ENPM702

Introductory Robot Programming

L3: Normal Pointers

v2.0

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED
GRADUATE ENGINEERING

Table of Contents

Changelog

- **v2.0**:
  - Expanded sections on dynamic memory allocation and references.
- **v1.0**: Original version.

Learning Objectives

By the end of this session, you will be able to:

- Understand memory allocation types: static, automatic, and dynamic.
- Declare, initialize, and use normal (raw) pointers effectively.
- Apply pointer operations including dereferencing and comparisons.
- Implement const-correctness with pointers and understand pointer-to-const vs const-pointer.
- Recognize and avoid common pointer pitfalls: memory leaks, dangling pointers, and undefined behavior.
- Use Valgrind for memory error detection and debugging.
- Understand when to use references as an alternative to pointers.

Valgrind

Valgrind

Valgrind is a programming tool used primarily for memory management, memory error detection, and profiling.

- **Memory Leak Detection** – One of the main uses of Valgrind is to find memory leaks in a program. If a program allocates memory but does not free it before termination, Valgrind will detect and report it.
- **Memory Error Detection** – Valgrind can identify a variety of memory errors such as accessing memory that has not been initialized, accessing memory after it has been freed, and writing past the end of an allocated block of memory.
- **Profiling** – With the **Callgrind** and **Cachegrind** tools that come with Valgrind, you can profile your program to find bottlenecks or understand cache usage.
- **Concurrency Bugs** – The **Helgrind** and **DRD** tools are used to detect race conditions in multithreaded programs.
- **Heap and Stack Analysis** – The **Massif** tool provides a detailed breakdown of heap and stack usage, helping developers optimize memory usage.

Valgrind Setup

- `sudo apt install valgrind`
- Add the function below to CMakeLists.txt

```
# Add Valgrind target for lecture3_cpp
find_program(VALGRIND_EXECUTABLE valgrind)
if(VALGRIND_EXECUTABLE)
    add_custom_target(valgrind_lecture3
        COMMAND ${VALGRIND_EXECUTABLE} --tool=memcheck --leak-check=full
        ↪  --show-leak-kinds=all --track-origins=yes
        ↪  $<TARGET_FILE:lecture3_cpp>
        DEPENDS lecture3_cpp
        COMMENT "Running Valgrind on lecture3_cpp"
    )
endif()
```

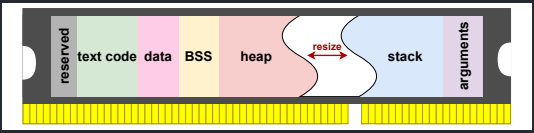
- Valgrind will execute each time you build your project.

Memory Allocation

Memory Allocation

C++ supports three fundamental types of memory allocation.

- Static memory allocation (data and BSS segments).
  - Used for static and global variables.
  - Memory is allocated once and persists throughout the life of your program.
- Automatic memory allocation (stack segment).
  - Used for formal parameters and local variables.
  - The user has no control about allocation/deallocation.
- Dynamic memory allocation (heap segment).
  - Some STL containers store data on the heap (e.g., `std::vector`).
  - The user may explicitly ask for memory to be allocated on the heap. The user is responsible for deallocating this memory.



Stack Segment

- Local variables are allocated on the stack.
- Stack memory allocation size is determined at compile time.
- Memory is automatically deallocated when variables go out of scope.
- Stack operations (allocation/deallocation) are very fast.
- The stack has limited size; exceeding it causes a **stack overflow**.
- You can check your stack size (in kilobytes) with `ulimit -s`

  Automatic memory allocation occurs in the stack segment.

Heap Segment

- Heap memory is allocated at runtime.
- Manual allocation requires explicit deallocation to prevent **memory leaks**.
- RAII (e.g., STL containers, smart pointers) provides automatic heap memory management.
- The heap is flexible in size, suitable for large or variable amounts of data.
- Data persists on the heap beyond function scope. **Dynamic memory allocation** occurs in the heap segment.

Key Difference: The Stack is for small, temporary data; The Heap is for large, persistent data.

Pointers

Pointers

A pointer is a variable …

Pointers

What is a pointer?

