# C Memory Management

- Memory allocation in C:
  - `calloc()`
  - `malloc()`
  - `realloc()`
  - Deallocated using the free() function.
- Memory allocation in C++
  - using the new operator.
  - Deallocated using the delete operator.

# C Memory Management

In C programming, memory management is a crucial aspect, primarily involving the manual handling of dynamic memory allocation, reallocation, and deallocation. Memory management allows you to control how memory is allocated and freed, which can optimize program performance and prevent memory leaks.

**Dynamic Memory Allocation:**

**malloc()**

void* malloc(size_t size);

Allocates a block of size bytes and returns a pointer to the beginning of the block. If the allocation fails (e.g., due to insufficient memory), malloc returns NULL. The memory is uninitialized, so it might contain garbage values.

**int* ptr = (int*) malloc(10 * sizeof(int));** // allocates memory for 10 integers

**calloc():**

void* calloc(size_t num, size_t size);

Allocates memory for an array of num elements, each of size bytes, and initializes all bytes to zero. Returns NULL if it fails to allocate the requested memory.

**int* ptr = (int*) calloc(10, sizeof(int));** // allocates and initializes memory for 10 integers

**realloc():**

void* realloc(void* ptr, size_t new_size);

Resizes a previously allocated block to new_size. If new_size is greater, it may move the block to a new location and leave the new memory uninitialized. If the block is moved, the old memory is freed.Passing NULL acts like malloc, and new_size 0 acts like free.

**int* ptr = (int*) malloc(5 * sizeof(int));**
**ptr = (int*) realloc(ptr, 10 * sizeof(int));** // resizing the block to hold 10 integers

**free():**

void free(void* ptr);

Deallocates the memory previously allocated with malloc, calloc, or realloc.
After calling free, the pointer is no longer valid (dangling pointer). It's good practice to set it to NULL.

**free(ptr);**
**ptr = NULL;** // avoids a dangling pointer

**Common C Memory Management Errors:**

Memory management in C is prone to subtle errors that can lead to crashes, undefined behavior, and security vulnerabilities. Here's a breakdown of common memory management errors, along with examples and solutions for each.

1**. Initialization Errors**

Uninitialized variables or memory can contain unpredictable "garbage" values, leading to unexpected behavior or security issues.

- Unpredictable results when reading uninitialized variables.
- Crashes or incorrect calculations.

Example:

```
int x; // Uninitialized
int result = x + 10; // Undefined behavior
```

Solution is

```
int x = 0; // Initialized
```

2. **Failing to check Errors:**

Many functions in C, such as malloc(), can fail and return NULL when they're unsuccessful. Not checking return values can lead to dereferencing invalid pointers, causing crashes or memory corruption.

Example:

```
int *ptr = malloc(sizeof(int) * 10);
*ptr = 10; // Potentially dereferencing NULL if malloc fails
```

Solution: Always Check Return Values

```
int *ptr = malloc(sizeof(int) * 10);
if (ptr != NULL) {
    *ptr = 10;
} else {
    // Handle allocation failure
}
```

3. **Dereferencing Null Pointers:**

Dereferencing a NULL or otherwise invalid pointer leads to crashes or undefined behavior and Segmentation Faults.

Example:

```
int *ptr = NULL;
*ptr = 10; // Dereferencing NULL pointer
```

Solution: Initialize pointers to Null.Ensure pointers pointing to valid memory before accessing them.

```
int *ptr = malloc(sizeof(int));
if (ptr != NULL) {
    *ptr = 10;
}
```

**4. Referencing Freed Memory:**

Using memory after it's been freed can lead to undefined behavior and is often referred to as a dangling pointer. This can result in crashes or, worse, corruption of unrelated data.

Example:

```
int *ptr = malloc(sizeof(int));
free(ptr);
*ptr = 10; // Undefined behavior: ptr points to freed memory
```

Solution: Set pointers Null immediately after freeing them to avoid accidental access.

```
int *ptr = malloc(sizeof(int));
free(ptr);
ptr = NULL;
```

**5. Freeing memory multiple times:**

Freeing the same memory location more than once can lead to crashes or memory corruption, as the memory is no longer owned by the program after the first free().

