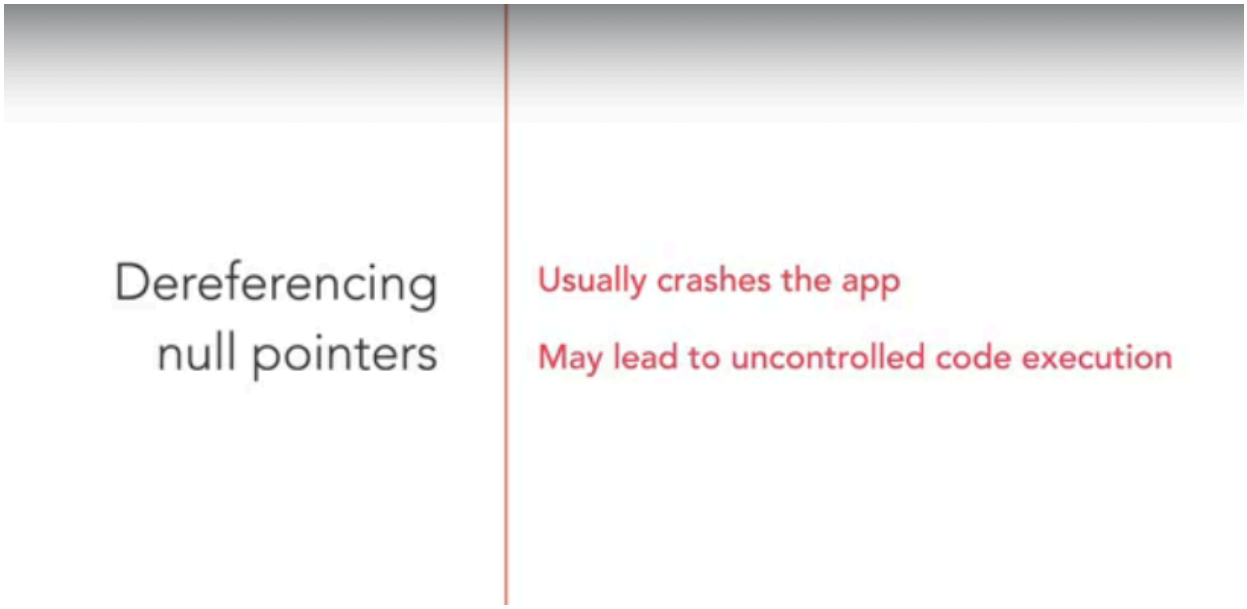
> I've always heard them used in the opposite sense:
>
> - `&` is the reference operator -- it gives you a reference (pointer) to some object
> - `*` is the dereference operator -- it takes a reference (pointer) and gives you back the referred to object;

Dereferencing null pointers.



A null pointer is a pointer that does not point to any valid memory address.

Dereferencing such a pointer can lead to undefined behaviour and crashes.



Take a look at this code snippet.

```
int* ptr = NULL;
*ptr = 10;// this will cause a runtime error
```

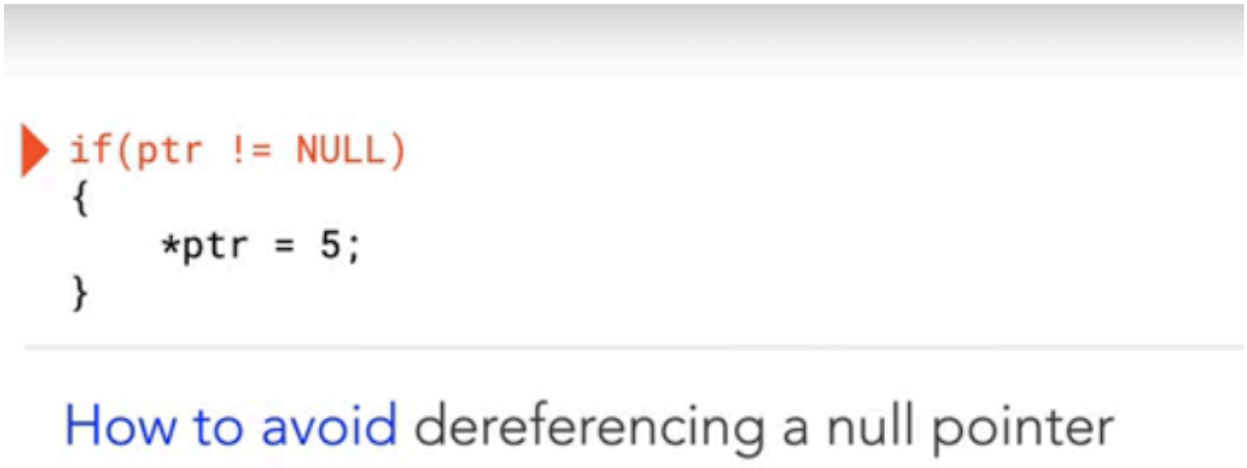
Dereferencing a null pointer

In this example, we initialise the pointer to null and then we attempt to assign the value of 10 to
the memory location it points to. Since it's a null pointer, the assignment is invalid and will typically result in a segmentation fault.

However, certain computers and embedded systems have memory and registers mapped at address zero, leading to unexpected outcomes when overwriting their content.

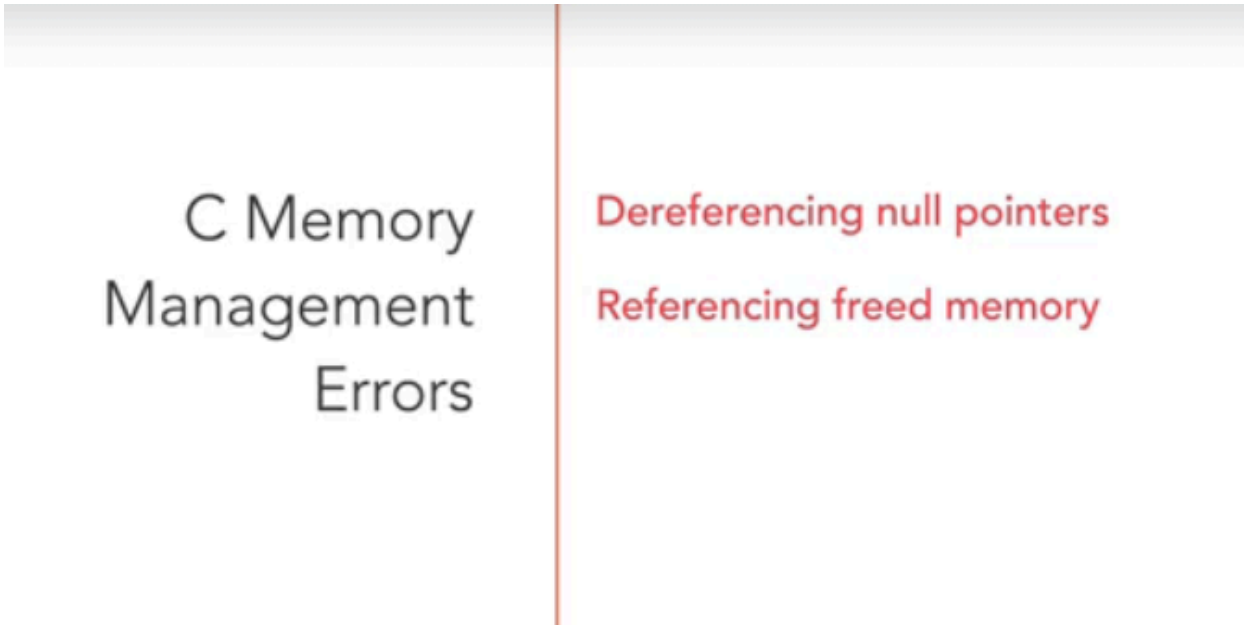
In certain cases, this can even result in unintended execution of arbitrary code, creating serious
security vulnerabilities.

The simplest way to avoid dereferencing null pointers is to check whether the pointer is null before accessing its memory.

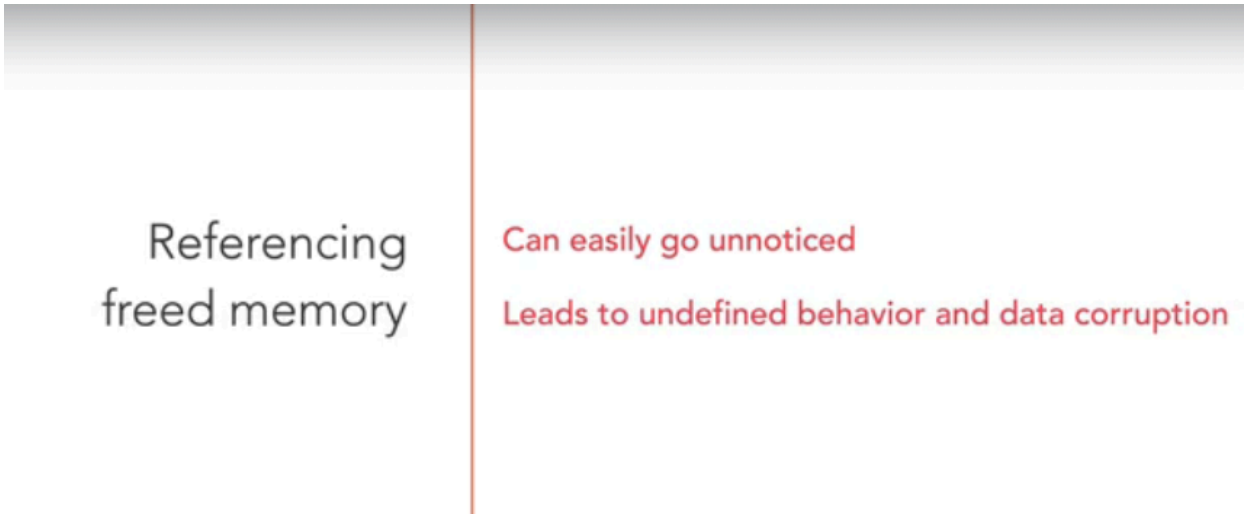


Referencing freed memory.

Next we have referencing freed memory,



A common mistake that often goes unnoticed. When we free memory using the free() function, we must not access that memory anymore.



In the following code, we allocate memory for an integer and store its address in 'value'.

