CHALMERS
UNIVERSITY OF TECHNOLOGY

Introduction to Memory Management in C++ and OpenFOAM:

(What on earth is going on with `autoPtr<>` and `tmp<>`)

CFD with OpenSource Software Course

Chalmers University of Technology

Outline

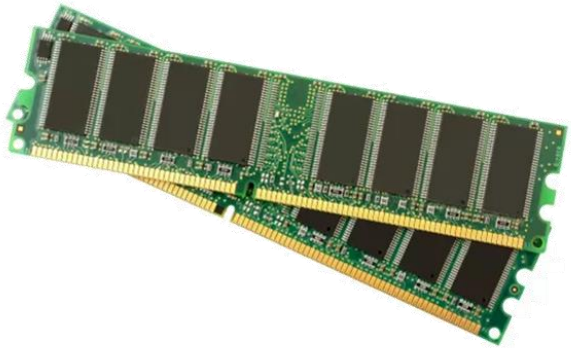
- Memory allocation
- Handling raw pointers
- Memory management in classes
- Smart pointers
- Memory management in OpenFOAM
- `autoPtr<>` and `tmp<>`

Test codes and examples are found at the GitHub repository https://github.com/salehisaeed/OSCFD

Memory Allocation

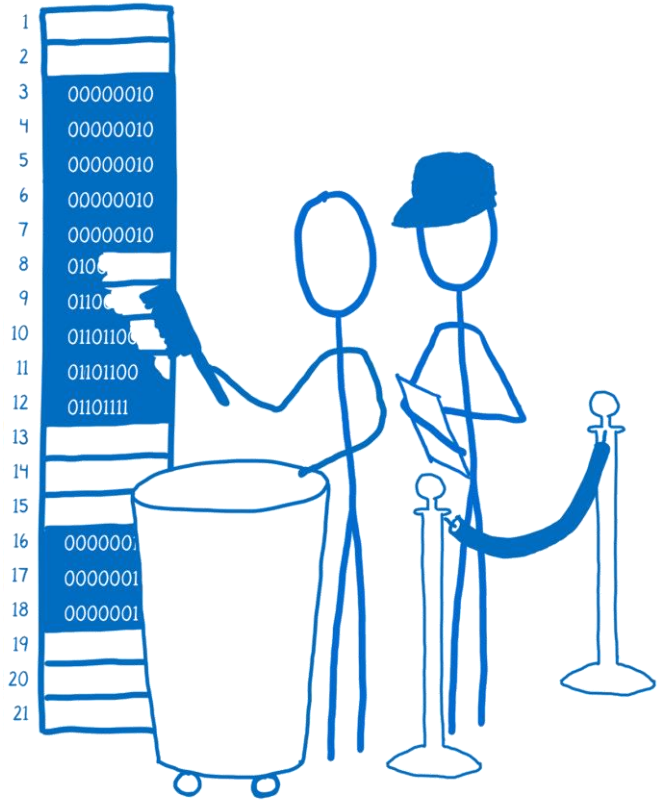
Memory

- Memory: a finite sequence of fixed-size units
- Each unit has an address (imagine a single-street city)
- Memory is not storage!
- For all data, memory must be *allocated*



Memory management

- The process of managing computer memory
- Provide ways for programs to:
  - ✓Request *dynamic memory allocation* when needed
  - ✓Release it for reuse once it's no longer in use
- We keep the address of the allocated memory in a variable, i.e., **Pointer** `int* ptr = new int;`
- When do we know the size to allocate?
  - ✓Compile time: Static allocation (on Stack)
  - ✓Run-time: Dynamic allocation (on Heap)



Heap allocation

- **Heap Allocation**: Memory is allocated on the *heap* at runtime.
- **`new`** Operator: Used to allocate memory dynamically.
- **Manual Control**: The developer controls memory size and lifetime.

`00_SimpleAllocation.cpp`

```cpp
#include <iostream>
int main()
{
    int var = 42;        // creates an integer on the stack
    int* ptr = new int; // Allocates memory for an integer on the heap
    *ptr = 42;           // Assigns a value to the allocated memory
    return 0;
}
```

- Is there any problem with this code?

Memory leak

```cpp
#include <iostream>
int main()
{
    int var = 42; // creates an integer on the stack
    int* ptr = new int; // Allocates memory for an integer
    *ptr = 42;          // Assigns a value to the allocated memory
    return 0;
}
```

```
$ g++ -g 00_SimpleAllocation.cpp -o 00_SimpleAllocation

$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
./00_SimpleAllocation &> log.00_SimpleAllocation
```

```
(...)
==2505926==    by 0x10915E: main (00_MemoryLeakSimple.cpp:4)
==2505926==
==2505926== LEAK SUMMARY:
==2505926==    definitely lost: 4 bytes in 1 blocks
==2505926==    indirectly lost: 0 bytes in 0 blocks
==2505926==      possibly lost: 0 bytes in 0 blocks
==2505926==    still reachable: 0 bytes in 0 blocks
==2505926==         suppressed: 0 bytes in 0 blocks
```