A pointer is a variable that holds a memory address as its value…

Pointers

What is a pointer?

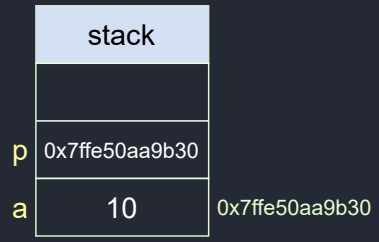
A pointer is a variable that holds a memory address as its value. This memory address belongs to either another **variable**, a **literal**, or a **function**.

Pointers

What is a pointer?

A pointer is a variable that holds a memory address as its value. This memory address belongs to either another **variable**, a **literal**, or a **function**.

- "The variable p holds the address of the variable a" or
- "p is a pointer to a" or
- "p points to a"

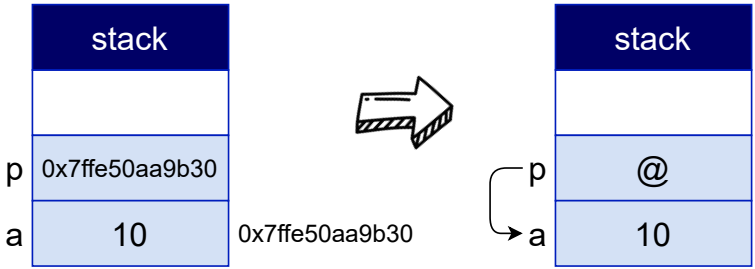


Why Learn Pointers?

- Pointers are fun!! ☺
- Pointers (and reference variables) allow memory savings during function calls.
- Pointers (and reference variables) inside a function enable modification of data declared outside the function.
- Pointers (and reference variables) are used in polymorphism.
- Pointers allow dynamic memory allocations.
- Pointers (smart pointers) are widely used in ROS 2.
- Third-party libraries often use pointers.

In this lecture, we will learn what pointers are. In the following lectures, we will explore when to use them.

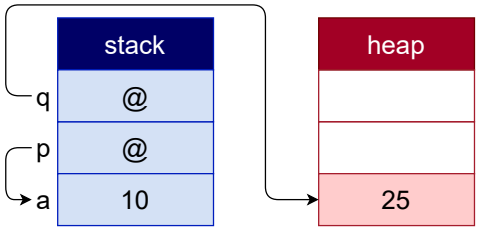
Representation



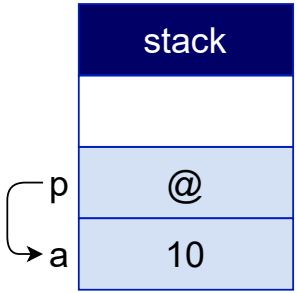
Normal Pointers

A normal pointer (also known as a raw or regular pointer) can reference data located on the heap, on the stack, or in other memory regions.

- **Stack-allocated data**: Regular pointers can also point to local variables on the stack, but the stack memory is automatically cleaned up when the function where the variable is declared exits, potentially leaving the pointer dangling.
- **Heap-allocated data**: Regular pointers are commonly used to point to dynamically allocated memory, which resides on the heap. This memory persists until it is manually deallocated.



```cpp
int a{10};
int *p{&a};
std::cout << &a << '\n';
std::cout << p << '\n';
```



? What is the type of &a?

```
int a{10};
std::cout << typeid(&a).name() << '\n';
```

```
./lecture3_cpp | c++filt -t
```

? What is the type of p?

```
int *p;
std::cout << typeid(p).name() << '\n';
```

```
./lecture3_cpp | c++filt -t
```

The Address-of Operator &

The address-of operator & does not return the address of its operand as a literal. Instead, it returns a pointer containing the address of the operand, whose type is derived from the argument, e.g., taking the address of an `int` will return the address as a pointer to `int`.

The following should make more sense now.

```
int a{10};
int *p{&a};
```

> The code snippet below is correct. However, at first glance it is not obvious that p is a pointer to an `int`. It is better to write `int *p{&a}` instead of `auto p{&a};`

```
auto a{10};
auto p{&a};
```

```
type* identifier;
```

- `type` – Although all pointers hold the same type of data (memory address), we still need to provide a type to a pointer.
- The **dereference operator** * tells the compiler this variable stores a memory address.
- `identifier` – A pointer is a variable, so its identifier must follow the convention for regular variable naming.