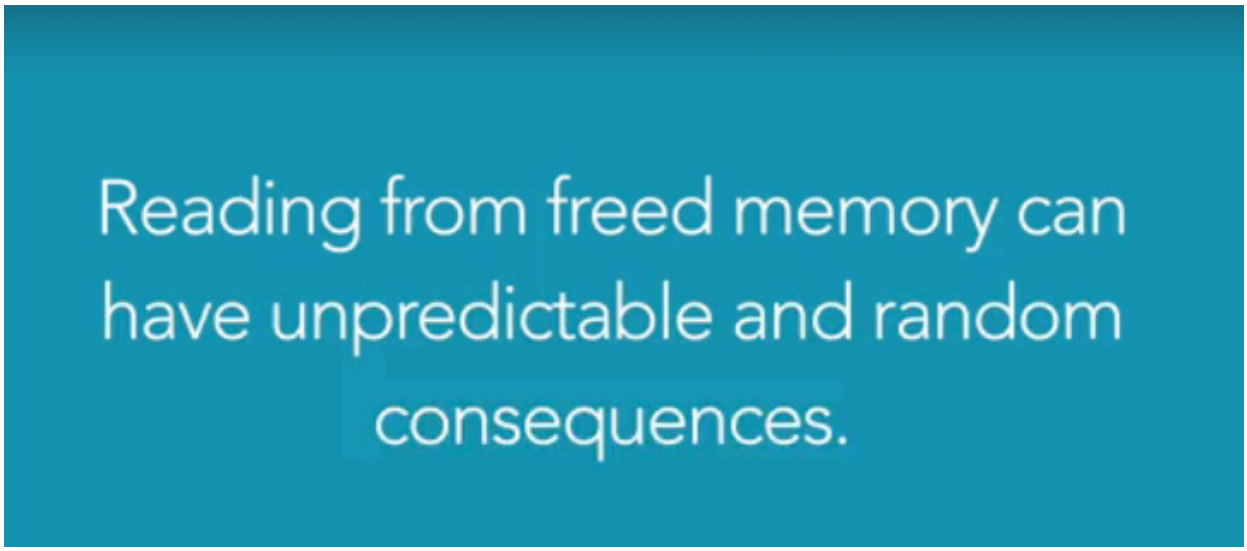
```
int *value = malloc(sizeof(int));
free(value);

// ... Do some operations

// forget that value was freed and try to use it
*value = 5;
```

Referencing freed memory

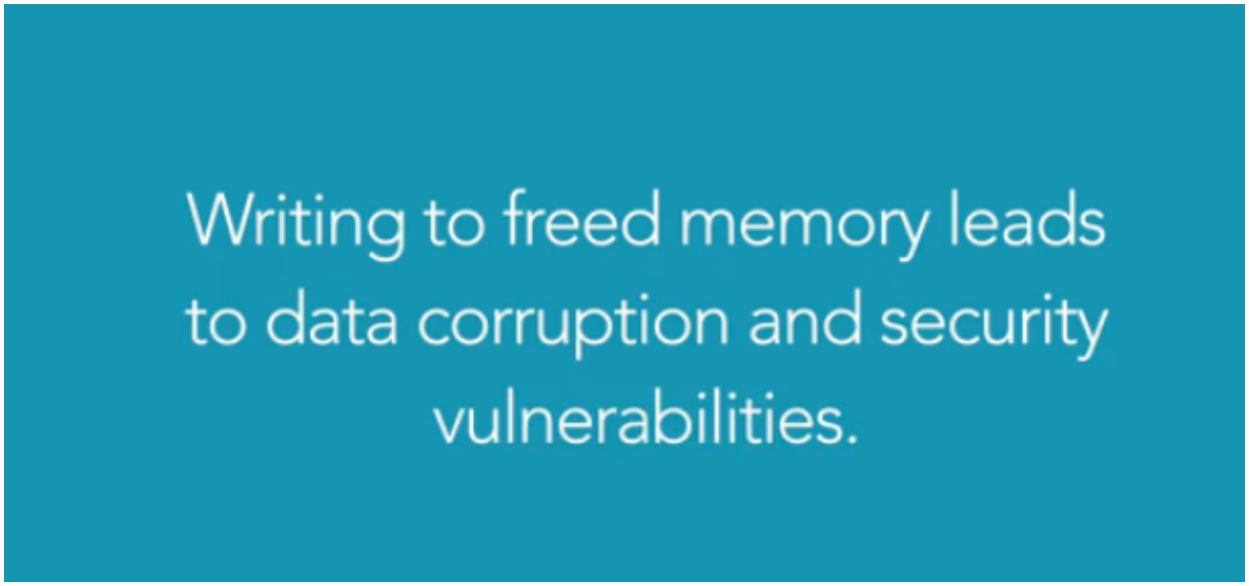
Later, we free the memory. However, we mistakenly try to write the value 5 to the now freed memory. This leads to undefined behaviour and data corruption.

Even though reading from memory that has been freed may not result in a memory fault, there is a chance that the memory has already been reallocated and its contents are entirely different.

Now, this can cause errors that are not detected during testing but surface during actual operation.



Similarly, writing to memory that has been freed may not result in a memory fault immediately, but
It can create various issues.



If the memory has been reallocated, the program's data can be overwritten, exposing it to potential security threats.

To prevent such problems, it is crucial to avoid referencing freed memory.

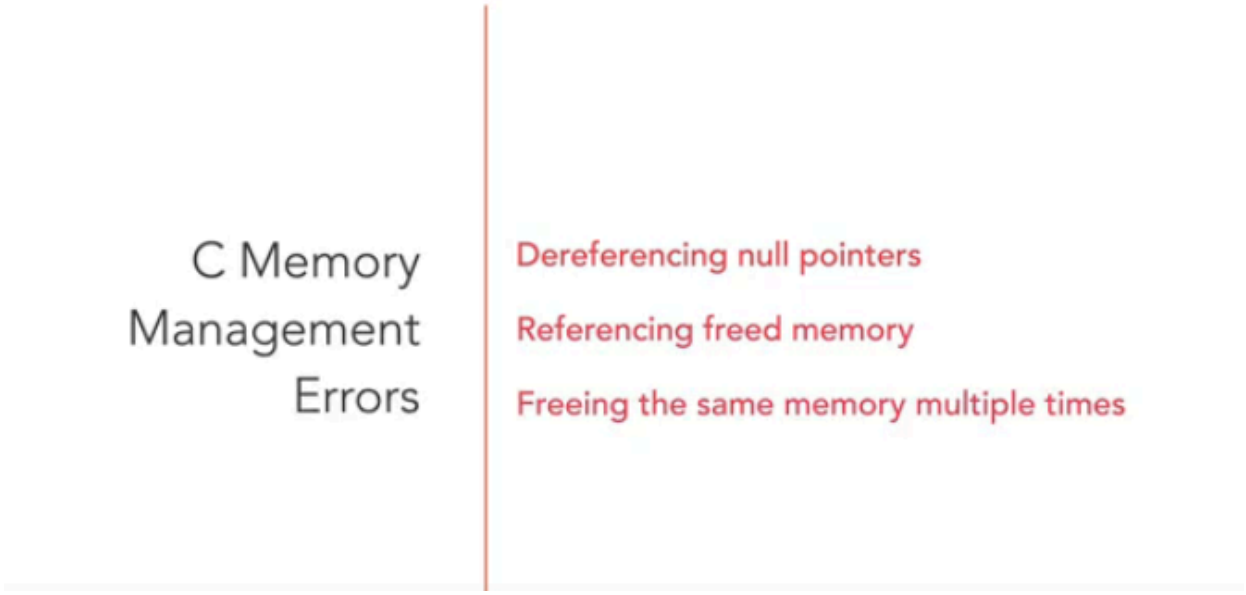
Implementing defensive coding practices can help reduce the risk of accessing freed memory.

For example, setting pointers to null after freeing them can help identify attempts to access freed memory by causing a crash.

```
int *value = malloc(sizeof(int));
free(value);
value = NULL;// set to NULL after calling free
// ... Do some operations

// forget that value was freed and try to use it
*value = 5; // this will cause a runtime error
```

How to prevent referencing freed memory

Freeing the same memory multiple times.



Another mistake is freeing the same memory block more than once. When we free memory, it becomes available for reuse by the system.

Attempting to free the same memory block again can lead to undefined behaviour, memory corruption and security vulnerabilities that can be exploited.

In this code, we allocate memory for an array of integers and then free it.

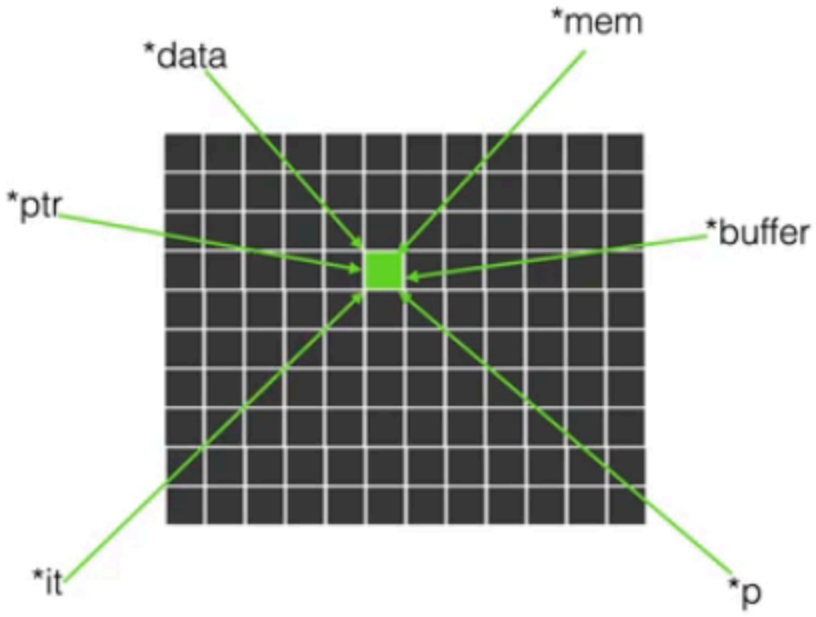
```
int *array = malloc(5 * sizeof(int));
free(array);
// ... Do some operations

// forget that array was freed and deallocate
free(array);
```

Freeing the same memory multiple times

However, we mistakenly call free() again, which will trigger a runtime error--

In other words, it will crash the app.

Now, while I made the error in this example quite obvious, it's easy to accidentally free the same
memory chunk multiple times, and it can be hard to spot. In large and complex codebases,
it becomes challenging to track the flow of memory management accurately.



Multiple pointers or variables might point to the same memory address. If one of these pointers is deallocated using free(), other pointers may still point to the same memory location.

If the programmer deallocates the memory again using another pointer, it leads to freeing the same memory chunk multiple times.

Error handling routines might free memory upon encountering certain conditions or exceptions.

These error-handling paths can become complicated, and if not carefully designed, they may unintentionally free memory that has already been deallocated.

To avoid this problem, set freed pointers to null. Calling free on a null pointer is safe and has
no effect: it simply does nothing and returns without any operation.

