# Memory, Pointers and References

C++ for Developers

lafftale
for developers

# Content

- Memory
- Pointers
- References
- RAII

# How is a program run?

C++ for Developers

# Memory

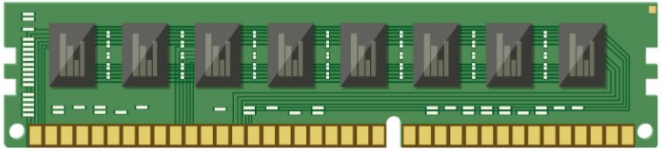
- Primary vs Secondary Memory
- Program Memory structure
- Stack
- Heap

**Primary Memory**
Volatile memory, meaning that it forgets everything if power is lost.

**RAM**
Random Access Memory

RAM is ultra-fast memory that the CPU can access directly. When a program is loaded, the operating system assigns it a specific section of RAM to use.

The program can only read and write within its allocated space; if it tries to access memory outside of this area, the OS will block it and raise an exception (e.g., a segmentation fault).



**Secondary Memory**
Non-volatile, meaning it's persistant through turning the computer on / off.

| **SSD** Solid State Drive | | | **HDD** Hard Disk Drive |
|---|---|---|---|
| Faster | ✓ | ✗ | Slower |
| Shorter Lifespan | ✗ | ✓ | Longer Lifespan |
| More expensive | ✗ | ✓ | Cheaper |
| Non-mechanical (flash) | ✓ | ✓ | Mechanical (moving parts) |
| Shock resistant | ✓ | ✗ | Fragile |

Best for frequently used files and larger, more complex files like programs / operating systems.

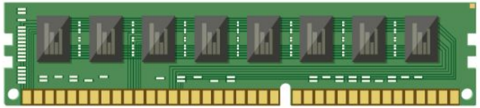
Best for storing data like movies, pictures or documents.

lafftale
for developers

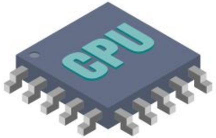
**From Secondary Storage to Running**

**1.** The program's image is copied from secondary storage

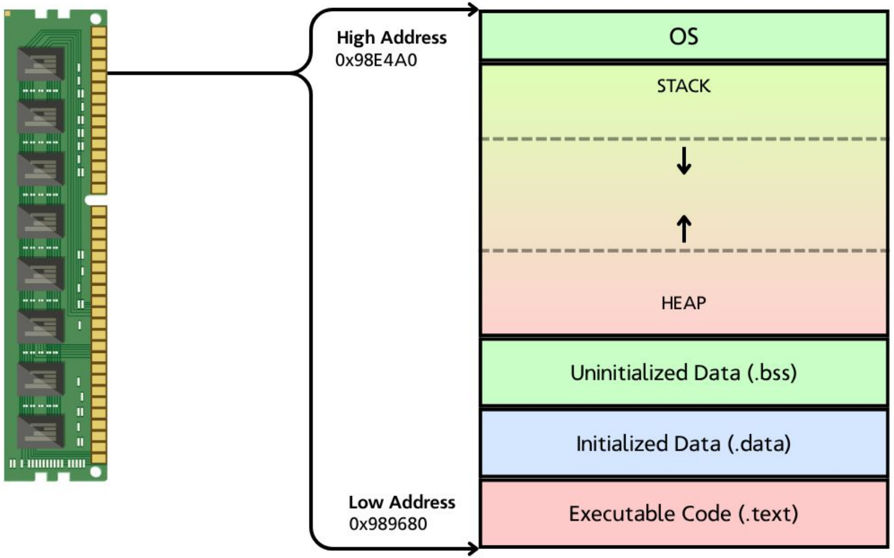
**2.** The image is loaded into RAM, and the operating system assigns it a specific memory space.



**3.** The CPU begins executing the program's instructions directly from RAM.



## Memory Space in RAM




Stores local variables and function call information. Memory is managed automatically. Grows and shrinks as we go in and out of scope.

