Demo
Constructor & Destructor
Dynamic Memory

Scope (dynamic array)

- Define a class called Cell with 2 member variables, m_sz and m_ptr, where m_sz is the size of the array dynamically allocated during class construction, m_ptr is a pointer to the dynamic array.

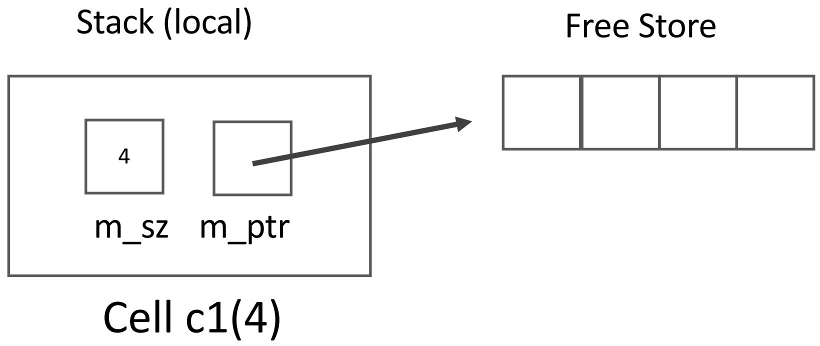
```
class Cell {

    public:

    protected:

    private:
        // size of the dynamic array
        int m_sz {0};

        // pointer to the dynamic array
        int *m_ptr {nullptr};
};
```



What is class constructor

- Class constructor is simply the creation of the class object(s), including initializing ALL the class members (variables) prior to its use.
- How class object constructed:
  - By Declaration: Cell c1(4)

    Normal
  - Based on EXISTING class instance, say c1:

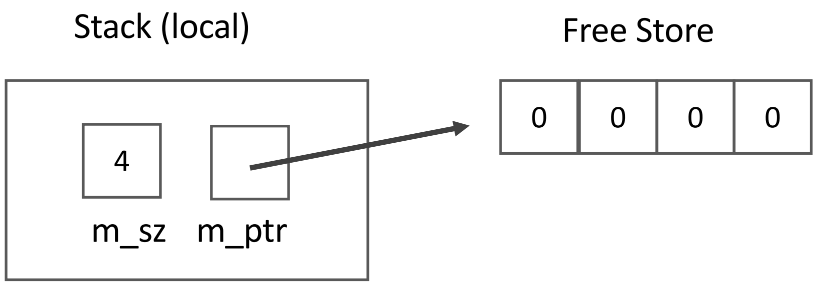
    Copy,
    Copy Assignment
    - c2 = c1, Cell c2 {c1}, Cell c2(c1)
  - Based on Temporary class instance, by function return or temporary instance:

    Move,
    Move Assignment
    - c2 = createNewCellInstance()
    - vector.push_end(Cell {4});

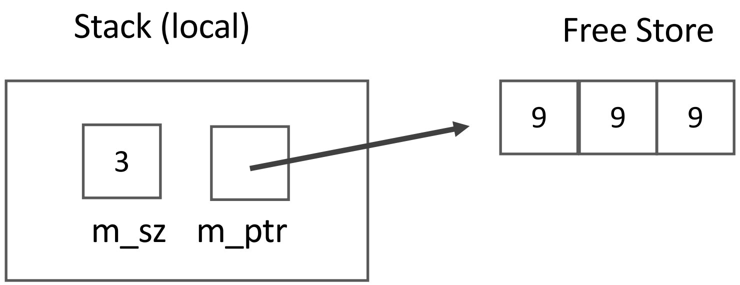
Normal Constructor

- Define how the class object is initialized by parameters.
- Define multiple constructors with different parameter signatures.
- For examples:
  - Cell(int sz) – a constructor that takes the size of the dynamic array
  - Cell(int sz, int value) - another constructor that takes the size of the array and the value used to initialize the array.