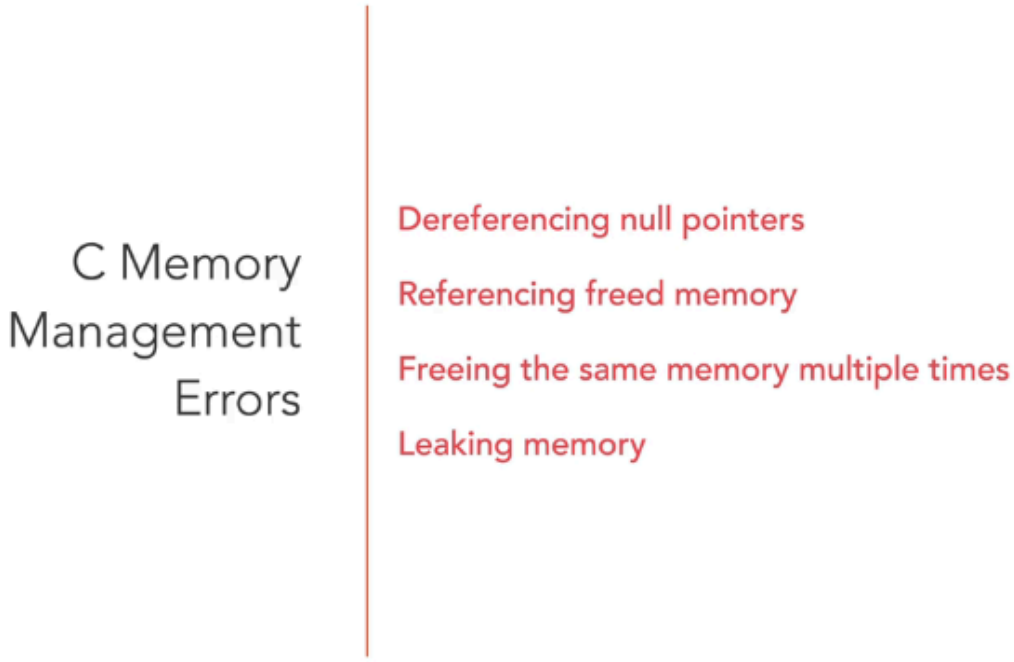
```c
int *array = malloc(5 * sizeof(int));
free(array);
array = NULL;
// ... Do some operations

// forget that array was freed and deallocate it again
free(array); // array is NULL, and free(NULL) is a no-op
```

Prevent undefined behavior when attempting to free the same memory multiple times

This behaviour is specified by the C standard and serves as a defensive measure to prevent undefined behaviour when attempting to free an invalid or uninitialized pointer.

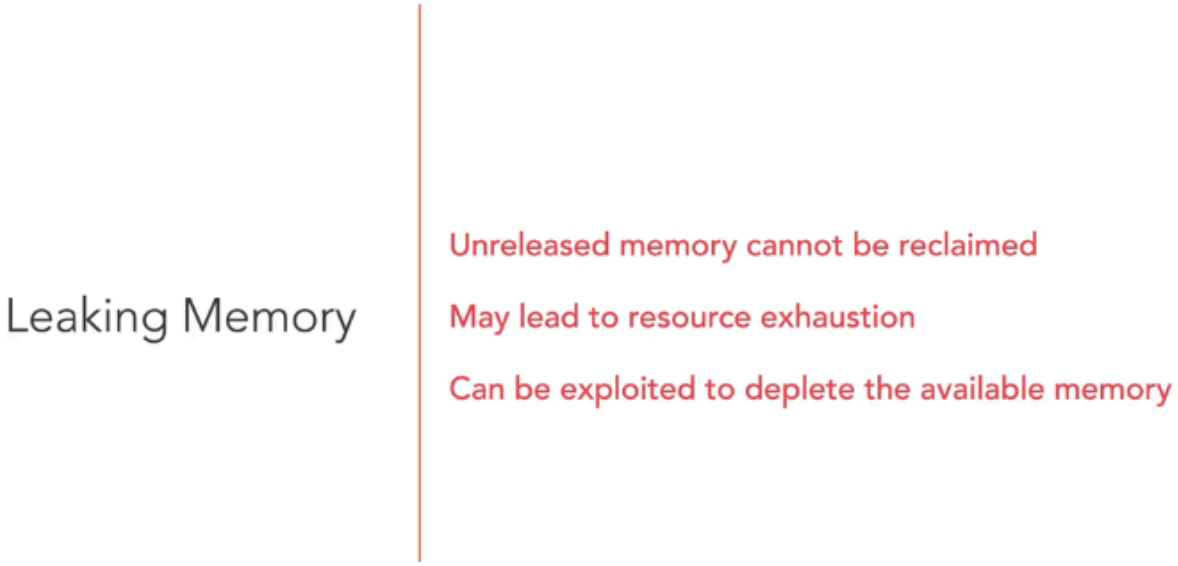
4.3 C Memory Management Mistakes - Part 2

Memory leaks occur when we fail to release dynamically allocated memory after it is no longer needed.



This unreleased memory cannot be reused by the program. Over time, memory leaks can cause resource exhaustion.

The program keeps using more and more memory without releasing it, which ultimately results in the system running out of available memory.



Attackers can exploit memory leaks to launch resource exhaustion attacks on a system by repeatedly requesting memory allocations without freeing them.

Attackers can quickly deplete the available memory, causing the system to become unresponsive or crash.

This can be particularly damaging in systems that run continuously, such as servers or embedded devices.

Here's a code example that demonstrates a subtle memory leak by allocating memory in a loop.

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

int main()
{
    const int n = 10; // Number of iterations
    char *buffer;

    for (int i = 0; i < n; ++i)
    {
        // malloc called multiple times without a corresponding free
        buffer = (char *)malloc(100 * sizeof(*buffer)); // Allocate memory for 100 characters
        // Do some operations with the allocated memory
        sprintf(buffer, "This is iteration %d\n", i);
        printf("%s", buffer);
    }

    free(buffer);

    return 0;
}

```

In this example, I allocate memory for 100 characters in the loop and use sprintf() to populate the buffer with a string indicating the current iteration number. The program then prints the contents of the buffer. After exiting the loop, I call free() to release the memory allocated for the buffer.

Do you see any potential issues with this code?

There are various memory leak detection tools available. The program has nine leaks for 1008 total leaked bytes.

Can you help me identify the issue?

Upon closer inspection, it seems that we are allocating memory within the loop, but the free() function is being called outside of the loop.

This means that only the last allocated block of memory is being released and the first nine allocations are causing memory leaks.

To fix the memory leaks,

we need to move the free(buffer) statement inside the loop to release each allocated block of memory after using it.

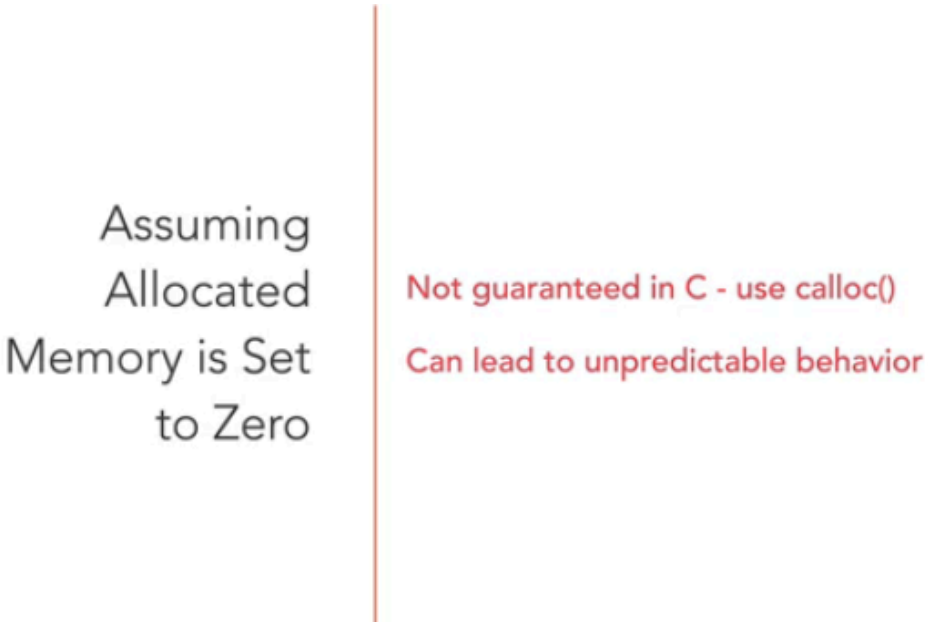
So let's do it.

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

int main()
{
    const int n = 10; // Number of iterations
    char *buffer;

    for (int i = 0; i < n; ++i)
    {
        // malloc called multiple times without a corresponding free
        buffer = (char *)malloc(100 * sizeof(*buffer)); // Allocate memory for 100 characters
        // Do some operations with the allocated memory
        sprintf(buffer, "This is iteration %d\n", i);
        printf("%s", buffer);

        free(buffer);
    }

    return 0;
}
```

Awesome!

4.4 Beyond the Basics: Subtle C Memory Management Errors

The code snippet allocates memory for an integer array with arraySize elements using the malloc() function.



However, it makes an erroneous assumption that the memory contents are initially set to zero.

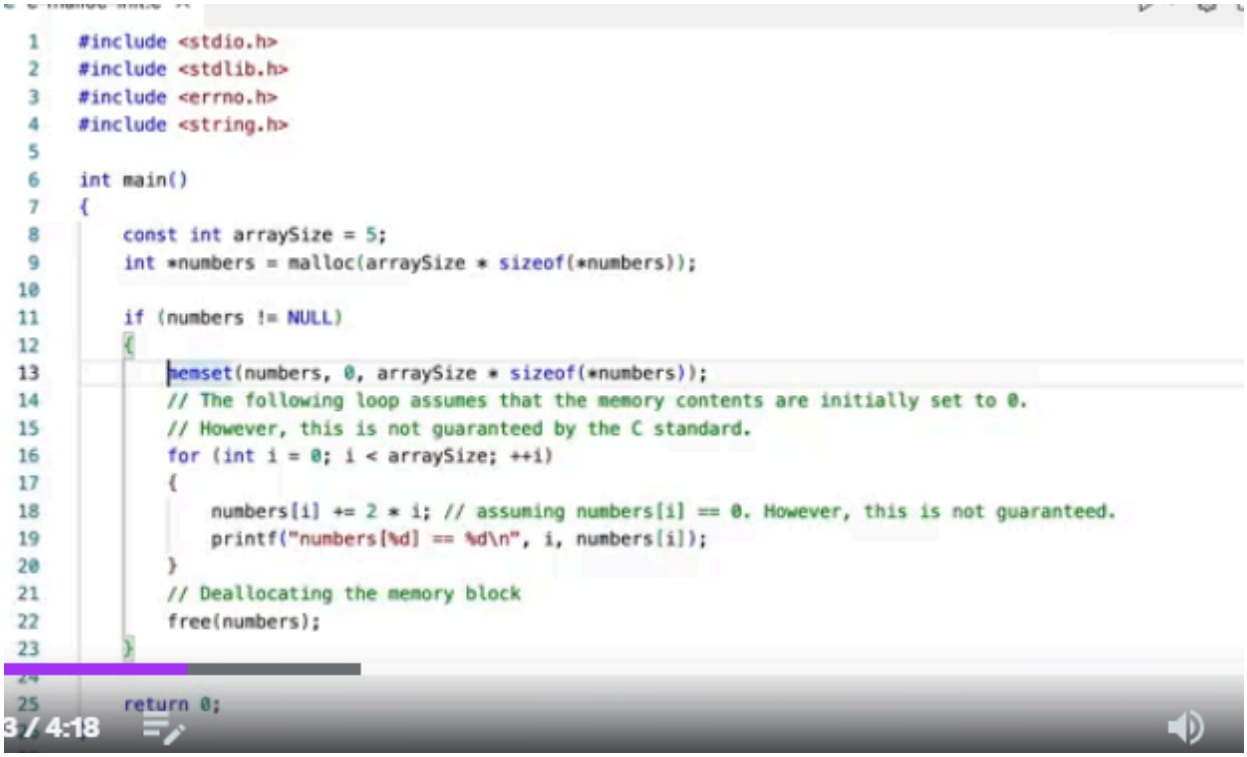
The subsequent loop performs arithmetic operations on the elements, assuming that their initial values are zero.