Static and dynamic allocation

Memory allocation

Static

The size is known at compile time

Dynamic

The size is determined at run-time

Static memory allocation: **Stack**

```cpp
#include <iostream>
int main()
{
    // Fixed-size array allocated statically
    int arr[5] = {1, 2, 3, 4, 5};
    return 0;
}
```

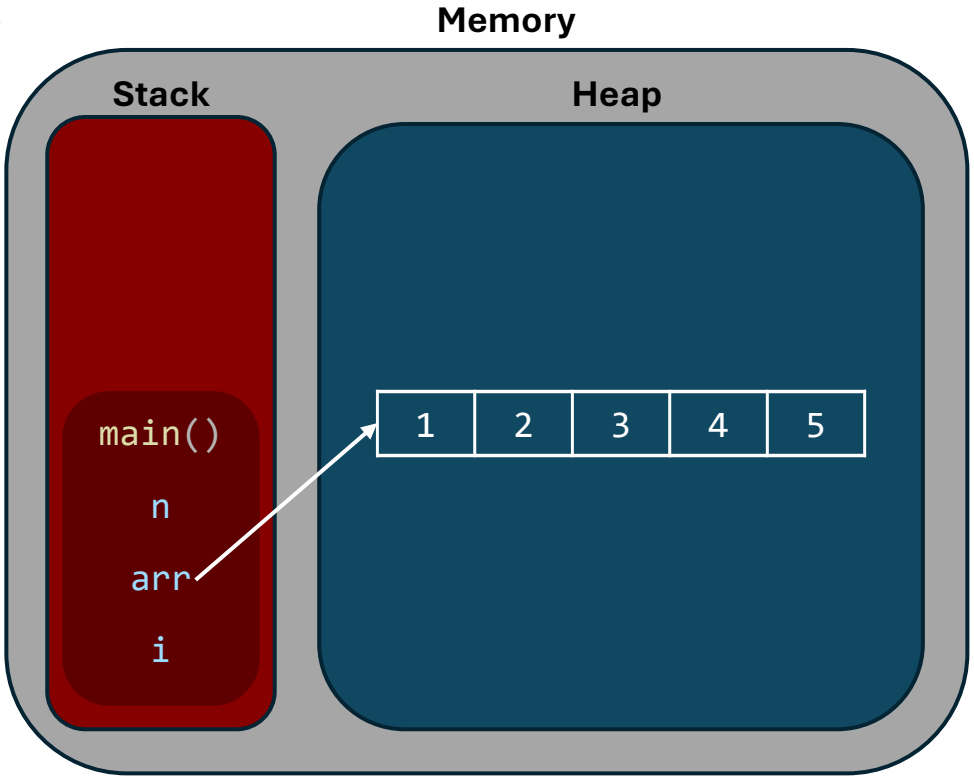
Dynamic memory allocation: **Heap**

```cpp
#include <iostream>
int main()
{
    int n;
    // Input size is determined by user
    std::cin >> n;
    // Dynamic allocation of an array
    int* arr = new int[n];
    for (int i = 0; i < 5; i++)
        arr[i] = i + 1;
    delete[] arr;
    return 0;
}
```

Memory layout

- Stack and Heap: Two key areas where memory is allocated.
- **Stack**: Stores local variables, function calls.
- **Heap**: Stores dynamically allocated memory.

```cpp
#include <iostream>
int main()
{
    int n;
    // Input size is determined by user
    cin >> n;
    // Dynamic allocation of an array
    int* arr = new int[n];
    for (int i = 0; i < 5; i++)
        arr[i] = i + 1;
    delete[] arr;

    return 0;
}
```



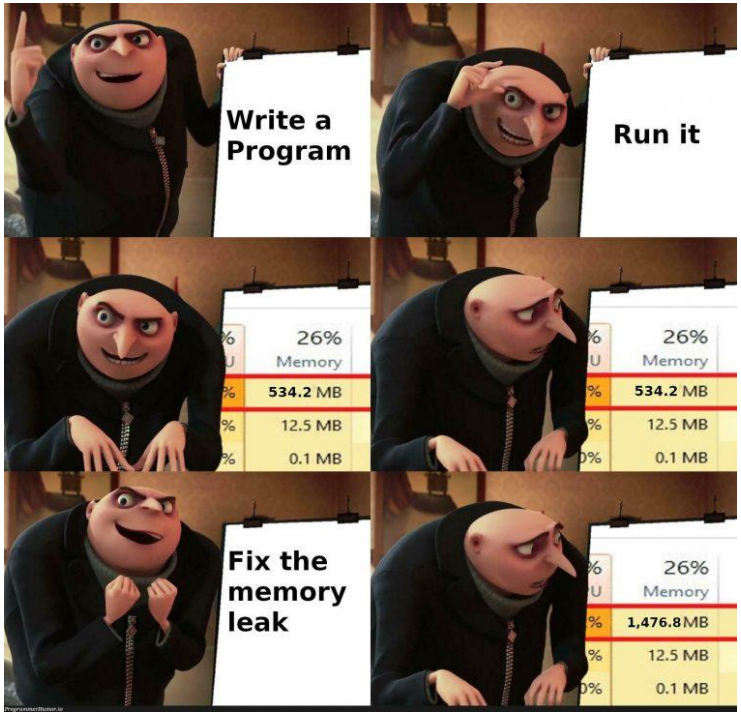
Stack vs. Heap

Feature	Stack	Heap
Allocation	Automatic (function calls)	Manual (`new` / `delete`)
Access Speed	Fast	Slower
Memory Size	Limited (small, fixed size)	Large (limited by system)
Lifetime	Automatic (end of scope)	Manual (until `delete` called)
Common Uses	Local variables, function calls	Dynamic data structures

- What about OpenFOAM? When do we need heap (dynamic) allocation?