You read a pointer declaration inside-out (from right to left).

```
int *a;       //a is a pointer to an int
double * b;   //b is a pointer to a double
//bad practice
int* c, d;    //c is a pointer to an int, d is a regular int
```

If you want to change the * notation:

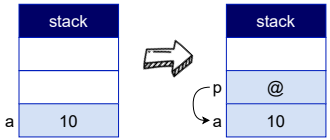
Ctrl+, → **C_Cpp** ❯ **Vc Format** ❯ **Space: Pointer Reference Alignment**


All pointers must either point to a valid object or be null.

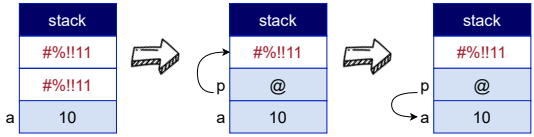
Valid Objects

A valid object is data for which memory has not been deallocated.

```
int a{10};  // valid object
int *p{&a}; // p points to a
```




```
int a{10};  // valid object
int *p; // p points to garbage
p = &a; // p points to a
```



Null Pointer

A null pointer is a value saved for indicating that the pointer does not refer to a valid object. A **null pointer** points to an unmapped or reserved memory location that can not be corrupt.

- Based on the documentation, the following ways can be used to initialize null pointers.

```
int *p1{nullptr};   //nullptr literal (from C++)
int *p2{NULL};      //NULL macro (from C)
int *p3{0};         //value initialization
int *p4{};          //zero initialization
```

Use the literal `nullptr` to initialize a pointer to null.

- A null pointer should not be confused with an uninitialized pointer: A null pointer is guaranteed to compare unequal to any pointer that points to a valid object.

```
int a{3};
int *p1{&a};
int *p2{nullptr};
if (p1≠p2)
    std::cout << "p1 is not null\n";
else
    std::cout << "p1 is null\n";
```

The Indirection Operator

In computing, indirection refers to the capability to reference a value by a pointer rather than directly. The term **indirection operator** (or **dereference operator**) usually pertains to the dereference operation that **retrieves the value pointed to by a pointer**.

```
int a{10};
std::cout << &a << '\n';          // 0×7fffffffdb3c
std::cout << *(&a) << '\n';       // What is the output?

int *p{&a};
std::cout << p << '\n';       // 0×7fffffffdb3c
std::cout << *p << '\n';      // What is the output?
```

Modify a Value

Besides getting the value stored at a specific address, the **indirection operator** is also used to modified the value stored at this address.

```
int a{10};
int *p{&a};

// Modify the value of a through p
*p = 20;

std::cout << a << '\n';     // 20
std::cout << *p << '\n';    // 20
```

Exercise #1

Trace the code.

The Operator * has multiple purposes:

- Multiplication: `int a{2 * 3};`
- Declare/initialize a pointer: `int* p{&a};`
- Dereference a pointer: `std::cout << *p << '\n';`

Exercise #2

What are the printed values?

```cpp
int a{10};
int *p{&a};

*p *= 2;

std::cout << a << '\n';
std::cout << *p << '\n';
```

```cpp
int a{2};
int *p{&a};
int **q{&p};
int ***r{&q};

std::cout << *p << '\n';
std::cout << **q << '\n';
std::cout << ***r << '\n';
```

Size of Pointers

Pointers always hold the same type of data: **Memory addresses**. Therefore, all pointers have the same size. The size of a pointer is dependent upon the architecture the executable is compiled for.

```cpp
int i{10};
double d{10.0};
float f{10.0f};
char c{'a'};

int *p{&i};
double *q{&d};
float *r{&f};
char *s{&c};

std::cout << sizeof(p) << '\n';
std::cout << sizeof(q) << '\n';
std::cout << sizeof(r) << '\n';
std::cout << sizeof(s) << '\n';
```

Typed Pointers

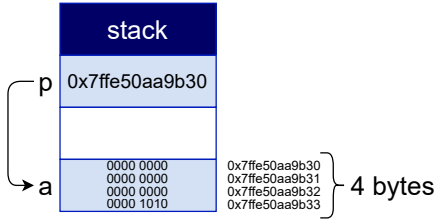
Although pointers always hold the same type of data (memory addresses), we still need to provide a type to a pointer.

