

# Pointers in C++

Dr. Anwar Shah, PhD, MBA(HR)

Assistant Professor in CS

FAST National University of Computer and Emerging Sciences CFD

# Overview

---

- What is a pointer?
- Declaring pointers
- Storing addresses in pointers
- Dereferencing pointers
- Dynamic memory allocation
- Pointer arithmetic
- Pointers and arrays
- Pass-by-reference with pointers
- const and pointers
- Using pointers to functions
- Potential pointer pitfalls
- What is a reference?
- Review passing references to functions
- const and references
- Reference variables in range-based for loops
- Potential reference pitfalls
- Raw vs. Smart pointers

What is a Pointer?

# What is a Pointer?

---

- A variable
  - whose value is an address
- What can be at that address?
  - Another variable
  - A function
- Pointers point to variables or functions?
- If x is an integer variable and its value is 10  
then I can declare a pointer that points to it
- To use the data that the pointer is pointing to you must know its type

# Why use Pointers?

---

Can't I just use the variable or function itself?

**Yes, but not always**

- Inside functions, pointers can be used to access data that are defined outside the function. Those variables may not be in scope so you can't access them by their name
- Pointers can be used to operate on arrays very efficiently
- We can allocate memory dynamically on the heap or free store.
  - This memory doesn't even have a variable name.
  - The only way to get to it is via a pointer
- With OO. pointers are how polymorphism works!
- Can access specific addresses in memory
  - useful in embedded and systems applications

Declaring Pointer?

## Declaring Pointers

---

```
variable_type *pointer_name;
```

```
int *int_ptr;  
double* double_ptr;  
char *char_ptr;  
string *string_ptr;
```

# Declaring Pointers

---

Initializing pointer variables to 'point nowhere'

```
variable_type *pointer_name {nullptr};
```

```
int *int_ptr {};  
double* double_ptr {nullptr};  
char *char_ptr {nullptr};  
string *string_ptr {nullptr};
```



# Declaring Pointers

---

Initializing pointer variables to 'point nowhere'

- Always initialize pointers
- Uninitialized pointers contain garbage data and can 'point anywhere'
- Initializing to zero or `nullptr` (C++ 11) represents address zero
  - implies that the pointer is 'pointing nowhere'
- If you don't initialize a pointer to point to a variable or function then you should initialize it to `nullptr` to 'make it null'

# Accessing the Pointer Address and Storing Address in a Pointer

## Accessing Pointer Address?

---

**&** the address operator

- Variables are stored in unique addresses
- Unary operator
- Evaluates to the address of its operand
  - Operand cannot be a constant or expression that evaluates to temp values

```
int num{10};
```

```
cout << "Value of num is: " << num << endl;  // 10
```

```
cout << "sizeof of num is: " << sizeof num << endl;  // 4
```

```
cout << "Address of num is: " << &num << endl;    // 0x61ff1c
```

## Accessing Pointer Address?

---

**&** the address operator - example

```
int *p;

cout << "Value of p is: " << p << endl; // 0x61ff60 - garbage

cout << "Address of p is: " << &p << endl; // 0x61ff18

cout << "sizeof of p is: " << sizeof p << endl; // 4

p = nullptr; // set p to point nowhere

cout << "Value of p is: " << p << endl; // 0
```

## Accessing Pointer Addresses?

---

`sizeof` a pointer variable

- Don't confuse the size of a pointer and the size of what it points to
- All pointers in a program have the same size
- They may be pointing to very large or very small types

```
int *p1 {nullptr};  
double *p2 {nullptr};  
unsigned long long *p3 {nullptr};  
vector<string> *p4 {nullptr};  
string *p5 {nullptr};
```

## Storing an Address in Pointer Variable?

---

Typed pointers

- The compiler will make sure that the address stored in a pointer variable is of the correct type

```
int score {10};  
double high_temp {100.7};
```

```
int *score_ptr {nullptr};
```

```
score_ptr = &score;    // OK
```

```
score_ptr = &high_temp; // Compiler Error
```

# Storing an Address in Pointer Variable?

---

& the address operator

- Pointers are variables so they can change
- Pointers can be null
- Pointers can be uninitialized

```
double high_temp {100.7};  
double low_temp {37.2};
```

```
double *temp_ptr;
```

```
temp_ptr = &high_temp; // points to high_temp  
temp_ptr = &low_temp;  // now points to low_temp
```

```
temp_ptr = nullptr;
```