Copy Constructor – given existing class instances (c1 and c2)



Cell c1(4)



Cell c2(3,9)

c2 = c1

Steps:

1. Free memory previously allocated by c2
2. Allocate NEW memory according to c1’s
3. Copy content of c1 to c2

Cell c3 = c1

Steps:

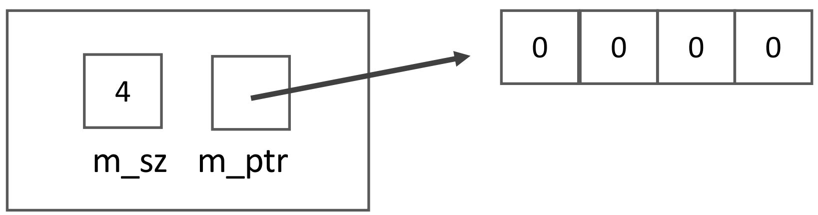
1. ~~Free memory previously allocated by c2~~
2. Allocate NEW memory according to c1’s
3. Copy content of c1 to c3

Move Constructor – temp. class instances

Cell foo() {

Stack (local)

Free Store



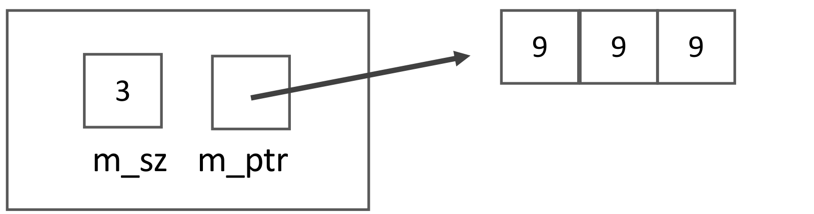
Cell c1(4)

return c1(4);

}

Stack (local)

Free Store



Cell c2(3,9)

c2 = foo()

Steps (what you would think):

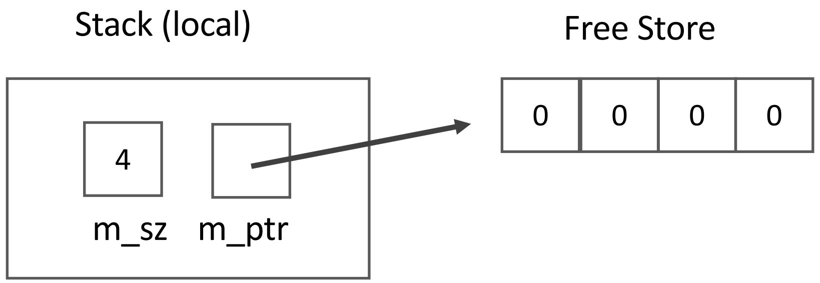
1. Free memory previously allocated by c2
2. Allocate NEW memory according to temp. object
3. Copy content of temp. obj to c2

OR

1. Swap member variables between c2 and temp. obj

Move Constructor – temp. class instances

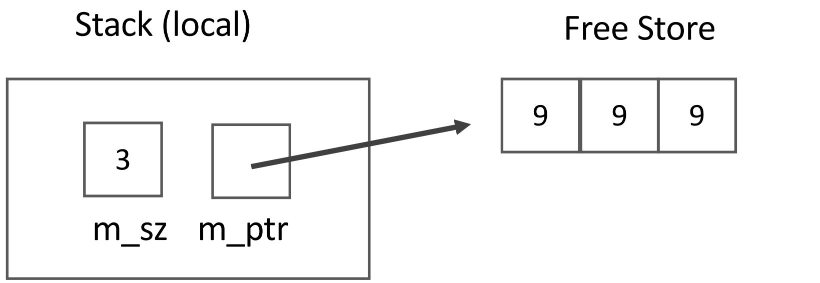
Cell foo() {



Cell c1(4)

return c1(4);

}



Cell c2(3,9)

c2 = foo()