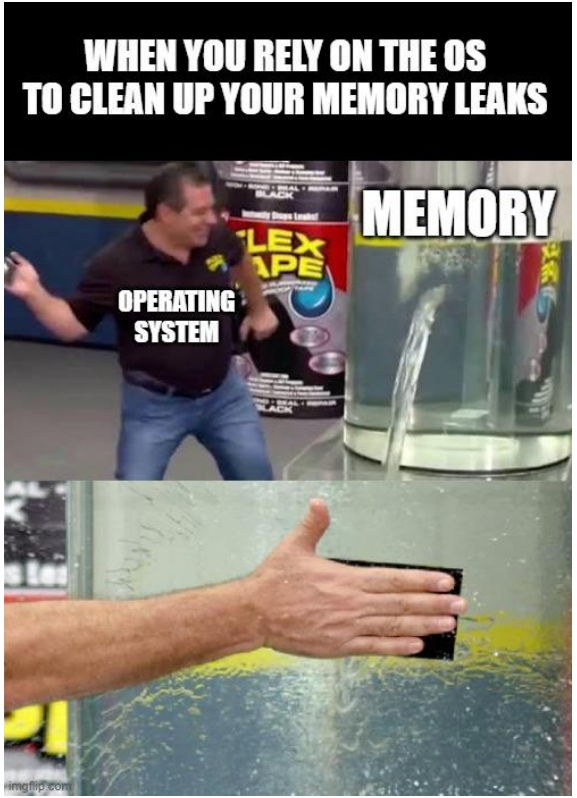
Deallocation

- Some programming languages (such as Python, Java) automatically deallocate (free) heap allocated memory
- It is usually called garbage collection (GC)
- GC adds some overhead
- C++ prioritizes performance and control over automatic management
- In C++, every single heap allocated memory should be freed manually
- In other words, every `new` operator in C++ should be followed by a corresponding `delete`



Why not rely on the OS?

- Most modern operating systems have garbage collection algorithms
- Why not just rely on the OS:
  - ✓Resource Waste
  - ✓Performance Issues
  - ✓Incomplete Cleanup
  - ✓Reliability
  - ✓Portability
  - ✓Hardware with no OS



Memory Management in Classes

Dynamically allocated member data

- Just like other examples, any dynamically allocated memory should be released.
- However, some other problems may also occur, if we are not cautious.
- Let’s have a look at Example 1.

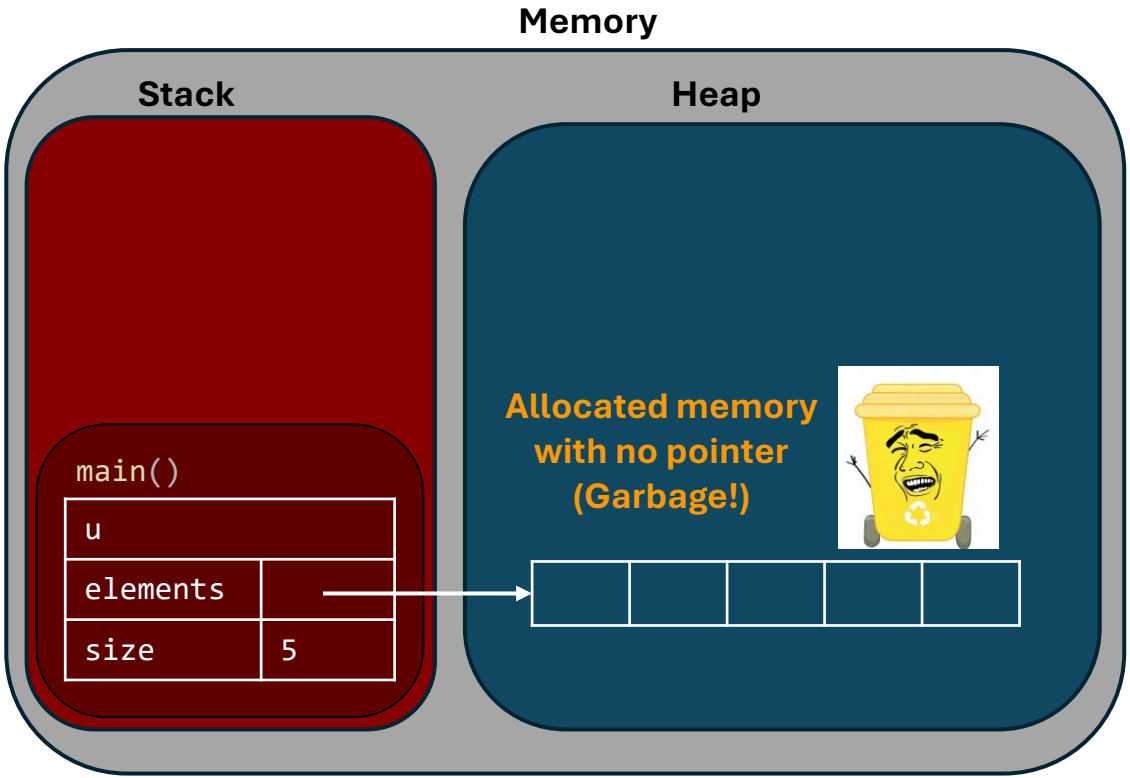
01_DynamicMemory.cpp

```
class myList
{
public:
    myList(int);
    int* elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = new int[size];
}

int main()
{
    cout << "Hello" << endl;
    myList u(5);
    cout << "Bye" << endl;
}
```