c-malloc-init.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    const int arraySize = 5;
    int *numbers = malloc(arraySize * sizeof(*numbers));

    if (numbers != NULL)
    {
        // The following loop assumes that the memory contents are initially set to 0.
        // However, this is not guaranteed by the C standard.
        for (int i = 0; i < arraySize; ++i)
        {
            numbers[i] += 2 * i; // assuming numbers[i] == 0. However, this is not guaranteed.
            printf("numbers[%d] == %d\n", i, numbers[i]);
        }
        // Deallocating the memory block
        free(numbers);
    }

    return 0;
}
```

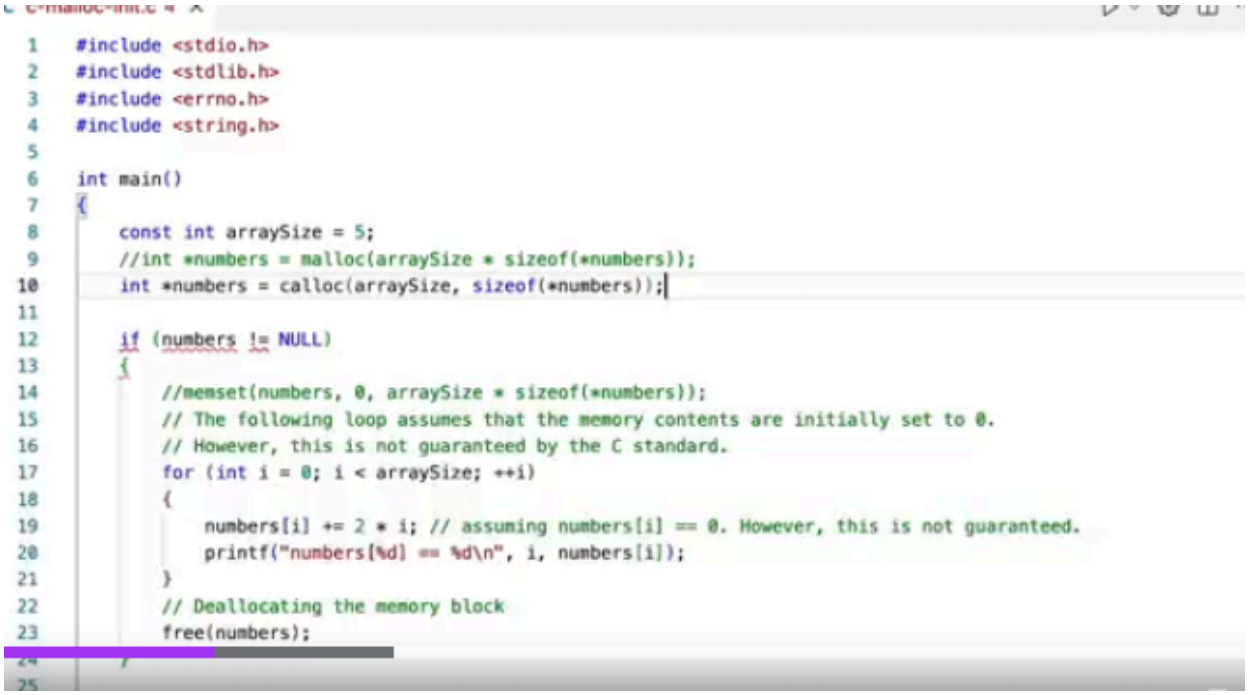
However, this is not guaranteed by the C standard, and the initial contents of the allocated memory are undefined.

To ensure correct behaviour, we could explicitly initialise the memory using the memset() function.

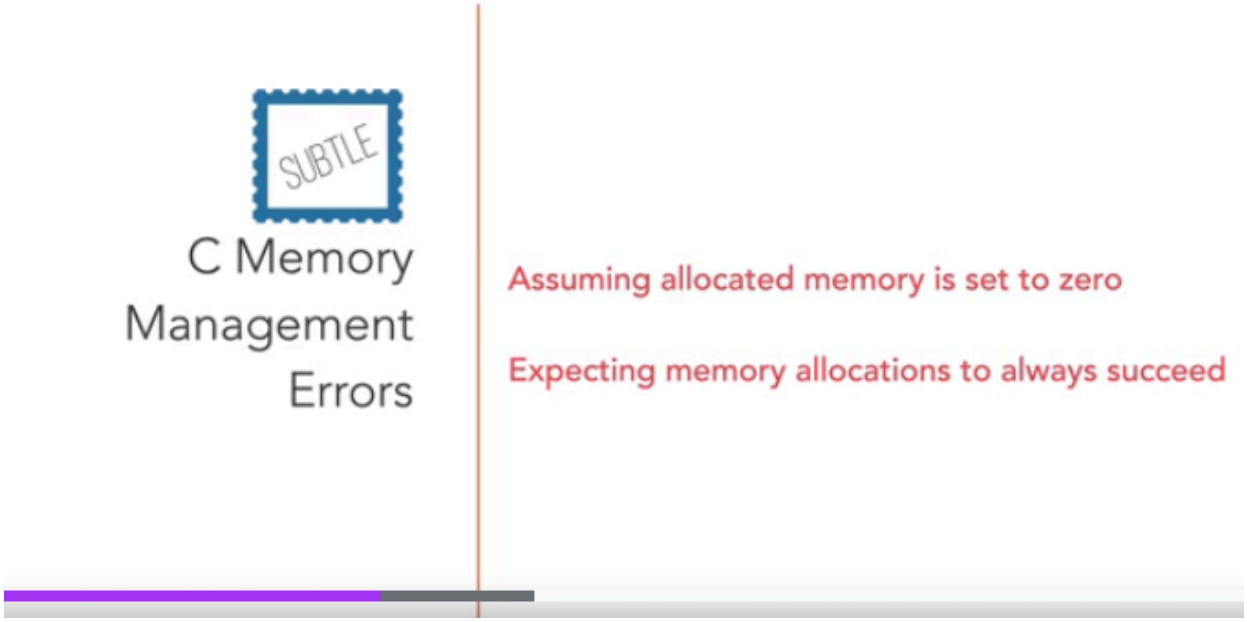


Here's an improved version of the code that explicitly initialises the memory before using it.

Or even better, we can use calloc() to fix the issue with even less typing.

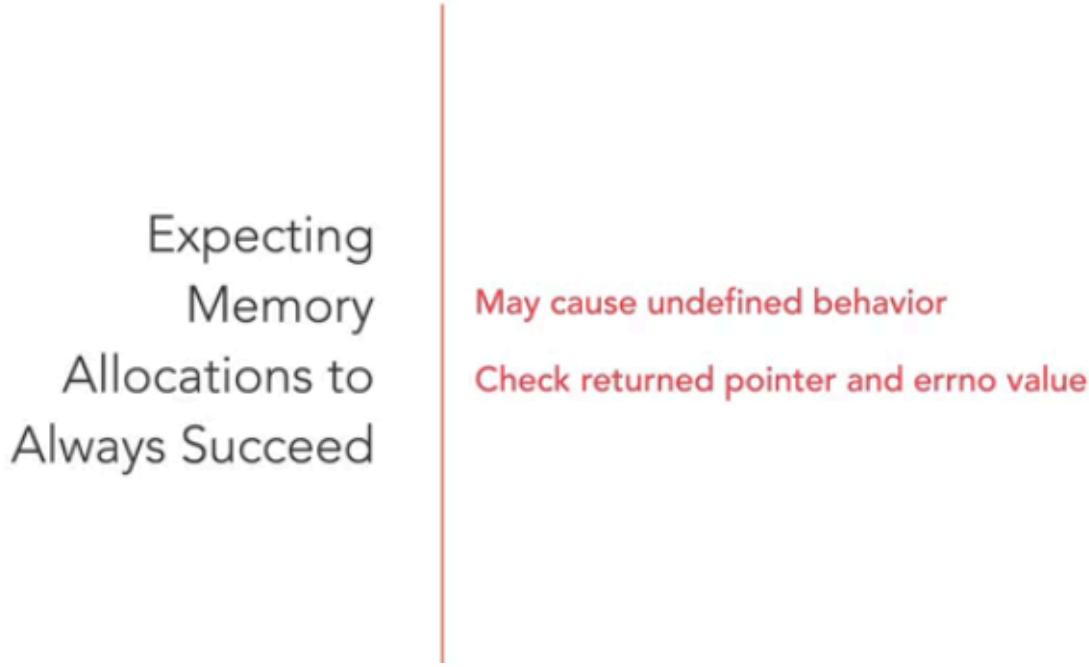


Assuming memory allocation is successful.



Here's the last, less obvious memory management mistake I want to discuss in this video.

We might be tempted to believe that memory allocation always succeeds since modern computers have a large amount of physical memory.

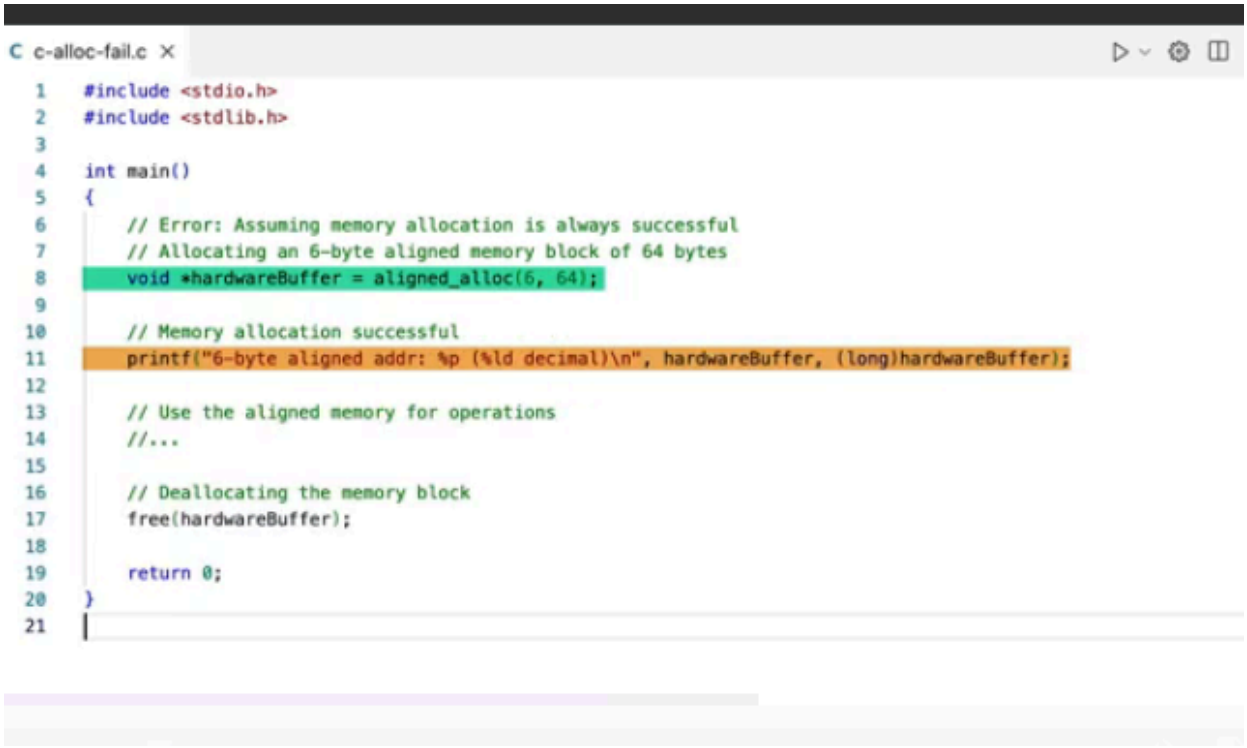


Yet, having a lot of RAM doesn't guarantee that memory will be readily available whenever we require it.

Neglecting to check if the allocation succeeded will result in undefined behaviour or crashes when trying to use the invalid pointer.

If the allocation fails, these functions will return NULL and set an appropriate error code in the errno variable, which can provide helpful information about the issue.

In the following code, we'll call aligned_alloc() to allocate 6-byte aligned memory, print out the returned address, and free the allocated memory when it's no longer needed.



Unfortunately, running this code will print out zero to the console.

What's going on here?

Let's find out.

I'll add a null pointer check first.

If aligned_allock() returned a null pointer,

I'll print out the value of the errno variable.

This extra information could help us in identifying the problem. But first, I'll need to include the errno header file.

Let's run the program.

c-alloc-fail.c 1

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    // Error: Assuming memory allocation is always successful
    // Allocating an 6-byte aligned memory block of 64 bytes
    void *hardwareBuffer = aligned_alloc(6, 64);

    if (hardwareBuffer != NULL)
    {
        // Memory allocation successful
        printf("6-byte aligned addr: %p (%ld decimal)\n", hardwareBuffer, (long)hardwareBuffer);

        // Use the aligned memory for operations
        //...

        // Deallocating the memory block
        free(hardwareBuffer);
    }
    else
    {
        printf("aligned_alloc failed with errno = %d\n", errno);
        return -1;
    }
```