---

# Dereferencing a Pointer



# Dereferencing a Pointer

---

Access the data we're pointing to – dereferencing a pointer

- If `score_ptr` is a pointer and has a valid address
- Then you can access the data at the address contained in the `score_ptr` using the dereferencing operator `*`

```
int score {100};  
int *score_ptr {&score};  
  
cout << *score_ptr << endl;    // 100  
  
*score_ptr = 200;  
cout << *score_ptr << endl;    // 200  
cout << score << endl;         // 200
```

---

## Dereferencing a Pointer

---

Access the data we're pointing to

```
double high_temp {100.7};  
double low_temp {37.4};  
double *temp_ptr {&high_temp};  
  
cout << *temp_ptr << endl;    // 100.7  
  
temp_ptr = &low_temp;  
  
cout << *temp_ptr << endl;    // 37.4
```

---

## Dereferencing a Pointer

---

Access the data we're pointing to

```
string name {"Frank"};
```

```
string *string_ptr {&name};
```

```
cout << *string_ptr << endl;    // Frank
```

```
name = "James";
```

```
cout << *string_ptr << endl;    // James
```

---

# Dynamic Memory Allocation

# Dynamic Memory Allocation

---

## **Allocating storage from the heap at runtime**

- We often don't know how much storage we need until we need it
  - We can allocate storage for a variable at run time
  - Recall C++ arrays
    - We had to explicitly provide the size and it was fixed
    - But vectors grow and shrink dynamically
  - We can use pointers to access newly allocated heap storage
-

# Dynamic Memory Allocation

---

using **new** to allocate storage

```
int *int_ptr {nullptr};

int_ptr = new int;           // allocate an integer on the heap

cout << int_ptr << endl;     // 0x2747f28

cout << *int_ptr << endl;    // 41188048 - garbage

*int_ptr = 100;

cout << *int_ptr << endl;    // 100
```

---

# Dynamic Memory Allocation

---

using `delete` to deallocate storage

```
int *int_ptr {nullptr};  
  
int_ptr = new int;    // allocate an integer on the heap  
  
. . .  
  
delete int_ptr;    // frees the allocated storage
```

---

# Dynamic Memory Allocation

---

**using `new[]` to allocate storage for an array**

```
int *array_ptr {nullptr};  
int size {};  
  
cout << "How big do you want the array? ";  
cin >> size;  
  
array_ptr = new int[size]; // allocate array on the heap  
  
// We can access the array here  
// we'll see how in a few slides
```

---



# Dynamic Memory Allocation

---

**using delete[] to deallocate storage for an array**

```
int *array_ptr {nullptr};
int size {};

cout << "How big do you want the array?";
cin >> size;

array_ptr = new int[size]; // allocate array on the heap

. . .

delete [] array_ptr; // free allocated storage
```

---

# Dynamic Memory Allocation

---

Memory leakage occurrence in dynamic memory allocation using pointers in C++

- **Failure to Free Memory:** When memory is allocated on the heap using operators like new or malloc, it must be explicitly released using delete or free, respectively, when it's no longer needed. Failure to do so leads to memory leakage.

```
int* ptr = new int; // Allocate memory on the heap
// ... do some work with ptr
// Memory should be released when done
// delete ptr; // If not deleted, it results in memory leakage
```

# Dynamic Memory Allocation

---

- **Lost Pointers:** If you lose all references to a dynamically allocated memory block (e.g., by overwriting the pointer or by going out of scope without deleting it), you won't be able to release that memory properly.

```
int* ptr = new int; // Allocate memory on the heap  
ptr = nullptr;     // The original pointer is lost, and the memory can't be freed  
// Memory leakage occurs because the original pointer is gone
```

# Dynamic Memory Allocation

---

- **Missing Deallocation in Exception Handling:** If an exception occurs and you haven't wrapped your memory management code in exception-safe constructs (like smart pointers or RAII), the memory may not be properly released.

```
int* ptr = new int; // Allocate memory on the heap
```

```
// Exception occurs here
```

```
// delete ptr; // If not included in exception handling, memory may leak
```