The compiler ensures that there is a match between the type of the pointer and the variable address the pointer is being assigned to.

```
int a{5};
double b{2.5};
int *p{nullptr};   // OK
p = &a;            // OK
p = &b;            // Error
```

The type of a pointer tells the compiler how to derefence a pointer.

```
int a{10};
int *p{&a};
std::cout << *p << '\n';
```



Example

```cpp
int a{5};
int b{5};

int *p1{&a};  // p1 points to a
int *p2{&b};  // p2 points to b
int *p3{&a};  // p3 points to a

std::cout << std::boolalpha;    // print bools as true/false
std::cout << (p1 == p2) << '\n';
std::cout << (p1 == p3) << '\n';
```

Exercise #3

Trace the code.

Example

```cpp
int a{10};
int b{20};

int *p1{&a};
int *p2{&b};
int *p3{&a};

std::cout << std::boolalpha;  // print bools as true/false
std::cout << (*p1 == *p2) << '\n';
std::cout << (*p1 == *p3) << '\n';
std::cout << (*p1 > *p2) << '\n';
std::cout << (*p1 ⩾ *p2) << '\n';
std::cout << (*p1 < *p2) << '\n';
std::cout << (*p1 ⩽ *p2) << '\n';
std::cout << (*p1 ≠ *p2) << '\n';
```

Exercise #4

Trace the code.

Exercise #5

Write the code to display the value of each pointer and the value of the pointed-to data.

```
int a{10};
int b{20};

int *p1{&a};
int *p2{&b};
int *p3{&a};

// write your code here
```

Const-Correctness

Pointers have two modes of “constness”:

1. Pointers that do not allow modifications to the data.
2. Pointers that must always point to the same address.

- Pointer to constant data.
  - Modification of the data through the pointer is not allowed.
  - The declaration of **const** requires the **const** precede the *.

```
const type* variable; //inside-out: pointer to const (preferred)
type const* variable; //inside-out: pointer to const
```

- Pointers with constant memory address.
  - The memory address of the pointer is constant. Because the address is **const**, the pointer must be assigned a value immediately.

```
type* const variable {address}; //inside-out: const pointer
```

- Pointers to constant data with constant memory address.

```
const type* const variable; //inside-out: const pointer to const
type const* const variable; //inside-out: const pointer to const
```

Example

```
int a{2};
int b{3};

/* pointer to const */
const int *p1;
p1 = &a; // OK
p1 = &b; // OK
*p1 = 3; // Error

/* const pointer */
int *const p2{&a};
*p2 = 3; // OK
p2 = &b; // Error

/* const pointer to const */
const int *const p3{&a};
*p3 = 3; // Error
p3 = &b; // Error
```

Wild Pointers

A wild pointer is a pointer that has not been initialized to a known memory address. When a pointer is declared but not assigned a specific address (either of a variable or nullptr), it holds a **garbage** value. This garbage value is essentially a random memory address that could be pointing anywhere in your program's address space—to valid data, to code, or to memory that is not even allocated to your program.

Example

```cpp
int *p; // p is a wild pointer. It holds a garbage memory address.

// The following line is UNDEFINED BEHAVIOR.
// We are attempting to write the value 100 to an unknown memory address.
*p = 100;
// The program might crash here, or it might seem to continue,
// having corrupted some unknown part of memory.
std::cout << "This line may or may not be reached.\n";
```

Dynamic Memory Allocations

Dynamic Memory Allocations

Dynamic memory allocation refers to the process of reserving and freeing memory during the runtime of a program, as opposed to static or automatic memory allocation which is determined at compile-time.

This mechanism allows programs to handle variable amounts of data, which might not be possible with fixed-size arrays or data structures.