The terminal output tells us that aligned_alloc() failed with **errno 22.**

We can check what this value means in the errno header file.

So, 22 means EINVAL--

Invalid argument. aligned_alloc() returns this error code when the alignment argument is not a power of two.

It's clear that 6 is not a power of two, so I'll use the value 8 instead.

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main()
{
    // Error: Assuming memory allocation is always successful
    // Allocating an 6-byte aligned memory block of 64 bytes
    void *hardwareBuffer = aligned_alloc(8, 64);

    if (hardwareBuffer != NULL)
    {
        // Memory allocation successful
        printf("6-byte aligned addr: %p (%ld decimal)\n", hardwareBuffer, (long)hardwareBuffer);

        // Use the aligned memory for operations
        //...

        // Deallocating the memory block
        free(hardwareBuffer);
    }
    else
    {
        printf("aligned_alloc failed with errno = %d\n", errno);
        return -1;
```

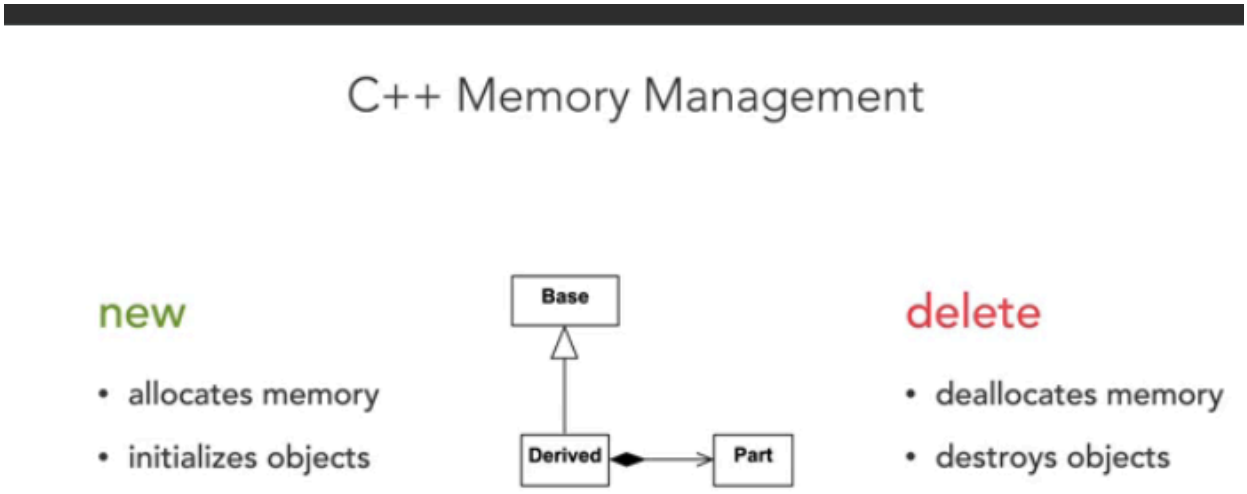
Let's also update the debug message.

And now, the program runs without issues, and displays the 8-byte aligned address.

4.5 C++ Memory Management: new and delete

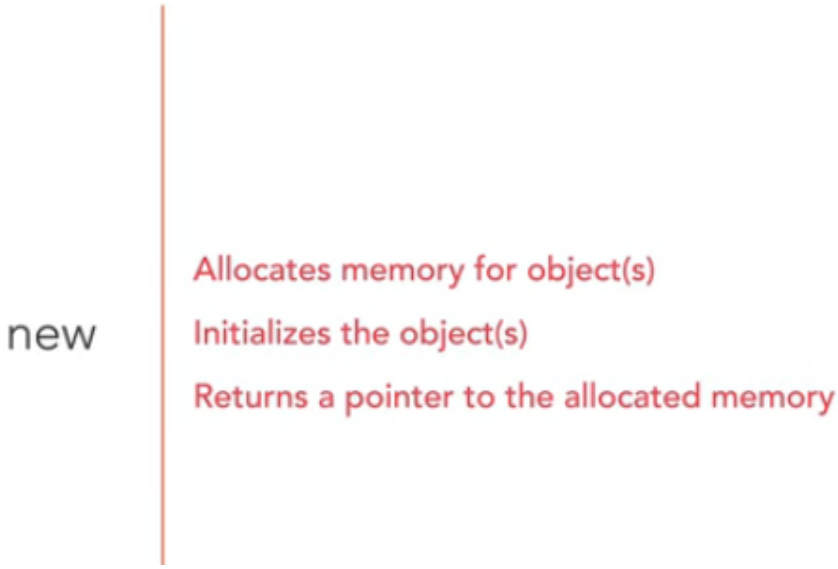
C++ replaced C's memory management functions with two powerful operators: new and delete.

These operators not only allocate and deallocate memory, but also initialise and destroy objects.



This makes it more convenient to work with classes, objects, and inheritance in an object-oriented programming paradigm.

Let's start by exploring the new operator. In C++, new serves two main purposes:



it allocates enough memory to hold an object of a specified type or an array of objects, and it initialises the objects if necessary.

When used with a single object, new allocates memory to store that object and calls the object's constructor to initialise it. Similarly, when used with an array of objects, new allocates enough memory to hold the entire array and invokes the constructors for each element to initialise the array.

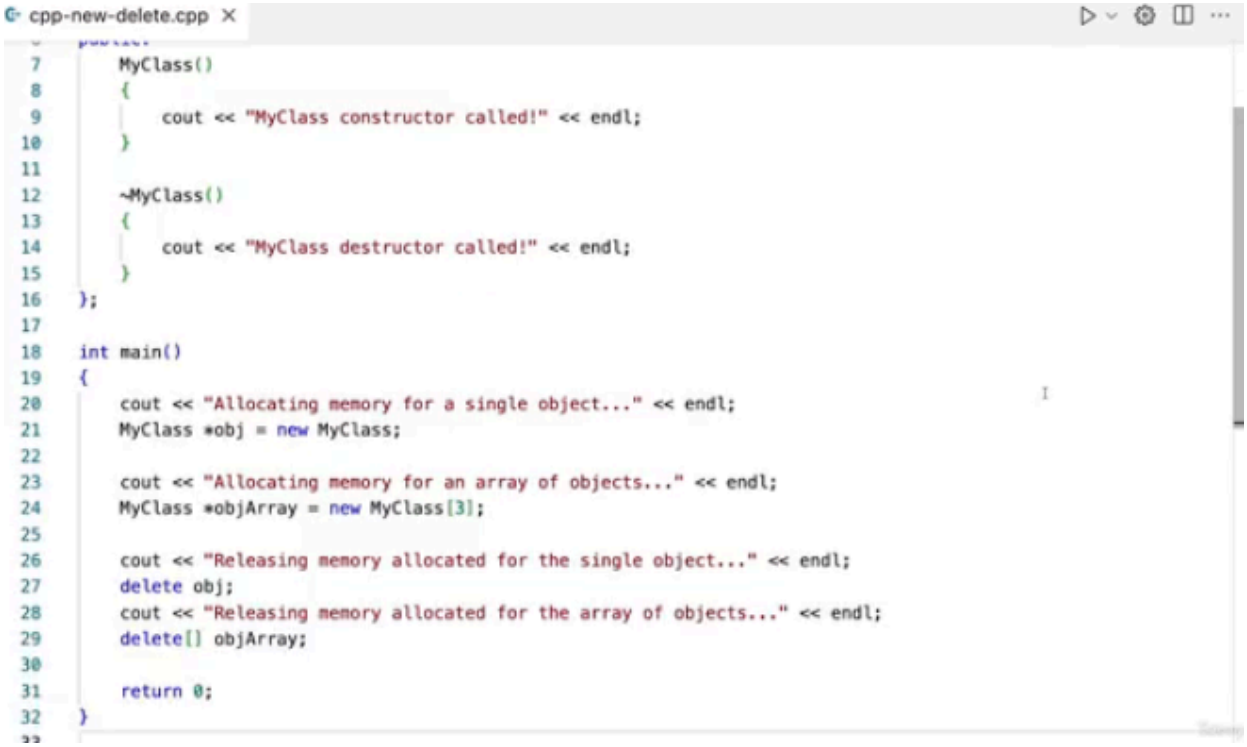
The new operator returns a pointer to the newly allocated memory, pointing to the created object or the first element of the array in case of arrays.

Memory allocated with the new operator has a dynamic storage duration.

This means that the memory will persist beyond the scope in which it was allocated until it gets explicitly released using delete.

Let's take a look at some code examples to solidify your understanding.

In this example, we create a MyClass object using new, which allocates memory and calls the constructor to initialise the object.



We also create an array of three MyClass objects using new[], which allocates enough memory for the array and calls the constructors for each element to initialise the array.

To ensure proper memory management, we use the delete operator to deallocate the single object and delete[] to deallocate the array of objects.

This way, we prevent memory leaks and guarantee that the destructors are called to clean up resources associated with the objects.

4.6 C++ Memory Management: Initialization and Placement new

Now let's talk about initialising objects with new.

When you use new, memory is allocated, and objects are constructed and initialised if you provide constructor arguments.

For example, in the following code snippet, we pass 42 as an argument when creating a single object.