# Dynamic Memory Allocation

---

- Always pair memory allocation (new, malloc, etc.) with proper deallocation (delete, free, etc.) when you're done with the memory.
  - Use smart pointers (e.g., `std::shared_ptr` or `std::weak_ptr`) when appropriate, as they automatically manage memory.
  - Follow RAII (Resource Acquisition Is Initialization) principles to ensure that resources, including dynamically allocated memory, are automatically released when they go out of scope.
  - Be cautious about losing references to dynamically allocated memory.
  - Use tools and practices for memory leak detection and debugging when working with complex codebases or applications.
-

# Relationship between Array and Pointer

# Relationship Between Arrays and Pointers

---

- The value of an array name is the address of the first element in the array
- The value of a pointer variable is an address
- If the pointer points to the same data type as the array element then the pointer and array name can be used interchangeably (almost)



## Relationship Between Arrays and Pointers

---

```
int scores[] {100, 95, 89};

cout << scores << endl;    // 0x61fec8
cout << *scores << endl;   // 100

int *score_ptr {scores};

cout << score_ptr << endl; // 0x61fec8

cout << *score_ptr << endl; // 100
```

---



## Relationship Between Arrays and Pointers

---

```
int scores[] {100, 95, 89};  
  
int *score_ptr {scores};  
  
cout << score_ptr[0] << endl; // 100  
  
cout << score_ptr[1] << endl; // 95  
  
cout << score_ptr[2] << endl; // 89
```

---

## Using pointers in expressions

---

```
int scores[] {100, 95, 89};
```

```
int *score_ptr {scores};
```

```
cout << score_ptr << endl;    // 0x61ff10
```

```
cout << (score_ptr + 1) << endl; // 0x61ff14
```

```
cout << (score_ptr + 2) << endl; // 0x61ff18
```

---

## Using pointers in expressions

---

```
int scores[] {100, 95, 89};
```

```
int *score_ptr {scores};
```

```
cout << *score_ptr << endl; // 100
```

```
cout << *(score_ptr + 1) << endl; // 95
```

```
cout << *(score_ptr + 2) << endl; // 89
```

---

## Subscript and Offset notation equivalence

---

```
int array_name[] {1,2,3,4,5};  
  
int *pointer_name {array_name};
```

Subscript Notation	Offset Notation
<code>array_name[index]</code>	<code>*(array_name + index)</code>
<code>pointer_name[index]</code>	<code>*(pointer_name + index)</code>

---

# Pointer Arithmetic

# Pointer Arithmetic

---

- Pointers can be used in
  - Assignment expressions
  - Arithmetic expressions
  - Comparison expressions
- C++ allows pointer arithmetic
- Pointer arithmetic only makes sense with raw arrays



## ++ and --

---

- (++) increments a pointer to point to the next array element

```
int_ptr++;
```

- (--) decrements a pointer to point to the previous array element

```
int_ptr--;
```

---

## + and -

---

- (+) increment pointer by  $n * \text{sizeof}(\text{type})$

`int_ptr += n; or int_ptr = int_ptr + n;`

- (-) decrement pointer by  $n * \text{sizeof}(\text{type})$

`int_ptr -= n; or int_ptr = int_ptr - n;`

---



## Subtracting two pointers

---

- Determine the number of elements between the pointers
- Both pointers must point to the same data type

```
int n = int_ptr2 - int_ptr1;
```

## Comparing two pointers == and !=

---

Determine if two pointers point to the same location

- does NOT compare the data where they point!

```
string s1 {"Frank"};  
string s2 {"Frank"};
```

```
string *p1 {&s1};  
string *p2 {&s2};  
string *p3 {&s1};
```

```
cout << (p1 == p2) << endl;    // false  
cout << (p1 == p3) << endl;    // true
```

---

## Comparing the data pointers point to

---

Determine if two pointers point to the same data

- you must compare the referenced pointers

```
string s1 {"Frank"};  
string s2 {"Frank"};
```

```
string *p1 {&s1};  
string *p2 {&s2};  
string *p3 {&s1};
```

```
cout << (*p1 == *p2) << endl;    // true  
cout << (*p1 == *p3) << endl;    // true
```

---

# Constant and Pointer

# Passing pointers to a function

---

## `const` and Pointers

- There are several ways to qualify pointers using `const`
    - Pointers to constants
    - Constant pointers
    - Constant pointers to constants
-