Stores dynamically allocated variables (e.g., using **new** in C++). Managed manually or by RAII.

Holds uninitialized global and static variables, which are zero-initialized at program start.

Holds initialized global and static variables. These values are loaded from the program image.

Contains the program's executable machine code - the instructions the CPU executes.

# Stack

Perfect data structure

- Grows automatically when we go into scope
- Shrinks automatically when we go out of scope

The LIFO structures removes the unnecessary data we don’t need anymore.

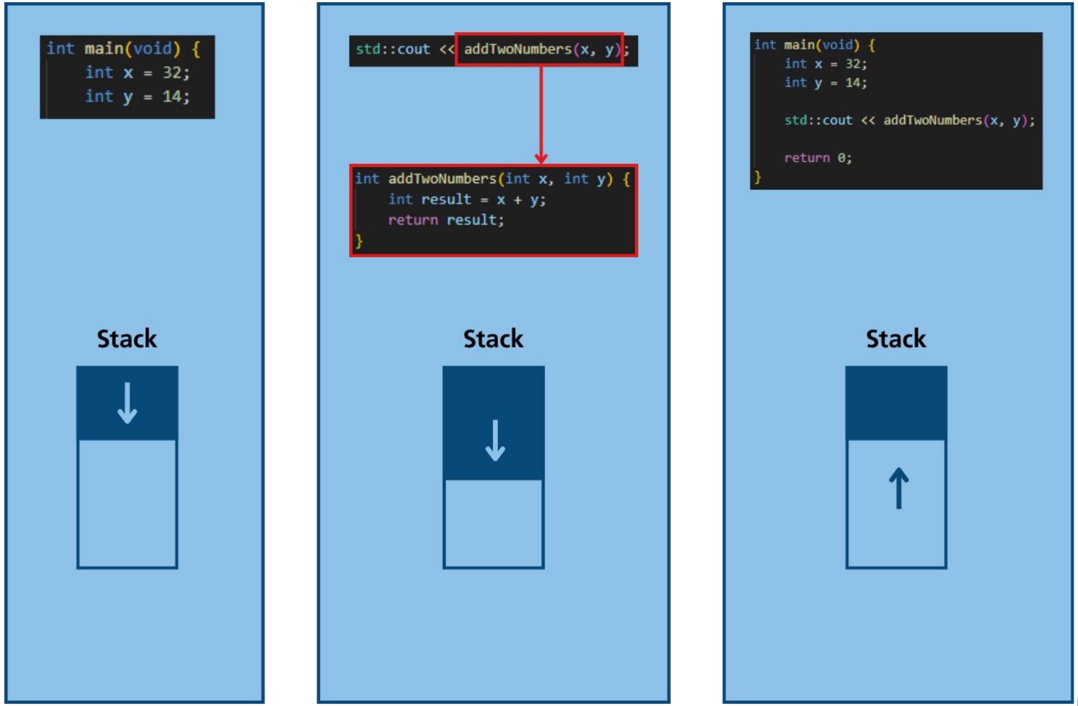
Cons

- Objects can’t grow or shrink in size dynamically on the stack.

# Stack Behaviour

```cpp
int addTwoNumbers(int x, int y) {
    int result = x + y;
    return result;
}

int main(void) {
    int x = 32;
    int y = 14;

    std::cout << addTwoNumbers(x, y);

    return 0;
}
```

# Stack Behaviour



# Heap

Used to dynamically allocate memory

- Resize objects that grows and shrinks during runtime

Cons

- Needs to free the memory explicitly
- Memory Leaks
- Fragmentation

# Accessing elements on the heap

To reach elements on the heap, we need to access them through pointers * or references **&**

A pointer or a reference, is a variable that stores the address of a variable - rather than its own value.