```cpp
#include <iostream>
using namespace std;

class MyClass
{
public:
    MyClass()
    {
        cout << "MyClass constructor called!" << endl;
        data = 0;
    }

    explicit MyClass(int value) : data(value)
    {
        cout << "Constructing MyClass object with value: " << value << endl;
    }

    ~MyClass()
    {
        cout << "MyClass destructor called!" << endl;
    }

private:
    int data;
};

int main()
```

new-init-placement-new.cpp

```cpp
int main()
{
    // Single object allocation and initialization
    cout << "Allocating memory for a single object with value..." << endl;
    MyClass *objWithValue = new MyClass(42); // Constructor with parameter called

    cout << "Allocating memory for a single object..." << endl;
    MyClass *obj = new MyClass; // Default constructor called

    // Don't forget to release the memory to avoid memory leaks
    cout << "Releasing allocated memory..." << endl;
    delete obj;
    delete objWithValue;
```

If no initialization parameters are provided, the object is default initialised.

Primitive built-in types, such as int, are not initialised unless we call new with an empty initializer. Otherwise, their initial value is undefined.

```
4
5       // Allocating memory and initializing built-in types
6       int *i = new int;   // uninitialized
7       int *j = new int(); // calling empty new initializer -> guaranteed to be initialized to 0
8
9       cout << "i: " << *i << endl;
0       cout << "j: " << *j << endl;
1
2       delete i;
3       delete j;
4
5
6
7
8
9
0
1
2
3
4
5
```

The same applies to Plain Old Data Type (POD) objects as demonstrated in the following example.

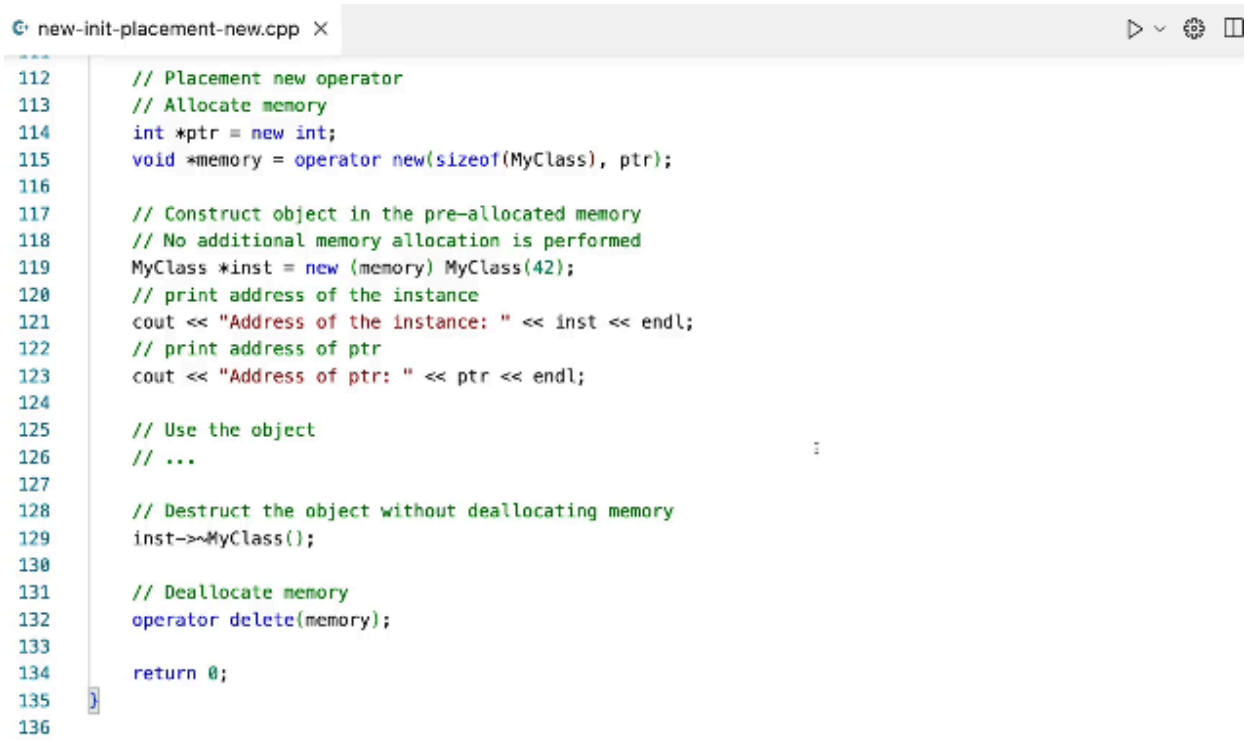
```
// POD type
struct MyStruct
{
    int i;
    double d;
};

// Not initialized
MyStruct *s1 = new MyStruct;
cout << "s1->i: " << s1->i << endl; // not initialized
cout << "s1->d: " << s1->d << endl; // not initialized

// Initialized to 0
MyStruct *s2 = new MyStruct();      // initialized to 0
cout << "s2->i: " << s2->i << endl; // initialized to 0
cout << "s2->d: " << s2->d << endl; // initialized to 0

delete s1;
delete s2;
```

Let's talk about the "**placement new**," which allows us to allocate memory without constructing objects.

The "**placement new**" syntax lets us construct objects at an arbitrary memory location.

It's a special scenario that can be useful when working with memory mapped hardware or custom memory pools.

In this example, we use the global operator new to allocate memory for a single MyClass object without constructing it.



Then we use the "placement new" syntax to construct the object in the pre-allocated memory.

After using the object, we explicitly call the destructor to clean up the object without deallocating
the memory.

4.6 C++ Memory Management: Error Handling

Now what happens if new fails to allocate memory?

Instead of returning a null pointer or an error code, new throws an exception of type std::bad_alloc to signal the error.

This exception can then be caught using a try-catch block, allowing you to gracefully handle the memory allocation failure and take appropriate actions.

cpp-ex-bad-alloc.cpp

```cpp
#include <iostream>
using namespace std;

int main()
{
    try
    {
        int *ptr = new int[100000000000000]; // Attempt to allocate a huge array
        // Use ptr...
        // ...

        // Release memory
        delete[] ptr;
    }
    catch (const std::bad_alloc &e) // bad_alloc exception thrown by new
    {
        cerr << "Memory allocation failed: " << e.what() << endl;
    }

    return 0;
}
```

Let's take a closer look at how exception handling works with the new operator.

Here, we use the new operator to attempt to allocate a massive array of integers. However, due to the limited available memory, the allocation will probably fail, and new will throw
a std::bad_alloc exception.

To handle this exception, we use a try-catch block.

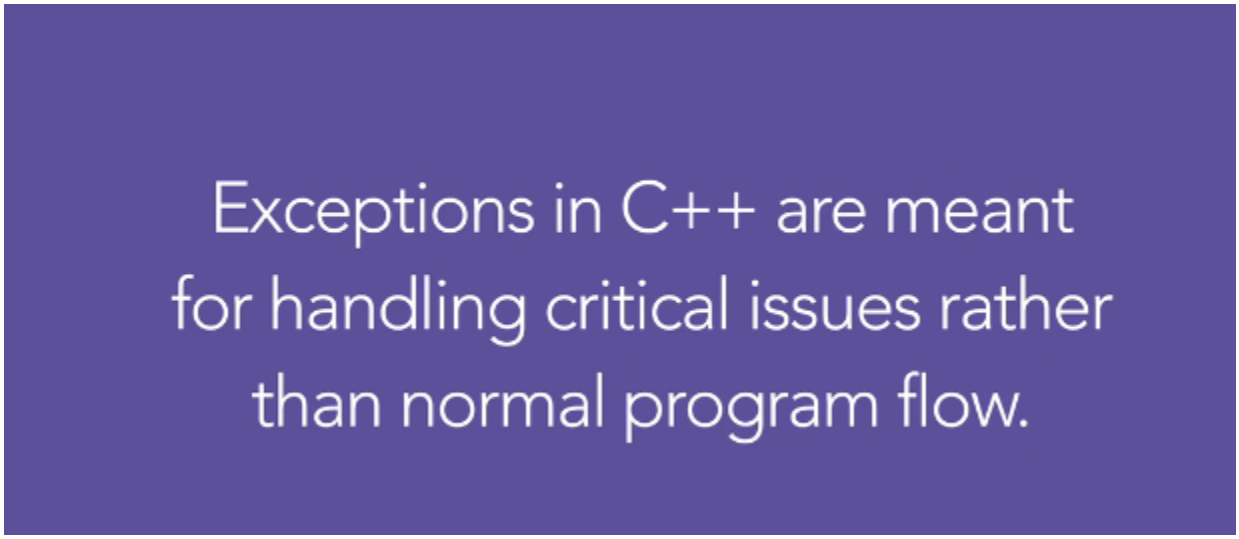
Inside the catch block, we can access the exception object, which provides information about the error, such as the reason for the allocation failure.

By using exceptions, you can separate the error handling code from the normal flow of your program, making it more readable and maintainable.

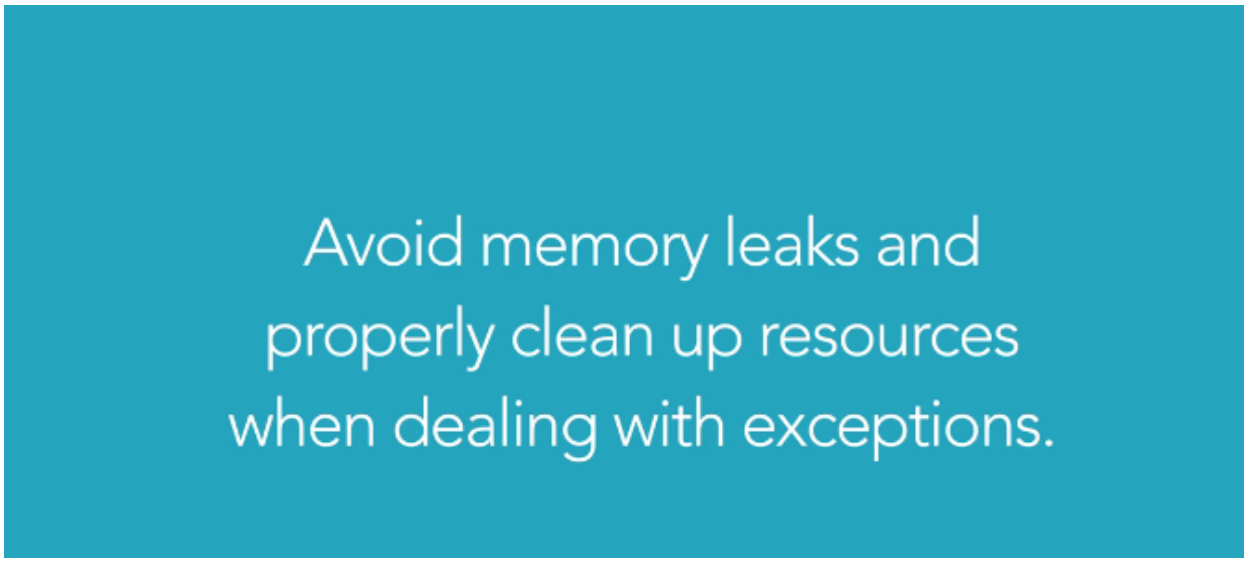
Exception handling also allows you to handle errors at a higher level, giving you more control over how your program responds to exceptional conditions.

It's important to note that exceptions in C++ should be used for exceptional situations, such as
memory allocation failures or other runtime errors.

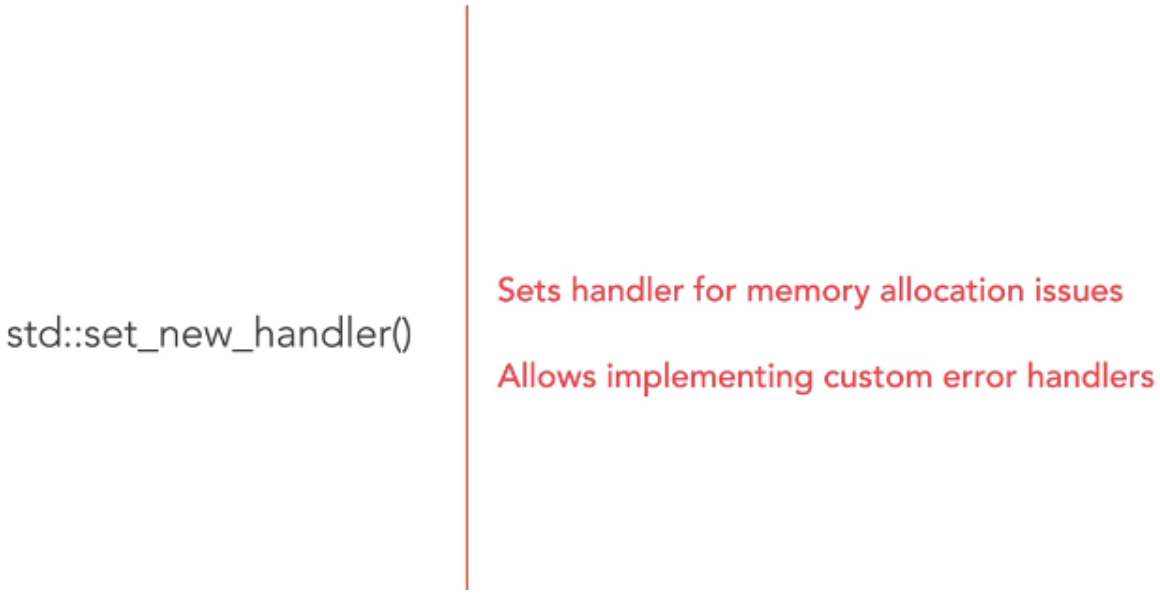
They should not be used for normal program flow.



Additionally, when using dynamic memory allocation, it's crucial to handle exceptions properly, to
avoid memory leaks and ensure the correct cleanup of allocated resources.



Lastly, let's discuss the **set_new_handler() function**.



This function allows you to set a custom handler to be called when a memory allocation fails.

You can use this to implement custom error handling or take specific actions when memory allocation fails.

cpp-set-new-handler.cpp