Memory leak

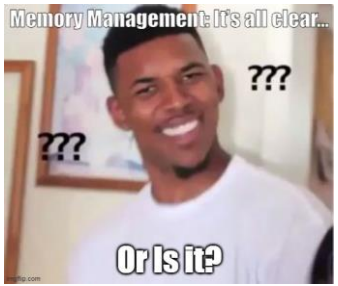
01_DynamicMemory.cpp

```
class myList
{
public:
    myList(int);
    int* elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = new int[size];
}

int main()
{
    cout << "Hello" << endl;
    myList u(5);
    cout << "Bye" << endl;
}
```



Destructor

- A special member function that is automatically called when an object goes out of scope or is explicitly deleted.
- Have a look at Example 2.
- Now the allocated memory is freed. The class does not have any memory management problem, and everything is fine.
- Or is it?



02_Destructor.cpp

```
class myList
{
public:
    myList(int);
    ~myList(); // destructor
    int *elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = new int[s];
}

myList::~myList()
{
    cout << " --> destructor called <-- \n";
    delete[] elements;
}
```

Shallow copy problem

- Problem with copying an object that contains pointers or dynamically allocated memory,
- Instead of copying the actual data, only the pointers (addresses) are copied.
- Let's have a look at Example 3.

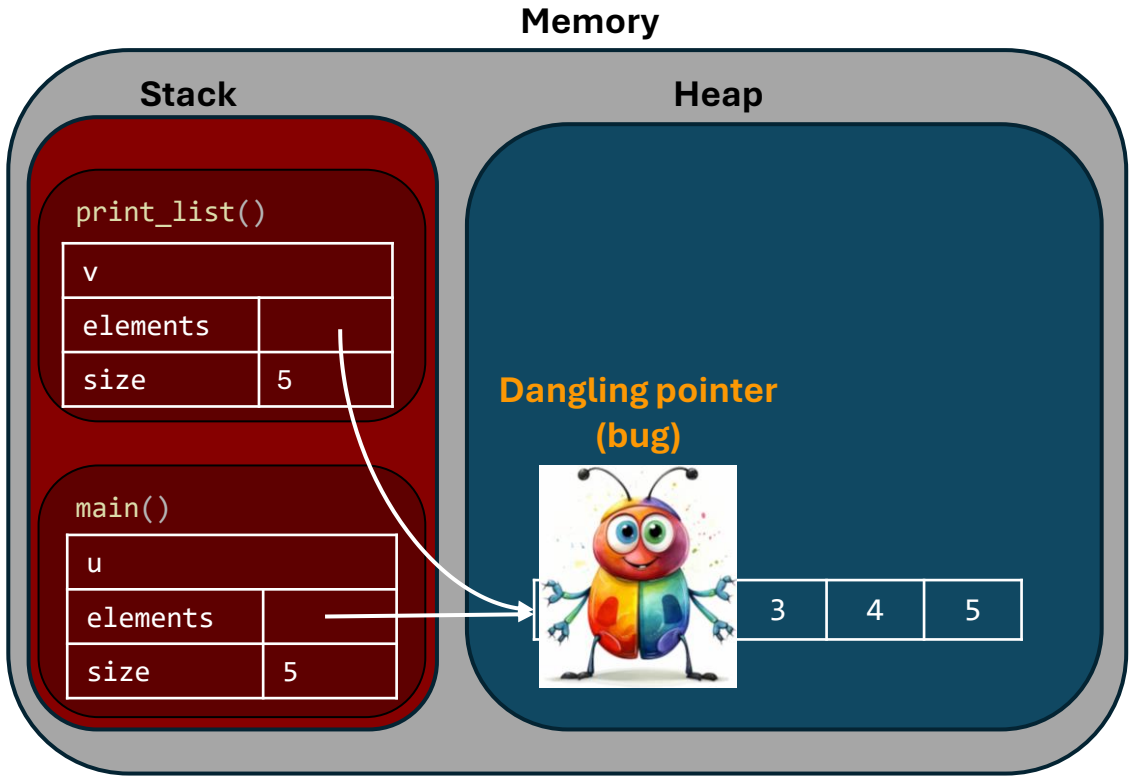
03_ShallowCopyProblem.cpp

```
(...)

void print_list(myList v)
{
    cout << "Print the list" << endl;
    for (int i = 0; i < v.get_size(); i++)
        cout << v.elements[i] << endl;
}

int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;
    print_list(u);
    cout << "Bye" << endl;
}
```