~~Steps (what you would think):~~

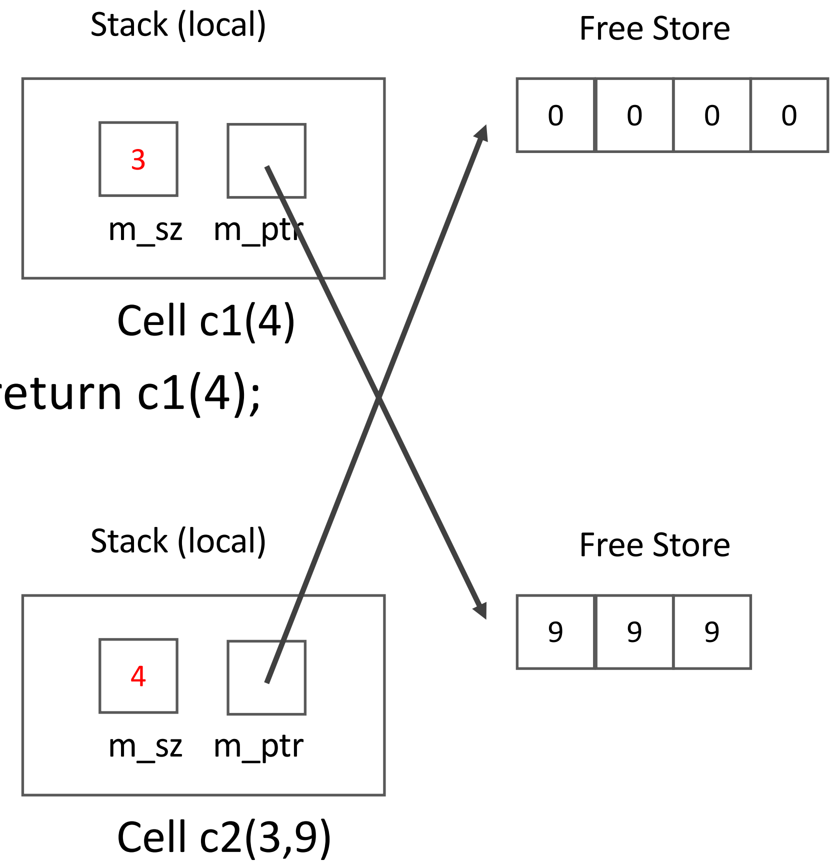
1. ~~Free memory previously allocated by c2~~
2. ~~Allocate NEW memory according to temp. object~~
3. ~~Copy content of temp. obj to c2~~

OR

1. Swap member variables between c2 and temp. obj

Move Constructor – temp. class instances

```
Cell foo() {
    return c1(4);
}
```



c2 = foo()

~~Steps (what you would think):~~

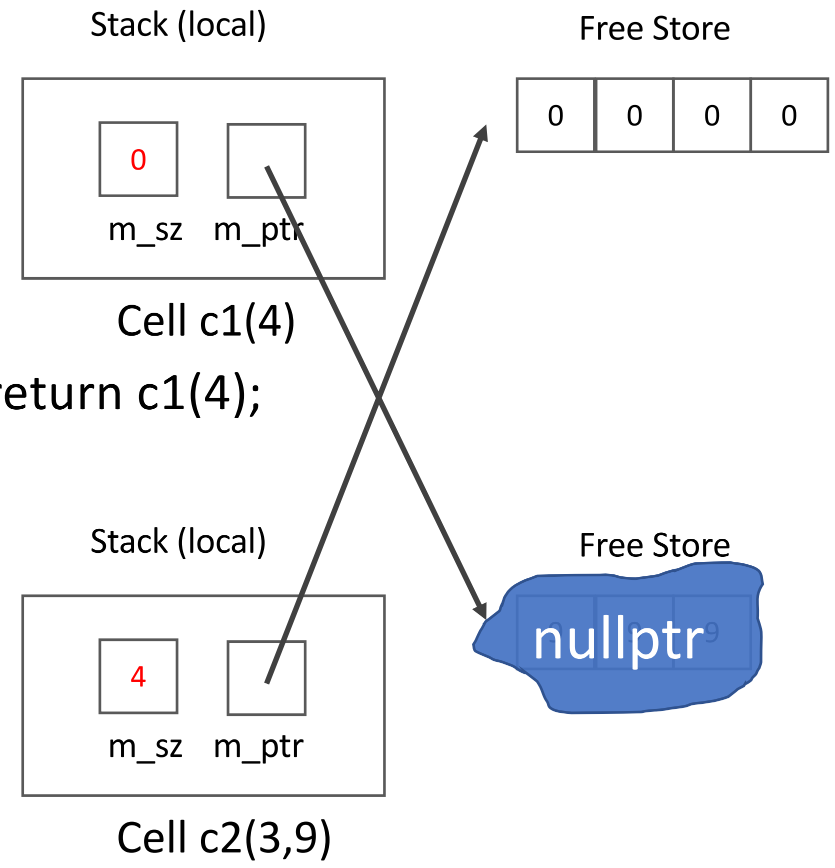
1. ~~Free memory previously allocated by c2~~
2. ~~Allocate NEW memory according to temp. object~~
3. ~~Copy content of temp. obj to c2~~