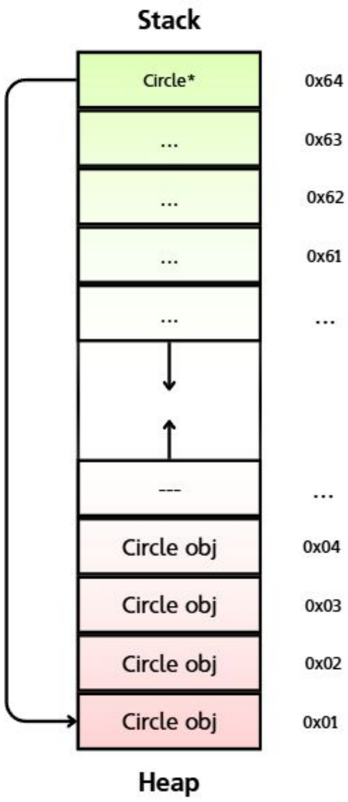
# Initializing on the heap

To explicitly place an object or basic data type on the heap, you use the **new** keyword during the initialization.

```
Circle* circle1 = new Circle(3);
```

Need to explicitly free the memory  with **delete**:

```
delete circle1;
```



# Risks of using the heap

**Memory Leaks**
When we forget to delete our objects and lose the pointer - we can't access this object anymore, but it still exists in memory.

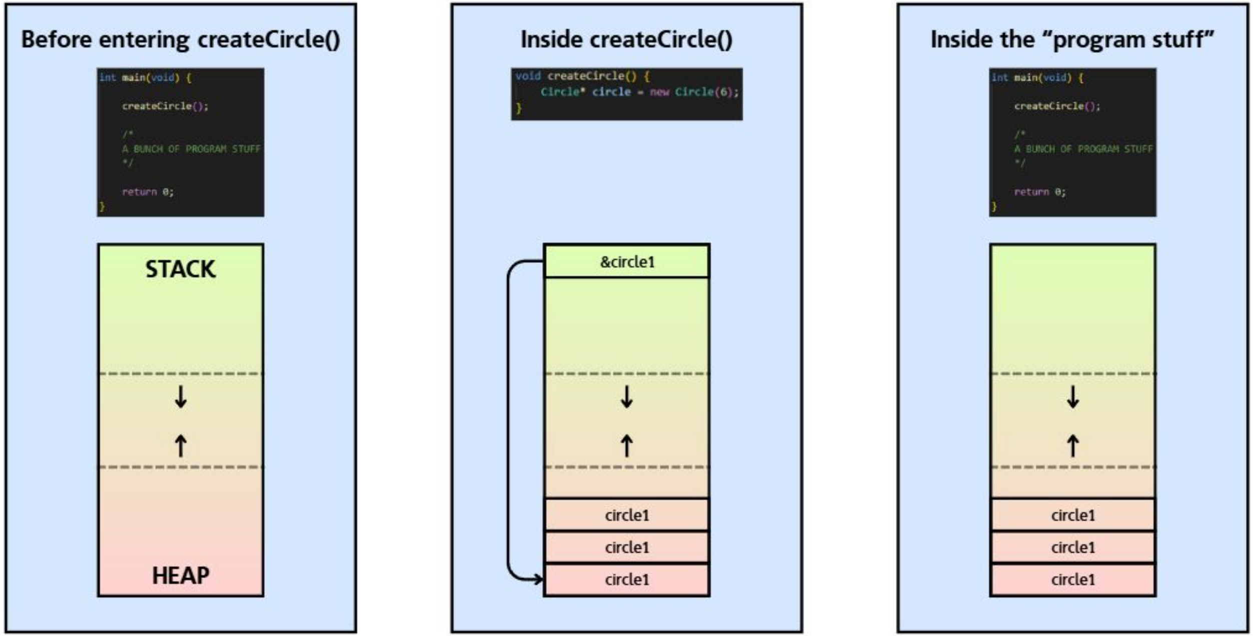
**Fragmentation**
When we allocate and delete objects on the heap frequently, we create spaces between allocated objects, *fragments*. So even if the total available space is large enough, we might not be able to fit in large data structures.