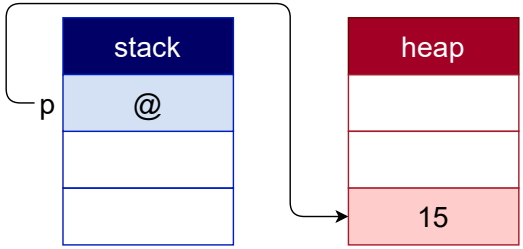
Do not use raw pointers for dynamic memory allocations, use smart pointers instead.

The Operator new

The operator **new** is used for dynamic memory allocation, which allocates memory on the heap during runtime. The memory allocated by **new** remains allocated until explicitly deallocated using the operator **delete** or until the program ends.

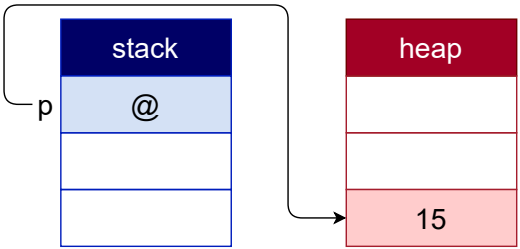
R.3: A raw pointer (a T*) is non-owning

```
int *p = new int{15};   or   int *p{new int{15}};
```



Unnamed Resource

We are requesting an `int`'s worth of memory from the OS. The `new` operator creates the object and then **returns a pointer** containing the address of the memory that has been allocated.



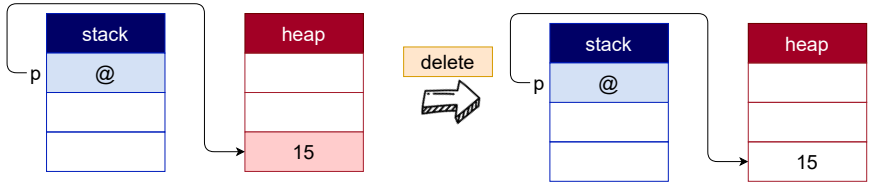
Dynamically allocated objects are never variables. They do not have an identifier and they can never go out of scope. The only way you can handle them is via a pointer that you get at the time of allocation.

The Operator `delete`

The **`delete`** operator is used to deallocate memory that was previously allocated by the **`new`** operator. Reclaiming memory is crucial to prevent **memory leaks**, which can lead to unnecessary memory consumption and potential program errors.

When you are done working with the resource in the heap, you **MUST** release it with the operator **`delete`**.

```
int* p{new int{15}};
/* do some work with p */
delete p; // release resource
```



The `delete` keyword does not delete anything!

- Calling **`delete`** simply deallocates memory.
  - The OS is free to reassign that memory to another application (or to this application again later).
- Calling **`delete`** does not destroy the pointer variable either. The pointer variable is destroyed when it goes out of scope.
- After a call to **`delete`**, the pointer points to a deallocated memory location.
  - This pointer is now a **dangling pointer**. Dangling pointers are not an issue if you do not dereference them.

```
int* p{new int{15}};
std::cout << p << '\n'; // 0×55555556b2b0
delete p;
std::cout << p << '\n'; // 0×55555556b2b0
std::cout << *p << '\n';// UB
```

Reassigning a pointer

After calling **delete**, you can reassign the pointer to point somewhere else. If you are not planning to reuse the pointer, assign it to **nullptr** (to be safe).

```
int *p{new int{5}};   // allocate and point to data on the heap
delete p;             // free the heap memory
int a{2};             // create a is on the stack
p = &a;               // point to data on the stack
p = new int{3};       // allocate and point to data on the heap
delete p;             // free the heap memory
p = nullptr;          // null pointer
```

Calling `delete` on Stack Resource

Calling **`delete`** on a pointer pointing to resource on the stack is **UB**.

```
int a{3};
int *p{&a};
delete p;  // UB
```


Only delete what you new!!!

Calling `delete` on a `nullptr`

§8.3.5/2 It is safe to call `delete` on a `nullptr`. This is a crucial and intentional feature of the C++ language. It allows you to safely clean up pointers without needing to know if they are "active".