Symptoms:

Crashes or heap corruption warnings when freeing memory.
Unpredictable behavior after freeing memory twice.

Example:

```
int *ptr = malloc(sizeof(int));
free(ptr);
free(ptr); // Double free
```

Solution:
After calling free(), set the pointer to NULL to prevent accidental double free.
Avoid using the pointer after freeing the associated memory.

**6. Memory Leaks:**

Memory leaks occur when allocated memory is not freed, which can cause a program to consume an increasing amount of memory over time, leading to slowdowns or crashes.

**Symptoms**:
Steadily increasing memory usage.
Potential crashes or performance degradation in long-running applications.

Example:

```
char *ptr = malloc(1024);
// Forget to free(ptr)
```

**7. Zero Length Allocations:**

```
int *ptr = malloc(0);
if (ptr != NULL) {
    // ptr is a non-NULL pointer, but dereferencing it is unsafe.
    // Best practice: do not dereference a zero-length memory block
}
```

**In many systems, malloc(0) returns a non-NULL pointer (often a "null pointer" that points to an empty memory block), but dereferencing or writing to this memory should be avoided.**

**7. Zero Length Allocations:**

calloc(0,n):

```
int *ptr = calloc(0, sizeof(int));
if (ptr == NULL) {
    // Allocation failed, handle error
} else {
    // Safe to skip dereferencing, as the memory block has size zero
}
```

In most systems, calloc(0, n) returns either NULL or a valid pointer (depending on the implementation), but dereferencing or accessing that memory is not valid.

**7. Zero Length Allocations:**

realloc(ptr,0):

```
int *ptr = malloc(10 * sizeof(int));
// Do some work
ptr = realloc(ptr, 0);  // Equivalent to free(ptr)
// ptr is now NULL, and the memory has been freed
```

```
#include <stdlib.h>  // For malloc, realloc, free
#include <stdio.h>

int main() {
    // Allocate memory for 10 integers
    int *ptr = malloc(10 * sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;  // Exit if malloc fails
    }

    // Some work, for example, initialize the memory
    for (int i = 0; i < 10; i++) {
        ptr[i] = i;
    }
    // Resize or free the memory by reallocating to 0 bytes
    ptr = realloc(ptr, 0);  // Equivalent to free(ptr)
    if (ptr == NULL) {
        printf("Memory successfully freed and pointer set to NULL.\n");
    }
    return 0;
```

# C++ Dynamic Memory Management

In C++, dynamic memory management refers to allocating and deallocating memory during runtime, as opposed to static memory allocation (which is done at compile time). This is necessary when the size of data structures or objects isn't known until the program is running, or when data needs to persist beyond the scope of the function that created it.

In C++, dynamic memory management is done using the heap memory, which is managed manually through new and delete operators, unlike stack memory, which is automatically managed.

**1. Allocating Memory with new and new[]**

In C++, memory is allocated dynamically using the new operator, which returns a pointer to the allocated memory. There are two versions of the new operator:

new: Used to allocate memory for a single object.
new[]: Used to allocate memory for an array of objects.

# C++ Dynamic Memory Management

**Ex:-**

```
int* ptr = new int;     // Allocates memory for a single integer
*ptr = 10;              // Assign a value to the allocated memory

// Use ptr...

delete ptr;             // Frees the allocated memory
```

**Ex:-**

```
int* arr = new int[10];  // Allocates memory for an array of 10 integers
arr[0] = 10;             // Assign values to the array

// Use arr...

delete[] arr;            // Frees the memory allocated for the array
```

# C++ Memory Allocation and Deallocation functions

```
#include <iostream>
#include <cstdlib> // For malloc/free

class MyClass {
public:
    MyClass(int val) : value(val) { std::cout << "Constructor: " << value << std::endl; }
    ~MyClass() { std::cout << "Destructor: " << value << std::endl; }

    int value;
};
int main() {
    // Using new and delete
    MyClass* obj1 = new MyClass(10);  // Allocates memory and calls constructor
    delete obj1;  // Deallocates memory and calls destructor

    // Using new[] and delete[]
    MyClass* arr1 = new MyClass[3]{ MyClass(1), MyClass(2), MyClass(3) };
    delete[] arr1;  // Deallocates memory for array of objects

    // Using malloc and free (C-style)
    MyClass* obj2 = (MyClass*)malloc(sizeof(MyClass));  // Allocates memory, but doesn't call constructor
    if (obj2) {
        new (obj2) MyClass(20);  // Manually call the constructor
        free(obj2);  // Deallocates memory, but doesn't call destructor
    }

    return 0;
}
```