# Example Class

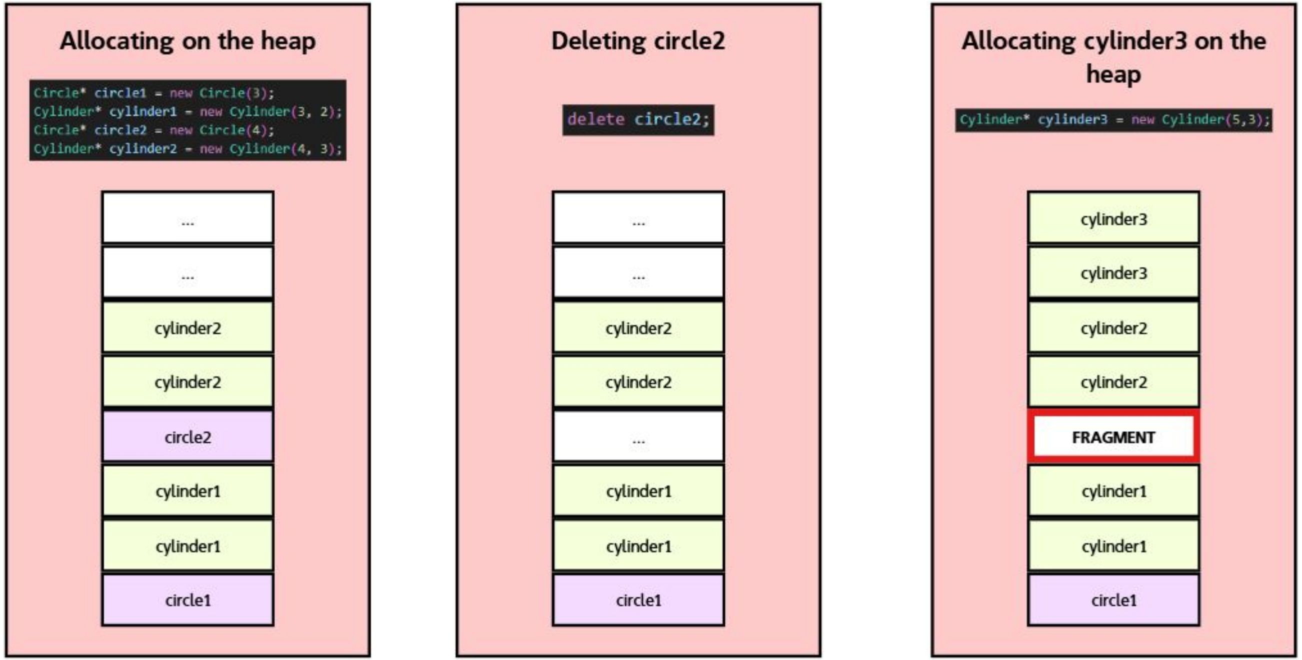
```cpp
class Circle {
private:
    double diameter;
public:
    Circle(double d) : diameter(d) {}

    double getArea() {
        double r = diameter / 2;
        return pow(r, 2) * M_PI;
    }
};
```

```cpp
class Cylinder : public Circle {
private:
    double depth;

public:
    Cylinder(double dia, double d) : Circle(dia), depth(d) {}

    double getVolume() {
        return getArea() * depth;
    }
};
```

# Memory Leak


Before entering createCircle()
int main(void) {
createCircle();
/*
A BUNCH OF PROGRAM STUFF
*/
return 0;
}
STACK
HEAP
Inside createCircle()
void createCircle() {
Circle* circle = new Circle(6);
}
&circle1
circle1
circle1
circle1
Inside the "program stuff"
int main(void) {
createCircle();
/*
A BUNCH OF PROGRAM STUFF
*/
return 0;
}
circle1
circle1
circle1

# Fragmentation




lafftale for developers

# How do we avoid these risks?

## Memory Leaks

Remember to use **delete** (or **delete[]** for arrays) or use **RAII**. (We will talk about it soon).