> Calling `delete` on a null pointer simply does nothing.

Example

```cpp
int *p{nullptr};
delete p;  // safe to delete a null pointer
```

Issues with Dynamic Memory Allocations

Dynamic memory allocations with raw pointers may lead to many problems if you are not careful enough.

Issues you may encounter with raw pointers:

- Dangling pointer dereferencing.
- Memory leaks.
- Double call to `delete`.
- Null dereferencing.

Dangling Pointer Dereferencing

A dangling pointer is a pointer that points to a memory location that has been deallocated. The pointer itself still holds the address of that memory, but the data at that address is no longer valid or guaranteed to be what you expect. The memory it points to might have been returned to the operating system or re-allocated for a completely different purpose.

Dereferencing a dangling pointer results in **UB**.

```cpp
int *p{new int{2}};
delete p;   // p is dangling
*p = 5;     // UB
std::cout << *p<< '\n';    // UB
```

Common Causes of Dangling Pointers

Here are the three most common ways dangling pointers are created:

1. **Deallocating Memory with `delete`**: You allocate memory, you free it, but you forget to nullify the pointer. The pointer now dangles.
   - **Example:** See previous slide.
2. **Pointer to a Variable That Goes Out of Scope**: A pointer can be created to a variable inside a smaller scope (e.g., inside an `if` statement, a `for` loop, a function). Once the program leaves that scope, the pointer dangles.

```cpp
int main() {
    int* p = nullptr;

    { // Inner scope starts
        int inner_variable{5};
        p = &inner_variable;
        std::cout << "Inside scope: " << *p << '\n';
    } // Inner scope ends, `inner_variable` is destroyed.
    // UB: Accessing memory that is out of scope.
    std::cout << "Outside scope: " << *p << '\n';
}
```

How to Safely Deal with Dangling Pointers?

1. The "Old School" C-Style Discipline: Set to **`nullptr`**
   - The most basic defensive measure is to set a pointer to **`nullptr`** immediately after you deallocate the memory it points to.

```cpp
int* p{new int(42)};
// ... use p ...

delete p;       // Deallocate the memory
p = nullptr;    // Set the pointer to null, preventing it from dangling

// Any check like `if (p ≠ nullptr)` will correctly show the pointer is invalid.
// Attempts to dereference it now is a guaranteed crash, not silent corruption.
```

2. The Modern C++ Solution: Smart Pointers (Highly Recommended)
   - Smart pointers are objects that wrap a raw pointer and automatically manage its memory lifecycle using a principle called RAII (Resource Acquisition Is Initialization).

Memory Leak

A **memory leak** occurs when a computer program incorrectly manages memory allocations in a way that memory which is no longer needed is not released.

- A memory leak reduces the performance of the computer by reducing the amount of available memory. Eventually, all or part of the system stops working correctly, the application fails, or the system slows down vastly.

Always fix memory leaks even if your program runs for a short time.

```
for (int i{0}; i < 100000; ++i) {
    // In each iteration, we allocate a new integer.
    // But we never deallocate the memory from the previous iteration.
    int* p{new int(i)};
}
```

Double Call to `delete`

Trying to free memory that has already been freed results in **UB**.

- Trace the following code on paper.

```
int *p1{new int{2}};
int *p2{p1};

delete p1;
p1 = nullptr;

delete p2;  // UB
```

Exercise #6

Trace the code.

Null dereferencing

Dereferencing a null pointer is **UB**.

```cpp
int *p{nullptr};
std::cout << *p << '\n';  // UB
```

Check a pointer is not null before derefencing it.

```cpp
int *p{nullptr};
if (p) { // if p is not null
    std::cout << *p << '\n';  // it's ok to derefence it
} else { // if p is null
    // print error, assign pointer, do nothing at all, etc
    // but do not derefence it
    std::cout << "pointer is null\n";
}
```

How to minimize these issues?

- There is no possibility to know if a memory location has been freed. Only with the use of tools and best practices you can avoid issues related to already freed memory.
  - Use Valgrind (or other tools) to check issues with your code.
  - Make your pointers point to null after calling `delete`. We saw that calling `delete` on a null pointer has no effect.
  - Do not use raw pointers to allocate memory on the heap (use smart pointers instead).