# C++ Memory Allocation and Deallocation functions

C++ does not have built-in garbage collection in the way languages like Java or C# do. In C++, memory management is explicit, meaning that the programmer is responsible for allocating and deallocating memory.

While this provides greater control over performance and memory usage, it also introduces the risk of memory-related bugs such as memory leaks, dangling pointers, and double frees.

# Common Errors with C++ Dynamic Memory Management

In C++, failing to correctly check for allocation failure is a critical error that can lead to undefined behavior, crashes, or memory corruption. Properly checking the result of dynamic memory allocations is essential to ensure that the program can handle situations where the system is unable to allocate the requested memory.

**Failing to correctly check for Allocation failure**

In C++, the new operator is supposed to throw a std::bad_alloc exception if it fails to allocate memory. However, if you don't catch the exception, it can cause your program to terminate unexpectedly.

```
int* ptr = new int[1000];  // If allocation fails, std::bad_alloc is thrown.
```

**Solution:**

```
try {
    int* ptr = new int[1000];
    // Use the allocated memory
} catch (const std::bad_alloc& e) {
    std::cerr << "Memory allocation failed: " << e.what() << std::endl;
    // Handle the failure (e.g., cleanup, exit, etc.)
}
```

# C++ Dynamic Memory Management

**Improperly Paired Memory Management Functions:**

- Improperly pairing memory management functions is one of the most common and dangerous errors in C++ programming, leading to undefined behavior, crashes, and memory corruption.
- In C++, when you allocate memory using one function, you must deallocate it using the corresponding function.
- Mixing up memory management functions, such as using malloc with delete, or using new with free, leads to undefined behavior, as these functions are not designed to work together.

# C++ Dynamic Memory Management

Ex:-

```
int* ptr = (int*)malloc(sizeof(int) * 10);  // Allocate with malloc
delete ptr;  // Error: malloc + delete is undefined behaviour
```

```
int* ptr = new int[10];  // Allocate with new[]
free(ptr);  // Error: new[] + free is undefined behavior
```

# C++ Dynamic Memory Management

**Table 4.1** Memory Function Pairings

| Allocator | Deallocator |
|---|---|
| `aligned_alloc()`, `calloc()`, `malloc()`, `realloc()` | `free()` |
| `operator new()` | `operator delete()` |
| `operator new[]()` | `operator delete[]()` |
| Member `new()` | Member `delete()` |
| Member `new[]()` | Member `delete[]()` |
| Placement `new()` | Destructor |
| `alloca()` | Function return |

# C++ Dynamic Memory Management

**Freeing memory multiple times:**

```
int* ptr = new int[10];
delete[] ptr;  // First free
ptr = nullptr;  // Nullify the pointer to avoid accidental second free

// Later, this is safe, even if delete is called again:
delete[] ptr;  // Does nothing, as ptr is nullptr
```

# C++ Dynamic Memory Management

Deallocation function throws an exception:

- When a deallocation function throws an exception, it can lead to undefined behaviour, resource leaks, and other serious issues.
- In C++, memory deallocation (using delete, delete[], or free()) is expected to be a side-effect-free operation, meaning that it should not throw any exceptions.
- The primary purpose of deallocation functions is to return memory to the system without introducing new errors.
- If these functions do throw exceptions, it complicates resource management and can result in unintended consequences.

# C++ Dynamic Memory Management

Why Deallocation function should not throw an exception**?**

- Unpredictability: If deallocation throws an exception, it can leave your program in an inconsistent state. This is especially problematic in resource management scenarios where multiple resources (memory, file handles, network connections) need to be cleaned up. If one cleanup operation fails, it could make it impossible to clean up others, leading to resource leaks.