## Pointers to constants

---

- The data pointed to by the pointers is constant and **cannot** be changed.
- The pointer itself can change and point somewhere else.

```
int high_score {100};  
int low_score { 65};  
const int *score_ptr { &high_score };
```

```
*score_ptr = 86;    // ERROR  
score_ptr = &low_score; // OK
```

---

## Constant pointers

---

- The data pointed to by the pointers can be changed.
- The pointer itself **cannot** change and point somewhere else

```
int high_score {100};  
int low_score { 65};  
int *const score_ptr { &high_score };
```

```
*score_ptr = 86;    // OK  
score_ptr = &low_score; // ERROR
```

## Constant pointers to constants

---

- The data pointed to by the pointer is constant and **cannot** be changed.
- The pointer itself **cannot** change and point somewhere else.

```
int high_score {100};  
int low_score { 65};  
const int *const score_ptr { &high_score };
```

```
*score_ptr = 86;    // ERROR  
score_ptr = &low_score; // ERROR
```



# Passing Pointer to Functions

# Passing pointers to a function

---

- Pass-by-reference with pointer parameters
  - We can use pointers and the dereference operator to achieve pass-by-reference
  - The function parameter is a pointer
  - The actual parameter can be a pointer or address of a variable
-

# Passing pointers to a function

---

Pass-by-reference with pointers – defining the function

```
void double_data(int *int_ptr);  
  
void double_data(int *int_ptr) {  
    *int_ptr *= 2;  
  
    // *int_ptr = *int_ptr * 2;  
}
```

---

# Passing pointers to a function

---

Pass-by-reference with pointers – calling the function

```
int main() {  
    int value {10};  
  
    cout << value << endl;    // 10  
  
    double_data( &value);  
  
    cout << value << endl;    // 20  
}
```

---

# Returning a Pointer from a Function

# Returning a Pointer from a Function

---

- Functions can also return pointers

```
type *function();
```

- Should return pointers to
    - Memory dynamically allocated in the function
    - To data that was passed in
  - Never return a pointer to a local function variable!
-

## returning a parameter

---

```
int *largest_int(int *int_ptr1, int *int_ptr2) {  
    if (*int_ptr1 > *int_ptr2)  
        return int_ptr1;  
    else  
        return int_ptr2;  
}
```

## returning a parameter

---

```
int main() {  
    int a{100};  
    int b{200};  
  
    int *largest_ptr {nullptr};  
    largest_ptr = largest_int(&a, &b);  
    cout << *largest_ptr << endl;    // 200  
    return 0;  
}
```



## returning dynamically allocated memory

```
int *create_array(size_t size, int init_value = 0) {  
  
    int *new_storage {nullptr};  
  
    new_storage = new int[size];  
    for (size_t i{0}; i < size; ++i)  
        *(new_storage + i) = init_value;  
    return new_storage;  
}
```

## returning dynamically allocated memory

```
int main() {  
    int *my_array;    // will be allocated by the function  
  
    my_array = create_array(100,20);    // create the array  
  
    // use it  
  
    delete [] my_array; /    / be sure to free the storage  
    return 0;  
}
```

## Never return a pointer to a local variable!!

```
int *dont_do_this () {  
    int size {};  
    . . .  
    return &size;  
}  
int *or_this () {  
    int size {};  
    int *int_ptr {&size};  
    . . .  
    return int_ptr;  
}
```

# Pointer Potential Pitfalls

# Pitfalls

In programming, pointers are variables that hold the memory address of another variable. However, sometimes pointers can point to invalid memory locations or uninitialized memory addresses, which can lead to unexpected program behavior. Some commonly used terms related to pointers are given in the coming slides:

# Potential Pointer Pitfalls

---

- Uninitialized pointers
- Dangling Pointers
- Not checking if new failed to allocate memory
- Leaking memory

# Uninitialized Pointers/ Wild Pointers

A wild pointer is a pointer that does not point to any valid memory location. It may contain a random or uninitialized memory address or it may point to a memory location that has already been released or deallocated. Dereferencing a wild pointer can lead to a segmentation fault or other types of undefined behavior.