## Fragmentation

Plan your allocation on the heap. It's better to allocate large and flexible objects on the heap - and small fixed-size on the stack. Group related data together in containers like std::vectors

# Stack vs Heap

Objects allocated on the stack have their size and layout determined at compile time, and their lifetime is tied to scope. The stack grows and shrinks automatically.

The heap is manually managed and allows dynamic sizes and lifetimes, but is also slower and must be explicitly freed.

# Pointers

C++ for Developers

# What is a pointer?

A pointer variable stores the memory address of another variable.

A pointer is declared by adding * after the
data type it’s pointing to.

When you place **&** before a variable, you get its memory address.
This is called the “address-of” operator.

To access the value in the memory address you de-reference it by adding * before the variable.

```cpp
int main(void) {

    int age = 29;
    int* pAge = &age;
    std::cout
        << *pAge
        << '\n';

    return 0;
}
```

lafftale
for developers

# Why do we use pointers?

```cpp
class HugeClass {
public:
    std::string name;
    std::string sound;
    std::string smell;
    std::string taste;
    std::string feeling;

    HugeClass(std::string n,
        std::string s,
        std::string sm,
        std::string t,
        std::string f)
        : name(n), sound(s), smell(sm), taste(t), feeling(f)
        {}

    void print() {
        std::cout
            << name << std::endl
            << sound << std::endl
            << smell << std::endl
            << taste << std::endl
            << feeling << std::endl;
    }
};
```

```cpp
void printHugeClass(HugeClass hc) {
    hc.print();
}

void printHugeClass(HugeClass* hc) {
    hc->print();
}
```

**Copy by Value**

vs

**Copy by Reference**

```cpp
int main(void) {

    HugeClass hc = HugeClass("Carl",
        "Loud",
        "Margiela",
        "Cinnamon",
        "Smooth");

    printHugeClass(hc);
    printHugeClass(&hc);

    return 0;
}
```

lafftale for developers

# Pointer Arithmetic

```
int main(void) {

    // '\0' is called a null-terminator
    // this signals end of string
    char name[] = "Carl\0";

    char* pName = name;

    while(*pName != '\0') {
        std::cout << *pName;
        pName++;
    }

    std::cout << '\n';

    return 0;
}
```

lafftale
for developers

# nullptr

A pointer can also point to **nullptr** - this is equals to setting it to nothing. We can use this mechanism to check results from functions:

```cpp
const Attack* Attacks::getAttack(int attackId) const {
    for(const Attack& attack : this->attacks) {
        if (attack.getAttackId() == attackId) return &attack;
    }

    return nullptr;
}
```

```cpp
for (auto it = row.begin() + CHAR_ATTACK_ID_S; it < row.end(); it++) {
    if (const Attack* attack = attacks.getAttack(std::stoi(*it))) {
        loadedAttacks.push_back(*attack);
    }
    else {
        throw std::invalid_argument("Unable to find attack with id " + std::stoi(*it));
    }
}
```

# Let’s look at an example

# Let’s do some exercises!

Exercises:

- Pointers
  - #1. Pointer fundamentals
  - #2. What happened?

# References

C++ for Developers

# What is a reference?

A reference acts as an alias for an existing variable.