Shallow copy problem

03_ShallowCopyProblem.cpp

```
(...)

void print_list(myList v)
{
    cout << "Print the list" << endl;
    for (int i = 0; i < v.get_size(); i++)
        cout << v.elements[i] << endl;
}

int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;
    print_list(u);
    cout << "Bye" << endl;
}
```



Copy constructor

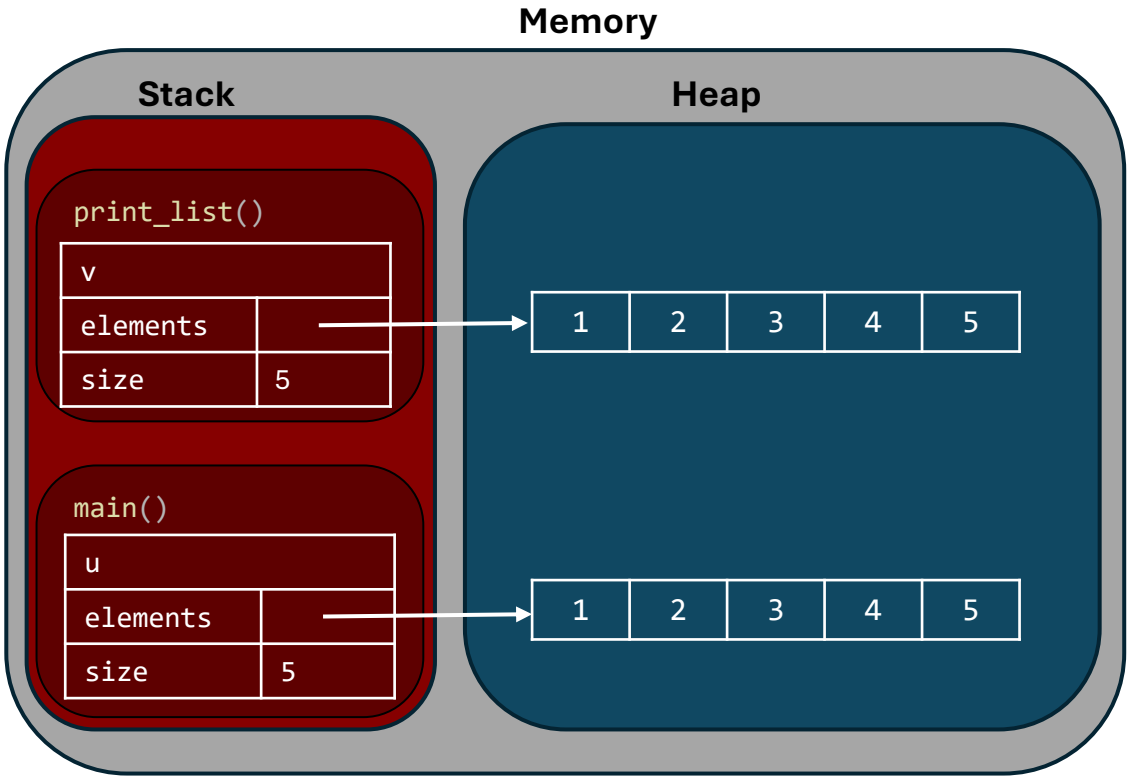
- To address the shallow-copy problem and perform a deep copy of the object, a non-default copy-constructor is required
- The copy constructor copies all the member data as well as the heap allocated memories
- Let's have a look at Example 4.

04_CopyConstructor.cpp

```
class myList
{
public:
    myList(int);
    myList(const myList&); // copy constructor
    ~myList(); // destructor
    int get_size();
    int* elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = new int[size];
}

myList::myList(const myList& u)
{
    cout << " --> copy constructor called <-- \n";
    size = u.size;
    elements = new int[u.size];
    for (int i = 0; i < u.size; i++)
        elements[i] = u.elements[i];
}
```

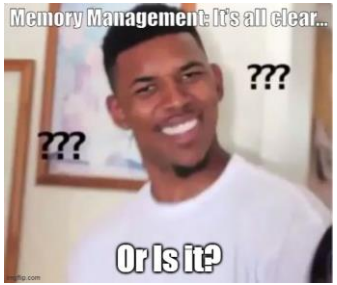
Deep copy

```
(...)

void print_list(myList v)
{
    cout << "Print the list" << endl;
    for (int i = 0; i < v.get_size(); i++)
        cout << v.elements[i] << endl;
}

int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;
    print_list(u);
    cout << "Bye" << endl;
}
```



Copy constructor

- Now with the copy constructor, the heap allocated memory is deeply copied, when passed by value.
- The class does not have any memory management problem, and everything is fine.
- Or is it?