Example:

```
int *p; // uninitialized pointer
```

```
*p = 5; // dereferencing a wild pointer
```

## Uninitialized pointers

---

```
int *int_ptr; // pointing anywhere
```

```
. . .
```

```
*int_ptr = 100; // Hopefully a crash
```



# Void or Generic Pointer

Void pointer: A void pointer is a pointer that has no specific type. It is used to hold the address of any type of object but cannot be directly dereferenced. It needs to be cast to a specific type of pointer before it can be dereferenced.

## **Example:**

```
void *p;  
int num = 10;  
p = &num; // void pointer holds the address of an integer variable  
int *q = static_cast<int*>(p); // casting the void pointer to an integer pointer  
cout << *q << endl; // dereferencing the integer pointer
```

# Void or Generic Pointer (More Explanation)

In C++, a void pointer, also known as a generic pointer, is a special type of pointer that can point to objects of any type. It is declared using the keyword "void" as the pointer's type. The void pointer is a way to achieve a higher level of flexibility and genericity in C++ programming.

The reason it is called a "void" pointer is because it doesn't have any associated data type information. It points to a memory address without specifying what type of data is stored at that address. This lack of type information makes it less safe to use, as you need to be careful when dereferencing or casting it.

# Void or Generic Pointer (More Explanation)

```
#include <iostream>
```

```
int main() {
```

```
    int num = 10;
```

```
    float value = 3.14;
```

```
    void* ptr; // Declaring a void pointer
```

```
    ptr = &num; // Assigning address of 'num' to the void pointer
```

```
    std::cout << "Value of 'num': " << *(static_cast<int*>(ptr)) << std::endl;
```

```
    ptr = &value; // Assigning address of 'value' to the void pointer
```

```
    std::cout << "Value of 'value': " << *(static_cast<float*>(ptr)) << std::endl;
```

```
    return 0;
```

```
}
```

# Void or Generic Pointer (More Explanation)

In this example, we have a void pointer called 'ptr' that can store the addresses of different types of variables. We assign the address of the 'num' variable to the void pointer and then use a static cast to interpret it as an integer pointer when dereferencing it. Similarly, we assign the address of the 'value' variable and interpret it as a float pointer to retrieve its value.

Note that when using a void pointer, you must cast it to the appropriate type before dereferencing it. This casting is necessary because the void pointer does not know the size or type of the data it points to. Therefore, it's important to be cautious when working with void pointers to avoid type-related errors and ensure type safety.

# Void or Generic Pointer (More Explanation)

**we have to cast it before dereferencing it, what does it mean?**

Casting a void pointer before dereferencing it means explicitly telling the compiler what type of data is stored at the memory location pointed to by the void pointer.

In C++, the void pointer doesn't have any type information associated with it, so the compiler doesn't know how to interpret the data at that memory location. When you want to access or manipulate the data pointed to by the void pointer, you need to cast it to a specific type to inform the compiler about the correct data type.

# Void or Generic Pointer (More Explanation)

```
void* ptr; // Declaring a void pointer
```

```
ptr = &num; // Assigning address of 'num' to the void pointer
```

```
std::cout << "Value of 'num': " << *(static_cast<int*>(ptr)) << std::endl;
```

```
ptr = &value; // Assigning address of 'value' to the void pointer
```

```
std::cout << "Value of 'value': " << *(static_cast<float*>(ptr)) << std::endl;
```

# Void or Generic Pointer (More Explanation)

Before dereferencing the void pointer using the `*` operator, we use the `static_cast` to explicitly cast the void pointer to the appropriate type. In the first case, we cast it to an int pointer (`static_cast<int*>(ptr)`), and in the second case, we cast it to a float pointer (`static_cast<float*>(ptr)`). This casting allows the compiler to know the correct size and interpretation of the data, enabling us to access the value stored at that memory location correctly.

Casting void pointers is necessary because the compiler needs to understand the size and type of data it's dealing with. Without the cast, the compiler wouldn't know how many bytes to read or how to interpret the binary data at the memory location, leading to undefined behavior and potential errors in your program.

# Null Pointer

A null pointer is a pointer that does not point to any valid memory location. It is typically used to indicate that the pointer is not currently pointing to any object. A null pointer value is represented by the constant `NULL` in C or `nullptr` in C++.