Once initialized, it cannot be changed to refer to another variable. It behaves similarly to a constant pointer that is automatically dereferenced.

```
int main(void) {

    int age = 29;
    int& rAge = age;

    std::cout
        << rAge
        << '\n';

    return 0;
}
```

lafftale
for developers

# Passing as arguments

Passing a const reference as argument ensures that we:

1. Saves memory by copying the reference and not the whole object
2. Do not enable the function to modify the object

```
void Battle::logAttack(const Character& attacker, const Character& defender, const Attack& attack, const int damage) {
    std::string message = attacker.getName() + " attacked " + defender.getName() +
        " with " + attack.getName() +
        " for " + std::to_string(damage) + " damage\n";

    attackHistory.push_front(message);
}
```

# As a return value

Returns a reference and not a copy. We also ensures it’s not modifiable by adding const at the beginning of the function.

This will return a read-only reference to a list of attacks.

```
const std::vector<Attack>& Attacks::getAllAttacks() const {
    return this->attacks;
}
```

# Let’s look at an example

# Let’s do some exercises!

Exercises:

- References
  - #1. Basic reference
  - #2. Reference Calculator

# RAII - Resource Acquisition is Initialization

C++ for Developers

# What is RAII?

RAII is an idiom meaning that a resource's existence is tied to the object lifetime. A resource can be:

- Dynamic memory
- File Streams
- Sockets

Without RAII, it is easy to forget to explicitly delete resources we acquire, causing memory leaks.

# Consequences without RAII

Dynamic Memory → Memory leaks

File handle → you might run out of file descriptors

Mutex → deadlock

Socket → resource exhaustion

# How do we implement RAII?

When using pointers to heap allocated objects, we use something called smart pointers!

- unique_ptr
- shared_ptr
- weak_ptr

Smart pointers automatically call the destructor of the managed object when they go out of scope, so you don't need to manually delete.

```cpp
int main(void) {

    // Sole owner of pointer
    // Destroyed when out of scope
    std::unique_ptr<int> pAge = std::make_unique<int>(29);

    // Shared owner
    // Destroyed when last owner is
    // out of scope
    std::shared_ptr<int> aGrade = std::make_shared<int>(2);
    std::shared_ptr<int> bGrade = aGrade;

    // No ownership
    std::weak_ptr<int> cGrade = aGrade;
}
```

# unique_ptr

An unique_ptr enforces exclusive ownership over an object. Trying to point two unique_ptrs to the same object will cause compile errors.

When it runs out of scope, the object's destructor will be called.

```
// Sole owner of pointer
// Destroyed when out of scope
std::unique_ptr<int> pAge = std::make_unique<int>(29);
```

# shared_ptr

A shared_ptr enables shared ownership of resources. It keeps a reference count of shared owners.

When the amount of shared owners reaches  0, it will call the objects destructor.

```
// Shared owner
// Destroyed when last owner is out of scope
std::shared_ptr<int> aGrade = std::make_shared<int>(2);
std::shared_ptr<int> bGrade = aGrade;
```

# weak_ptr

A weak_ptr doesn’t hold ownership over resources and will not call the objects destructor when out of scope.

```
std::shared_ptr<int> aGrade = std::make_shared<int>(2);
std::shared_ptr<int> bGrade = aGrade;

// No ownership
std::weak_ptr<int> cGrade = aGrade;
```

# Let’s look at an example

# Let’s do some exercises!

Exercises:

- RAII - Smart pointers
  - #1. Simple use of smart pointers
  - #2. Move ownership

# Moving forward

C++ for Developers

# Group objects on the heap inside C++ containers

When allocating objects inside containers like **std::vector** - the vector will destroy the objects inside it automatically when it runs out of scope.

# Why learn pointers and references?

More careful about using heap in embedded - use pointers and references to save memory. We will talk more about this in the Embedded Lesson!

# From now on:

- Pass large objects as references in arguments
- Return objects as references - add const if it should be read-only
- Use RAII friendly mechanisms when acquiring resources
    - Wrap objects in C++ containers like std::vector<T>
    - Use smart pointers instead of raw pointers when your objects acquires resources (ex .

You can keep using raw pointers / references for stack allocated data.

lafftale for developers

# Thank you!

Clafftale for developers