- Stack Unwinding: In C++, when an exception is thrown, the runtime starts stack unwinding, which means that all the local objects in the stack will be destroyed (their destructors will be called). If a destructor (or any other cleanup function) throws an exception during unwinding, it will call std::terminate, causing the program to terminate unexpectedly.

- Unclear State: If the deallocation function throws, the pointer that was previously freed is now in an invalid state. Re-using or re-deleting the pointer after this can lead to undefined behavior or memory corruption.

# C++ Dynamic Memory Management

Deallocation function throws an exception**:**

```
class MyClass {
public:
    MyClass() {
        data = new int[10];
    }

    ~MyClass() noexcept {  // Marking as noexcept ensures no
exceptions are thrown
        delete[] data;  // This should not throw
    }

private:
    int* data;
};
```

Deallocation function throws an exception:

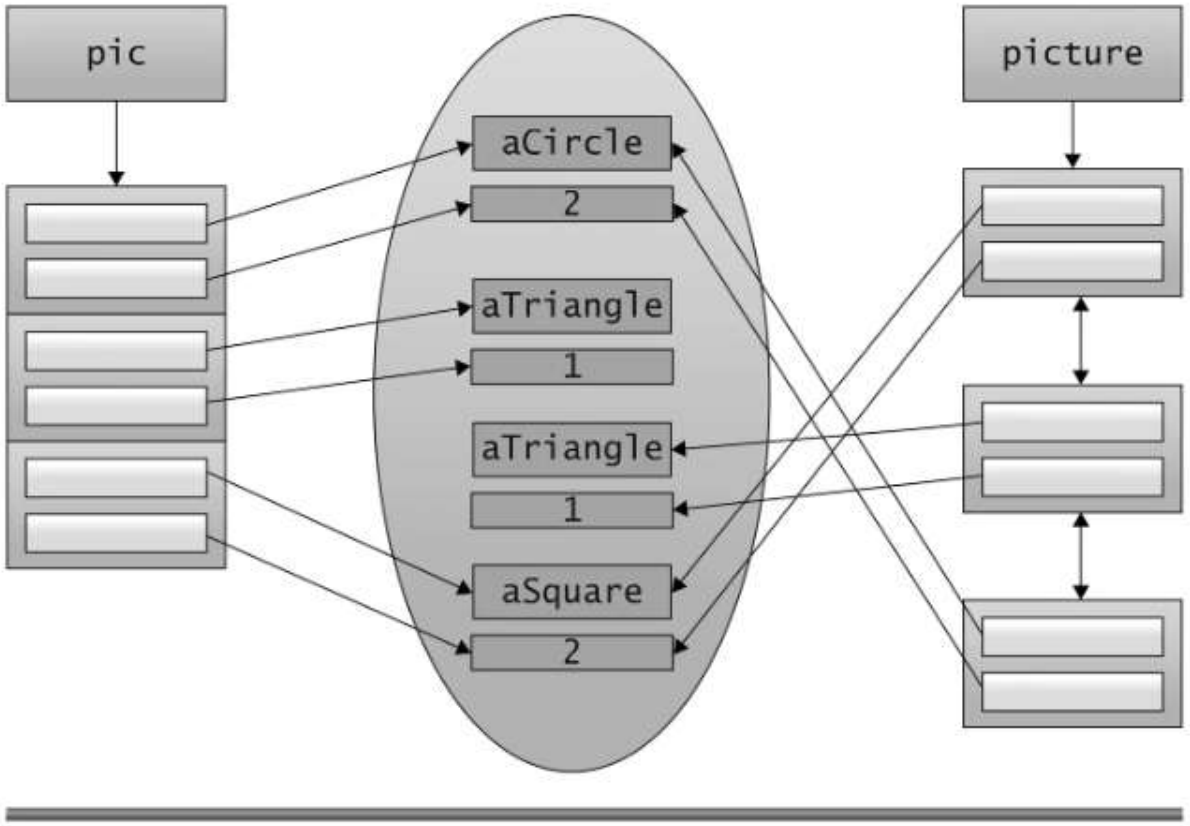


**Figure 4.2** The pic vector and the picture list with the pool of shared reference-counted objects