References

References

A reference is an alias or an alternate name for an already existing variable.

Once a reference is established, both the reference name and the variable name can be used interchangeably to refer to the same memory location.

```
type_name& reference_name{existing_variable};
```

Usage

- **Functions** – Pass and return by reference.
- **Range-based `for` loops.**

Characteristics

- A reference must be initialized to an existing variable.

```
int& ref{};  // error
```

- A reference is an alias for an existing variable.

```
int a{10};
int& ref{a};                // ref is a reference to a
ref = 20;                   // a is now 20
std::cout << a << '\n';     // 20
a = 30;                     // ref is now 30
std::cout << ref << '\n';   // 30
```

- A reference does not have its own identity.

```
std::cout << &a << '\n';    // 0×7fffffffdadc
std::cout << &ref << '\n';  // 0×7fffffffdadc
```

- A reference cannot be reseated or made to refer to a different variable after its initial binding. It is bound to its initial variable for life.

```
int b{3};
ref = b;    //This assigns the value of b to a
```

- References cannot be null.

Pointers vs. References

What is the shared purpose of pointers and references in C++?

```
// References
int a{10};
int& r{a};
r = 15;

// Pointers
int b{20};
int *p{&b};
*p = 25;
```

Pointers vs. References

Feature	Normal Pointers	References
What It Is	A variable that stores the memory address of another variable. It is a distinct object in memory.	An alias or an alternative name for an already existing variable. It does not represent a separate object.
Declaration & Initialization	Can be declared first and initialized later. An uninitialized pointer is a "wild pointer" and is dangerous.	Must be initialized at the time of declaration. You cannot have an uninitialized reference.
Reassignment	A pointer can be reassigned to point to different variables or memory locations after it has been initialized.	A reference cannot be reseated or made to refer to a different variable after its initial binding.
Nullability	A pointer can be null, meaning it doesn't point to any valid memory location. This is a common state to represent an "empty" or "invalid" pointer.	A reference cannot be null. It must always refer to a valid, existing object. This makes them inherently safer for some use cases as you don't need to check for nullness.
Usage	Requires an explicit dereference operator (`*`) to access the value of the variable it points to.	Acts exactly like the original variable.
Memory	A pointer has its own memory address and size.	A reference typically does not occupy separate memory (it's just an alias).

Most C++ compilers implement references using pointers.

When you create a reference `int& ref{x};`, the compiler will typically allocate memory for a pointer "under the hood", store the address of `x` in it, and make sure that pointer is constant (i.e., it cannot be changed to point to something else).

Your C++ Code	What the Compiler Often Generates (Conceptually)
`int& ref{x};`	`int* const ptr{&x};` (A constant pointer)
`ref = 20;`	`*ptr = 20;` (Automatic dereferencing)
`int y{ref};`	`int y{*ptr};` (Automatic dereferencing)

Summary

Pointers & References

For observing or accessing memory you don't own (e.g., on the stack).

- **Prefer references** when possible.
  - **Safer**: Cannot be null, cannot be reseated.
  - Cleaner syntax.
- **Use normal pointers** only when necessary.
  - You need nullability (optional resource).
  - C-style array manipulation or APIs.

Dynamic Memory

For creating objects you own on the heap.

- **Do NOT use raw pointers** for dynamic memory allocation.
  - Manual `new` and `delete` leads to memory leaks, dangling pointers, and double-frees.
- **Always use smart pointers** or other RAII types.
  - `std::unique_ptr` for exclusive ownership.
  - `std::shared_ptr` for shared ownership.

R.11: Avoid calling new and delete explicitly

Exercise #7

Identify the issues in the following code snippet.

```
1  int* p1{new int(10)};
2  int* p2{new int(20)};
3  int& ref{*p1};
4
5  ref = *p2;
6  *p2 = 30;
7  p1 = new int(40);
8  delete p2;
9  *p1 = *p2;
10 int* p3{p2};
11 ref = 50;
12 delete p3;
```

Next Class

- Quiz#2
- Lecture4: STL Containers