## **Example:**

```
int *p = nullptr; // initializing a null pointer
if (p == nullptr) {
    cout << "p is a null pointer" << endl;
}
```



## Dangling pointer

---

- Pointer that is pointing to released memory
  - For example, 2 pointers point to the same data
  - 1 pointer releases the data with delete
  - The other pointer accesses the release data
- Pointer that points to memory that is invalid
  - We saw this when we returned a pointer to a function local variable

# Dangling Pointer

Dangling pointer: A dangling pointer is a pointer that points to an object that no longer exists. This can happen when the memory allocated to an object is released or deallocated but the pointer is not updated to reflect this change. Dereferencing a dangling pointer can lead to undefined behavior.

Example:

```
int *p = new int(5); // dynamically allocating memory for an integer  
delete p; // deallocating memory  
cout << *p << endl; // p is now a dangling pointer
```

# Dangling Pointer

- Let us consider the program in the figure1.
- Memory is allotted to main on RAM and similarly to function fun();
- Now a int pointer is declared and also another block is declared using { }, in C the variables declared inside the block { } will have a scope inside that block only and if the variable is declared outside will have scope all over the { }.

for example: x has a scope inside the block in which it is declared only.

but \*xp has a scope in the { } of the function fun(); as it is declared inside it therefore its scope will also be entertained inside the block{ }, in which x is declared.

Figure 1

```
void fun(void);
main()
{
    fun();
}
void fun()
{
    int *p;
    {
        int x;
        p=&x;
        .....
    }
}
```

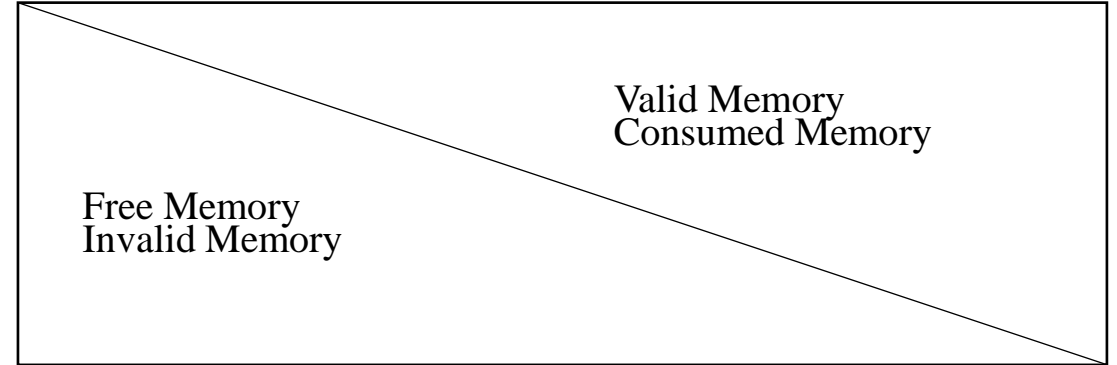
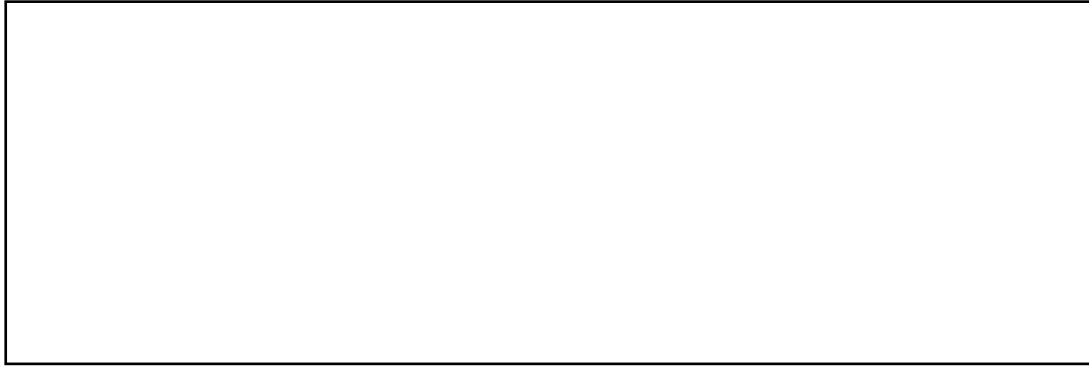
Figure 2

```
void fun(void);
main()
{
    fun();
}
void fun()
{
    int *p;
    {
        int x;
        p=&x;
        .....
        p=NULL;
    }
}
```