04_CopyConstructor.cpp

```
class myList
{
public:
    myList(int);
    myList(const myList&); // copy constructor
    ~myList(); // destructor
    int get_size();
    int* elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = new int[size];
}

myList::myList(const myList& u)
{
    cout << " --> copy constructor called <-- \n";
    size = u.size;
    elements = new int[u.size];
    for (int i = 0; i < u.size; i++)
        elements[i] = u.elements[i];
}
```

Assignment problem

- A copy constructor is used to initialize a previously **uninitialized** object from some other object's data.

```
myList v = s;
```

- An assignment operator is used to replace the data of a **previously initialized** object with some other object's data.

```
myList x;
x = s;
```

05_AssignmentProblem.cpp

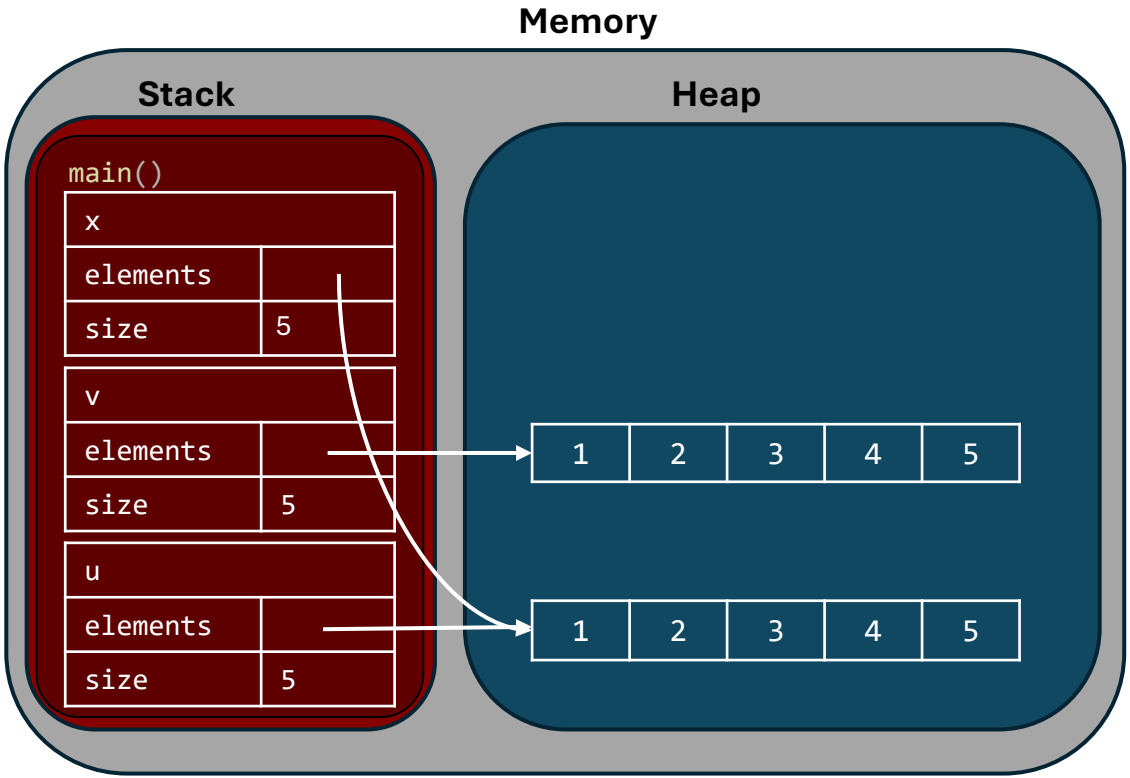
```
int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;

    myList v = u;    // copy constructor called
    myList w(u);     // copy constructor called

    myList x;
    x = u; // assignment
}
```

Assignment problem

05_AssignmentProblem.cpp

```
int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;

    myList v = u;    // copy constructor called
    myList w(u);     // copy constructor called

    myList x;
    x = u; // assignment
}
```