```cpp
#include <iostream>
#include <new>
using namespace std;

void customMemoryHandler()
{
    cerr << "Memory allocation failed. Custom memory handler called." << endl;
    abort();
}

int main()
{
    // Set custom memory handler
    set_new_handler(customMemoryHandler);

    int *ptr = new int[100000000000000]; // Attempt to allocate a huge array
    // Use ptr...
    // ...

    // Release memory
    delete[] ptr;

    return 0;
}
```

In this example, we set a custom memory handler using set_new_handler() that prints an error message and calls abort() when a memory allocation fails.

The new operator will call this custom handler when it encounters a memory allocation failure.

In conclusion, C++ memory management offers more advanced features compared to C.

The new and delete operators handle dynamic memory allocation, object construction and destruction.

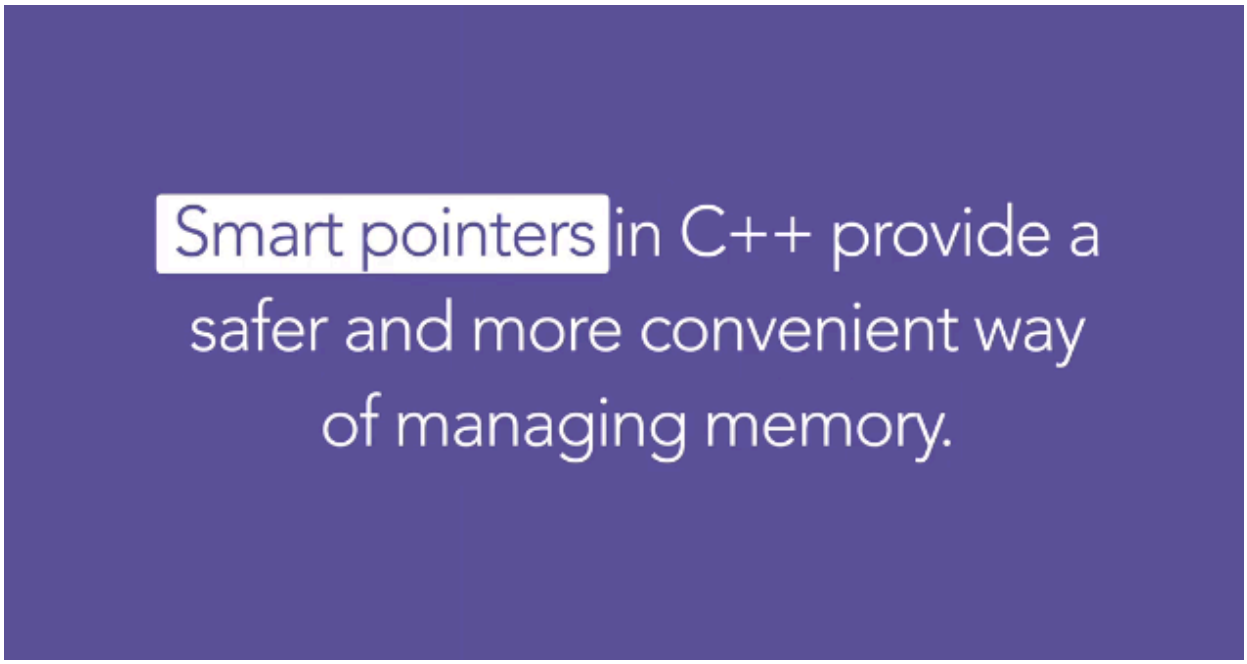
Understanding these concepts and using them effectively is crucial to writing robust, efficient and error free C++ programs.

Always remember to manage dynamically allocated memory properly, avoid memory leaks, and handle exceptions gracefully to ensure your C++ code performs well and remains reliable and secure.

4.7 Smart Pointers: Safer Memory Management in C++

Manual memory management in C++ can lead to various memory-related issues, such as memory leaks, dangling pointers, and incorrect use of delete.

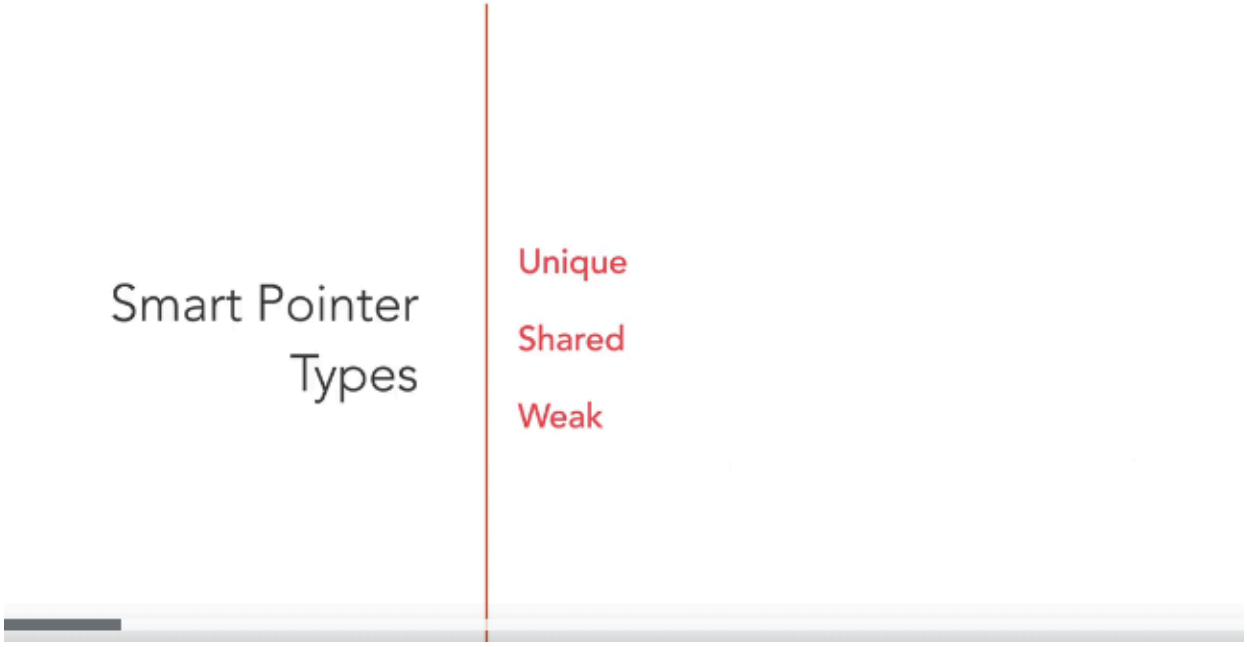
To address these problems and make memory management safer and more convenient,
C++ provides smart pointers.



Smart pointers are objects that behave like pointers but manage the memory they point to
automatically. They deallocate memory when it's no longer needed, making memory management less error-prone and more efficient.

C++ offers three types of smart pointers:

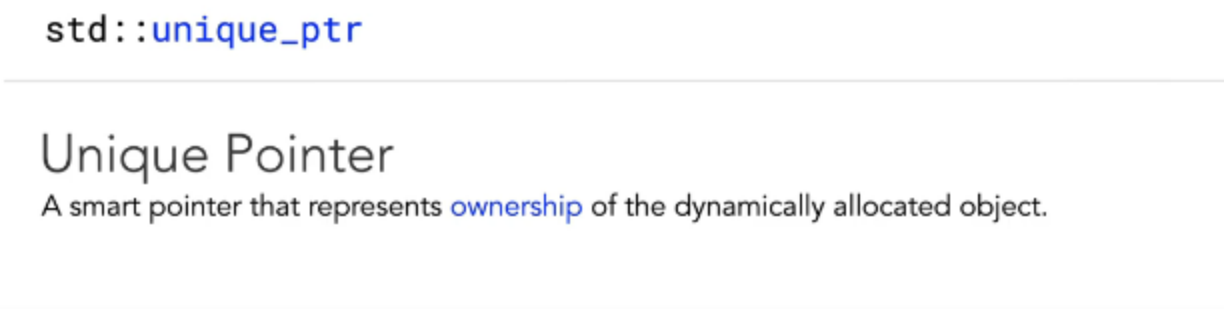
the unique pointer, the shared pointer, and the weak pointer.



Each has its unique characteristics and use cases.

Let's explore them with code examples.

Unique pointer is a smart pointer that represents ownership of the dynamically allocated object.



It ensures that only one unique pointer can own the object at a time. When the unique pointer is destroyed or goes out of scope, it automatically deallocates the memory it owns.

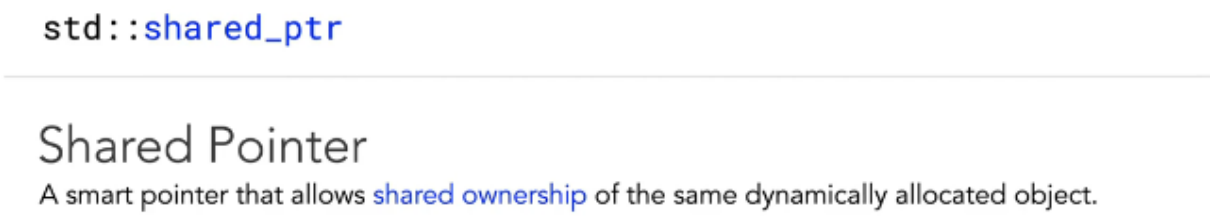
unique_ptr.cpp