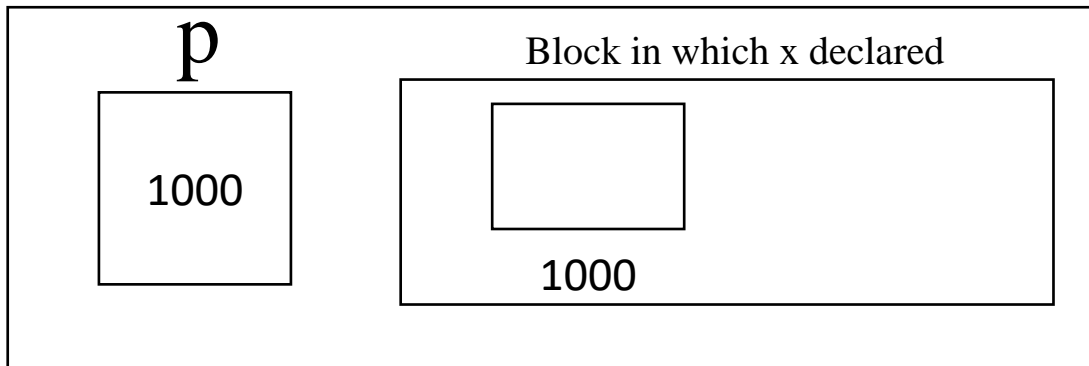
# Dangling Pointer continue...

- Now consider the figure 3.
- The address of `x` is 1000. which is inside pointer `p`.
- When the function `fun()`; executed i.e it came to the block `{}`, where `x` is declared. Memory will be allocated to the `x` and that memory will now be called as consumed memory. `x` will get the base address (say 1000). And as `p` pointing to `x` then the base address will be stored inside pointer `p`. when the block complete the execution it will be destroyed but the base address 1000 will remain inside the pointer. Now it means that the pointer is now pointing to a free memory location i.e invalid memory, same as wild pointer. In this case the pointer which once pointing to a valid memory address but now pointing to a invalid memory address is called dangling pointer
- What is the solution?
- When the inner block is going to finish the execution. It must be noted that pointer `p` should be assigned with a `NULL` Macro in C and `nullptr` in C++ (figure 2).

main()



fun()



fun()

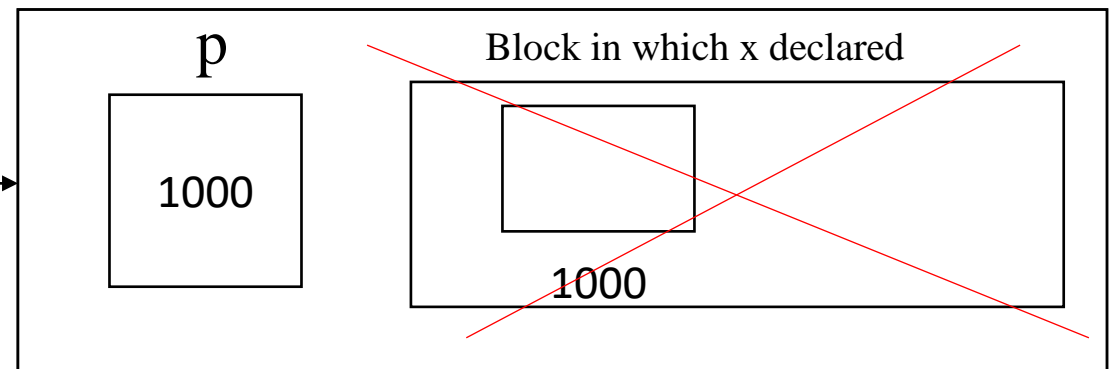


Figure 3

## Not checking if new failed

- If `new` fails an exception is thrown
- We can use exception handling to catch exceptions
- Dereferencing a null pointer will cause your program to crash

# Leaking memory

---

- Forgetting to release allocated memory with delete
- If you lose your pointer to the storage allocated on the heap you have not way to get to that storage again
- The memory is orphaned or leaked
- One of the most common pointer problems

# Pitfalls

In programming, pointers are variables that hold the memory address of another variable. However, sometimes pointers can point to invalid memory locations or uninitialized memory addresses, which can lead to unexpected program behavior. Here are some commonly used terms related to pointers:



# References

# What is a Reference?

---

- An alias for a variable
- Must be initialized to a variable when declared
- Cannot be null
- Once initialized cannot be made to refer to a different variable
- Very useful as function parameters
- Might be helpful to think of a reference as a constant pointer that is automatically dereferenced

# What is a reference?

---

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto str: stooges)
```

```
    str = "Funny";    // changes the copy
```

```
for (auto str:stooges)
```

```
    cout << str << endl;    // Larry, Moe, Curly
```

# What is a reference?

---

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto &str: stooges)
    str = "Funny";    // changes the actual
```

```
for (auto str:stooges)
    cout << str << endl;    // Funny, Funny, Funny
```

# What is a reference?

---

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto const &str: stooges)  
    str = "Funny";    // compiler error
```

# What is a reference?

---

Using references in range-based for loop

```
vector<string> stooges {"Larry", "Moe", "Curly"};
```

```
for (auto const &str:stooges)  
    cout << str << endl;    // Larry, Moe, Curly
```

# What is a reference?

---

Passing references to functions

L – Values and R - Values



# l-values

---

- l-values
  - values that have names and are addressable
  - modifiable if they are not constants

```
int x {100};    // x is an l-value  
x = 1000;  
x = 1000 + 20;
```

```
string name;    // name is an l-value  
name = "Frank";
```

# l-values

---

- l-values
  - values that have names and are addressable
  - modifiable if they are not constants

```
100 = x;           // 100 is NOT an l-value  
(1000 + 20) = x;  // (1000 + 20) is NOT an l-value
```

```
string name;  
name = "Frank";  
"Frank" = name; // "Frank" is NOT an l-value
```

# r-values

---

- r-value (non-addressable and non-assignable)
  - A value that's not an l-value
    - on the right-hand side of an assignment expression
    - a literal
    - a temporary which is intended to be non-modifiable

```
int x {100};           // 100 is an r-value  
int y = x + 200;       // (x+200) is an r-value
```

```
string name;  
name = "Frank";        // "Frank" is an r-value
```

```
int max_num = max(20,30); // max(20,30) is an r-value
```

## r-values

---

- r-values can be assigned to l-values explicitly

```
int x {100};  
int y {0};
```

```
y = 100;    // r-value 100 assigned to l-value y
```

```
x = x + y;  // r-value (x+y) assigned to l-value x
```

## I-value references

---

- The references we've used are l-value references
  - Because we are referencing l-values

```
int x {100};
```

```
int &ref1 = x;    // ref1 is reference to l-value  
ref1 = 1000;
```

```
int &ref2 = 100; // Error 100 is an r-value
```

# l-value references

---

- The same when we pass-by-reference

```
int square(int &n) {  
    return n*n;  
}
```

```
int num {10};
```

```
square(num);    // OK
```

```
square(5);    // Error - can't reference r-value 5
```

# When to use pointers vs. references parameters

- Pass-by-value
  - when the function does **not** modify the actual parameter, and
  - the parameter is small and efficient to copy like simple types (int, char, double, etc.)

# When to use pointers vs. references parameters

- Pass-by-reference using a pointer
  - when the function does modify the actual parameter,  
and
  - the parameter is expensive to copy,  
and
  - Its OK to the pointer is allowed a nullptr value



# When to use pointers vs. references parameters

- Pass-by-reference using a pointer to `const`
  - when the function does **not** modify the actual parameter,  
and
  - the parameter is expensive to copy,  
and
  - Its OK to the pointer is allowed a `nullptr` value

# When to use pointers vs. references parameters

- Pass-by-reference using a const pointer to const
  - when the function does **not** modify the actual parameter,  
and
  - the parameter is expensive to copy,  
and
  - Its OK to the pointer is allowed a nullptr value,  
and
  - You don't want to modify the pointer itself

# When to use pointers vs. references parameters

- Pass-by-reference using a reference
  - when the function **does** modify the actual parameter, and
  - the parameter is expensive to copy, and
  - The parameter will never be nullptr

# When to use pointers vs. references parameters

---

- Pass-by-reference using a `const` reference
  - when the function does **not** modify the actual parameter, and
  - the parameter is expensive to copy, and
  - The parameter will never be `nullptr`

Good Luck!