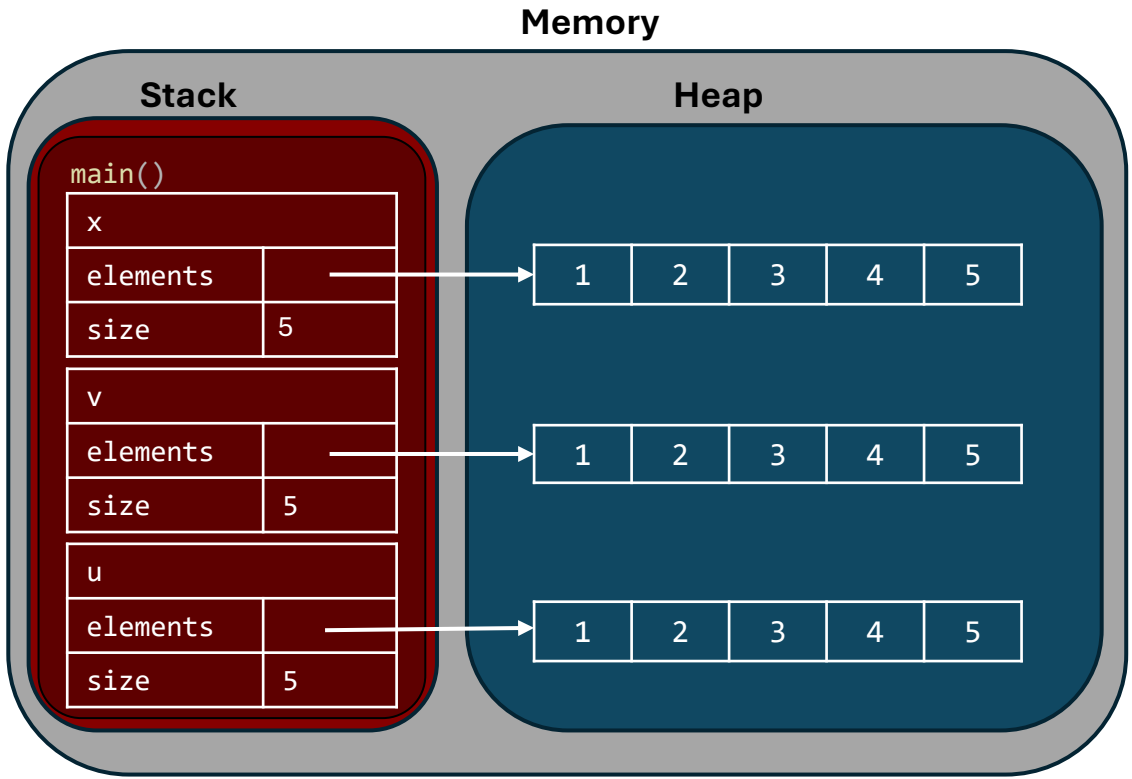
- The shallow copy problem again!



Overloading assignment operator

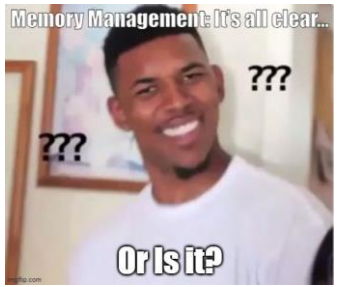
06_AssignmentOverload.cpp

```
myList& myList::operator=(const myList& u)
{
    size = u.size;
    delete[] elements;
    elements = new int[size];
    for (int i = 0; i < size; i++)
        elements[i] = u.elements[i];

    return *this;
}
(...)
int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;

    myList v = u;     // copy constructor called
    myList w(u);      // copy constructor called

    myList x;
    x = u; // assignment
}
```



Overloading assignment operator

- Now with the overloading assignment operator, the heap allocated memory is deeply copied by assignment.
- The class does not have any memory management problem, and everything is fine.
- Or is it?



- Don't worry, only one small issue remains!

06_AssignmentOverload.cpp

```
myList& myList::operator=(const myList& u)
{
    size = u.size;
    delete[] elements;
    elements = new int[size];
    for (int i = 0; i < size; i++)
        elements[i] = u.elements[i];

    return *this;
}
(...)
int main()
{
    cout << "Hello" << endl;
    myList u(5);
    for (int i = 0; i < u.get_size(); i++)
        u.elements[i] = i;

    myList v = u;    // copy constructor called
    myList w(u);     // copy constructor called

    myList x;
    x = u; // assignment
}
```

Self assignment problem

- Let's assume the user of your class writes the following funny code:

```
u = u;
```

- The `elements` of the u will be deleted and then we will end up with a pointer that points to a freed memory!

- The code should reliable enough to even handle funny usages

06_AssignmentOverload.cpp

```
myList& myList::operator=(const myList& u)
{
    size = u.size;
    delete[] elements;
    elements = new int[size];
    for (int i = 0; i < size; i++)
        elements[i] = u.elements[i];

    return *this;
}
```