```cpp
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    // Create a unique_ptr and allocate memory for an integer
    unique_ptr<int> ptr = make_unique<int>(42);

    // Access the value through the unique_ptr
    cout << "Value: " << *ptr << endl;

    // No need to call delete, memory is automatically deallocated when ptr goes out of scope
    return 0;
}
```

This eliminates the need for explicit calls to delete.

Shared pointers.

The shared pointer allows multiple shared pointer objects to share ownership of the same dynamically allocated object.



The memory is automatically deallocated when the last shared pointer that owns the object is destroyed, as demonstrated in the following code example.

So first, we create a shared pointer and allocate memory for an array of integers.

shared_ptr.cpp

```cpp
#include <memory>
using namespace std;

int main()
{
    // Create a shared_ptr and allocate memory for an array of integers
    shared_ptr<int[]> ptr = make_shared<int[]>(5);

    // Access the elements of the array through the shared_ptr
    for (int i = 0; i < 5; ++i)
    {
        ptr[i] = i * 10;
        cout << ptr[i] << " ";
    }
    cout << endl;

    // No need to call delete, memory is automatically deallocated
    // when the last shared_ptr goes out of scope

    return 0;
}
```

Then we access the elements of the array through the shared pointer.

And finally, as you can see, there is no need to call delete.

Memory will automatically be deallocated when the last shared pointer goes out of scope.

Weak pointers.

The weak pointer is a smart pointer that can observe but does not own the object.

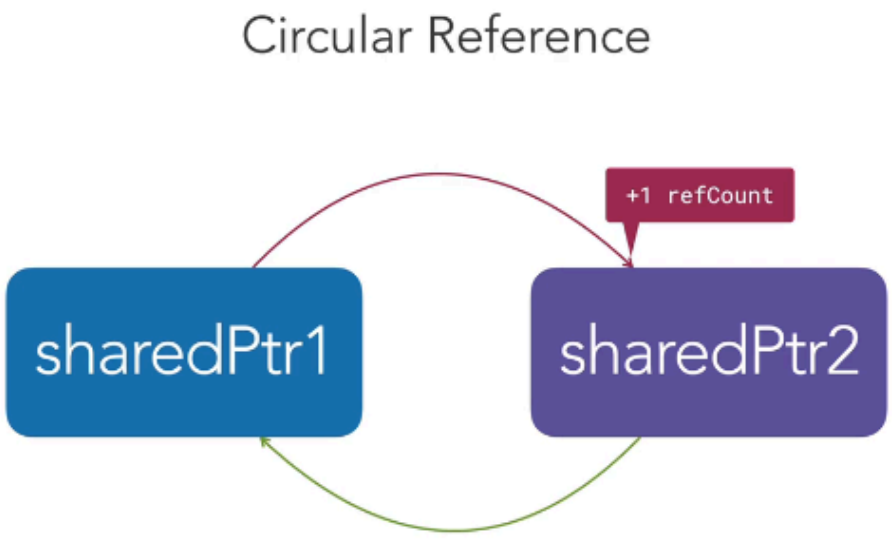
`std::weak_ptr`

Weak Pointer

A smart pointer that can observe but does not own the dynamically allocated object.

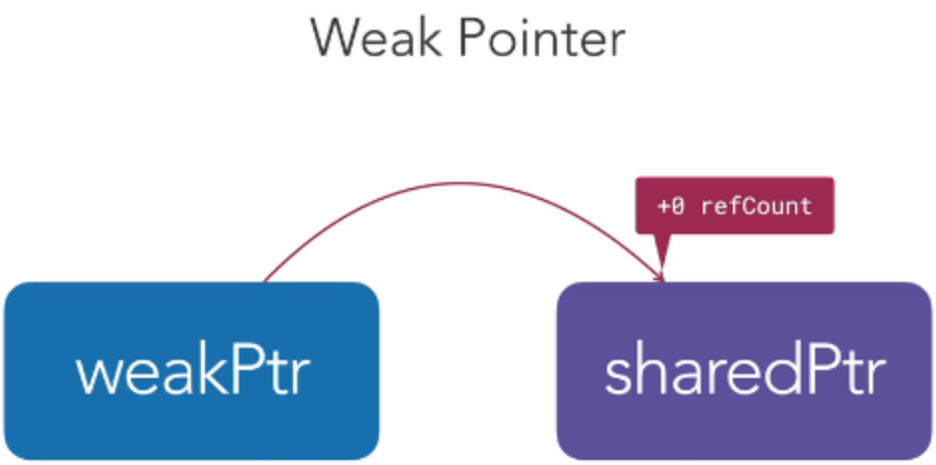
It is created from a shared pointer and provides a way to access the object without affecting its ownership count.

This helps prevent circular references that can lead to memory leaks when a smart pointer has a reference to another smart pointer, and that other pointer also has a reference back to the first one, that's when we get a circular reference.



This creates a loop that prevents these objects from being deleted because their reference count will never go to zero, leading to a memory leak.

When you create weak pointers from a shared pointer, the reference count of the shared pointer won't increase.

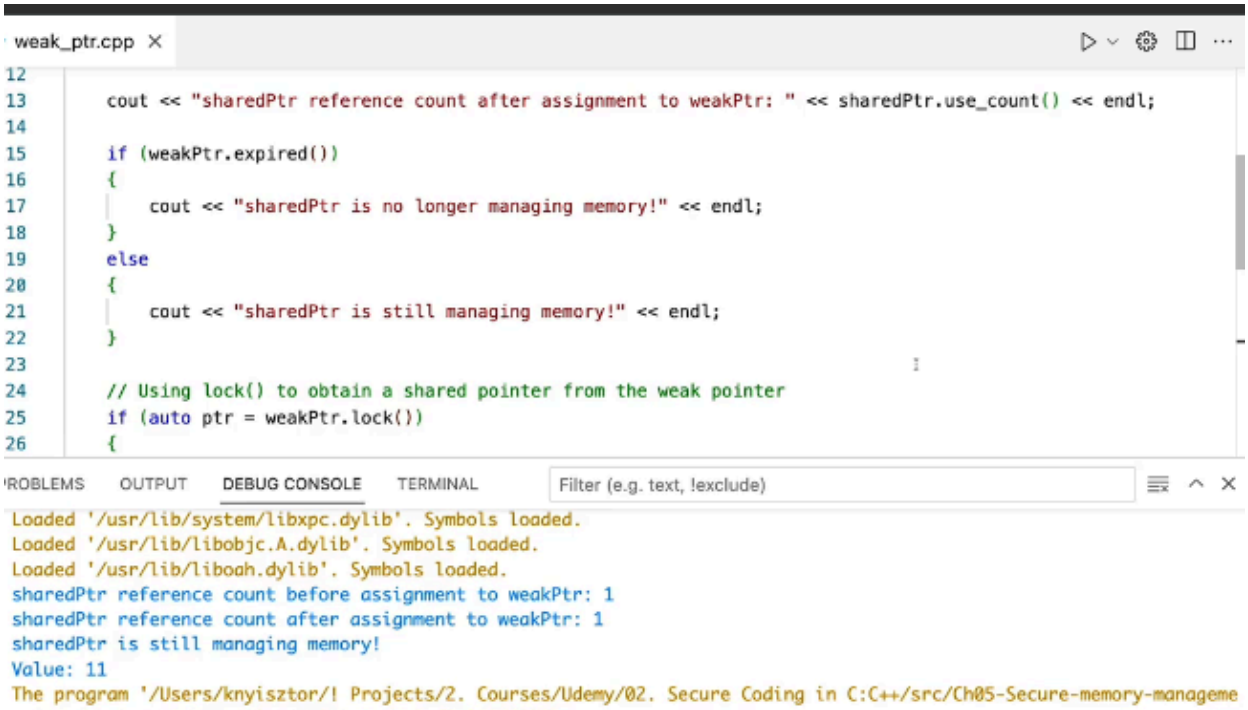


This way we can avoid the formation of circular references.

The following example demonstrates how to use weak pointers in C++.

weak_ptr.cpp

```cpp
#include <iostream>
#include <memory>
using namespace std;

int main()
{
    shared_ptr<int> sharedPtr = make_shared<int>(11);

    cout << "sharedPtr reference count before assignment to weakPtr: " << sharedPtr.use_count() << endl;
    // Create a weak pointer from sharedPtr
    weak_ptr<int> weakPtr = sharedPtr;

    cout << "sharedPtr reference count after assignment to weakPtr: " << sharedPtr.use_count() << endl;

    if (weakPtr.expired())
    {
        cout << "sharedPtr is no longer managing memory!" << endl;
    }
    else
    {
        cout << "sharedPtr is still managing memory!" << endl;
    }

    // Using lock() to obtain a shared pointer from the weak pointer
    if (auto ptr = weakPtr.lock())
    {
        cout << "Value: " << *ptr << endl;
```



First, we declare a shared pointer that stores the value 11.

Next, we create a weak pointer that references the initial shared pointer.

We print the reference count of the shared pointer to the console before and after the assignment.

To determine if the shared pointer still exists, we can utilise the expired() method of the weak pointer.

Now, while we can't use the weak pointer directly, we can call its lock() method to obtain a valid shared pointer if the original object has not been deleted.

Otherwise, lock() returns an empty shared pointer. By using weak references, we can safely and securely handle shared pointers without affecting their lifespan.

Let's run this code.

As you can see, the reference count of sharedPtr is the same before and after assigning it to
weakPtr.

Now let's talk about the benefits of smart pointers.

First, as we've seen, automatic deallocation. Smart pointers automatically manage the memory they point to, reducing the risk of memory leaks and double deletions.

Clear ownership semantics. Unique pointer and shared pointer clearly define ownership of the objects they point to, making the code more readable and less error prone.

Self-documenting code.

Smart pointers make the code self-documenting by explicitly expressing ownership relationships. And finally, exception safety.

Smart pointers ensure that memory is properly deallocated even if exceptions occur.

This enhances the robustness of your code at no cost.

In summary, smart pointers provide a safer and more efficient way to manage memory in C++.

By using smart pointers, you can avoid many common memory management mistakes and focus on writing reliable and maintainable code.

Choose the appropriate smart pointer based on your ownership needs and let C++ handle memory management for you.

Use smart pointers whenever possible.