Addressing the self assignment

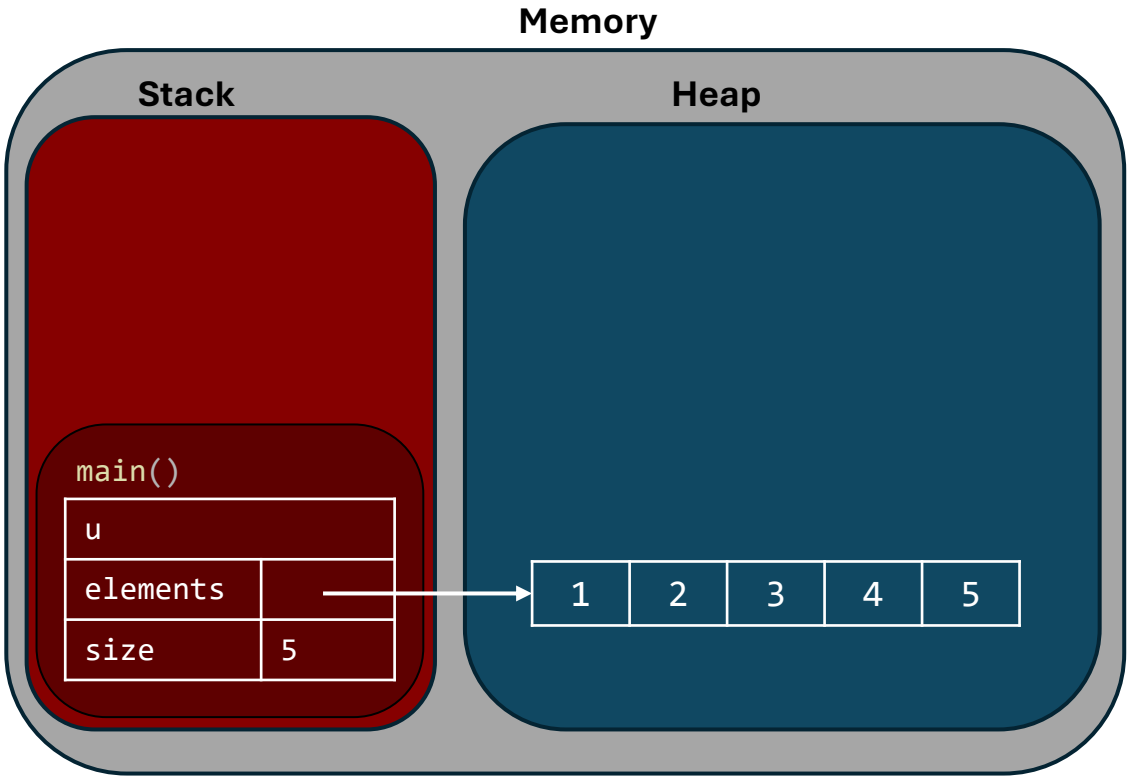
- To address this issue, we return in case of self-assignment
- Now the class really does not have any memory management problem, and everything is fine.

07_AssignmentOverload.cpp

```
myList& myList::operator=(const myList& u)
{
    if (this == &s)
        return *this;

    size = u.size;
    delete[] elements;
    elements = new int[size];
    for (int i = 0; i < size; i++)
        elements[i] = u.elements[i];

    return *this;
}
```

General rules

- For a class that dynamically allocates memory on the heap, one should explicitly define:
  - ✓A destructor
  - ✓A copy constructor
  - ✓A copy assignment operator
- Let's look at an OpenFOAM example: `fvMatrix` class in OpenFOAM-v2112



How OpenFOAM handles raw pointers

- Let's look at an OpenFOAM example: `fvMatrix` class in OpenFOAM-v2112

- Specifically, loo at the member data raw pointer `faceFluxCorrectionPtr_`

`fvMatrix.C` in OpenFOAMv2112

```
template<class Type>
Foam::fvMatrix<Type>::fvMatrix
(
    const GeometricField<Type, fvPatchField, volMesh>& psi,
    const dimensionSet& ds
)
:
    lduMatrix(psi.mesh()),
    psi_(psi),
    useImplicit_(false),
    lduAssemblyName_(),
    nMatrix_(0),
    dimensions_(ds),
    source_(psi.size(), Zero),
    internalCoeffs_(psi.mesh().boundary().size()),
    boundaryCoeffs_(psi.mesh().boundary().size()),
    faceFluxCorrectionPtr_(nullptr)
{
```

Smart Pointers

Smart Pointers

- In big project, it hard to keep track of all the heap allocated memories and `delete` them.
- Smart pointers automates this (that's all what they are!).
- Some people have this style of coding that they never ever use `new` and `delete`. Some others only use `new` and `delete`. OpenFOAM uses both!
- It recommended to use them, unless you can't for some reason.
- According to Microsoft: "smart pointers are used to help ensure that programs are free of memory and resource leaks and are exception-safe."
- Types of smart pointers:
  - ✓`std::unique_ptr<>`
  - ✓`std::shared_ptr<>`
  - ✓`std::weak_ptr<>`

std::unique_ptr<>

- Unique ownership of memory
- Scoped pointer: when the pointer goes out of scope the memory is released
- It has negligible overhead
- Unique: one cannot copy them (share the ownership of memory)
- Let's have a look at Example 8.
- Let's look at an OpenFOAM example: `fvMatrix` class in OpenFOAM-v2406

08_UniquePointer.cpp

```
class myList
{
public:
    myList(int);
    unique_ptr<int[]> elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = unique_ptr<int[]>(new int[size]); // with new
    // elements = make_unique<int[]>(size);      // with make_unique
}
```

std::shared_ptr<>

- Sharing ownership of a memory
- Uses reference counting to keep track of the number of pointer that point to the same memory
- Reference counting adds some overhead
- When all pointer die (ref. count = 0), the memory is released.
- Let's have a look at Example 9.
- `weak_ptr<>` is similar to `shared_ptr<>` without increasing the ref. count.

09_SharedPointer.cpp

```
class myList
{
public:
    myList(int);
    shared_ptr<int[]> elements;

private:
    int size;
};

myList::myList(int s)
{
    cout << " --> constructor called <-- \n";
    size = s;
    elements = shared_ptr<int[]>(new int[size]);
}
```