OR

1. Swap member variables between c2 and temp. obj

Move Constructor – temp. class instances

```
Cell foo() {
    return c1(4);
}
```



c2 = foo()

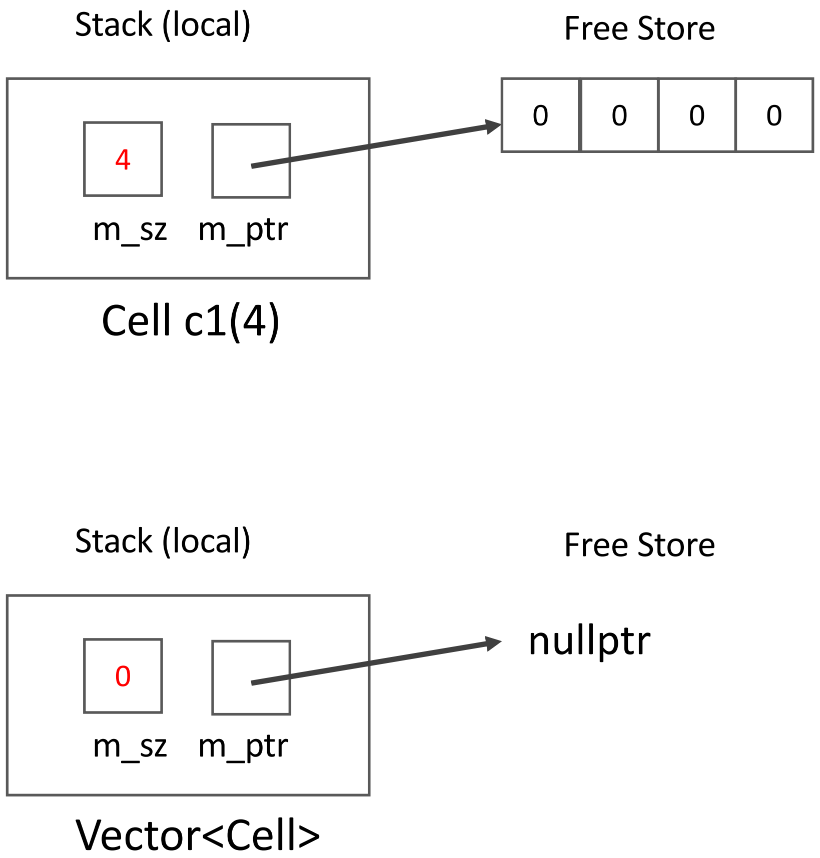
~~Steps (what you would think):~~

1. ~~Free memory previously allocated by c2~~
2. ~~Allocate NEW memory according to temp. object~~
3. ~~Copy content of temp. obj to c2~~

OR

1. Swap member variables between c2 and temp. obj
   - Temp. obj will be destructed automatically (destructor)

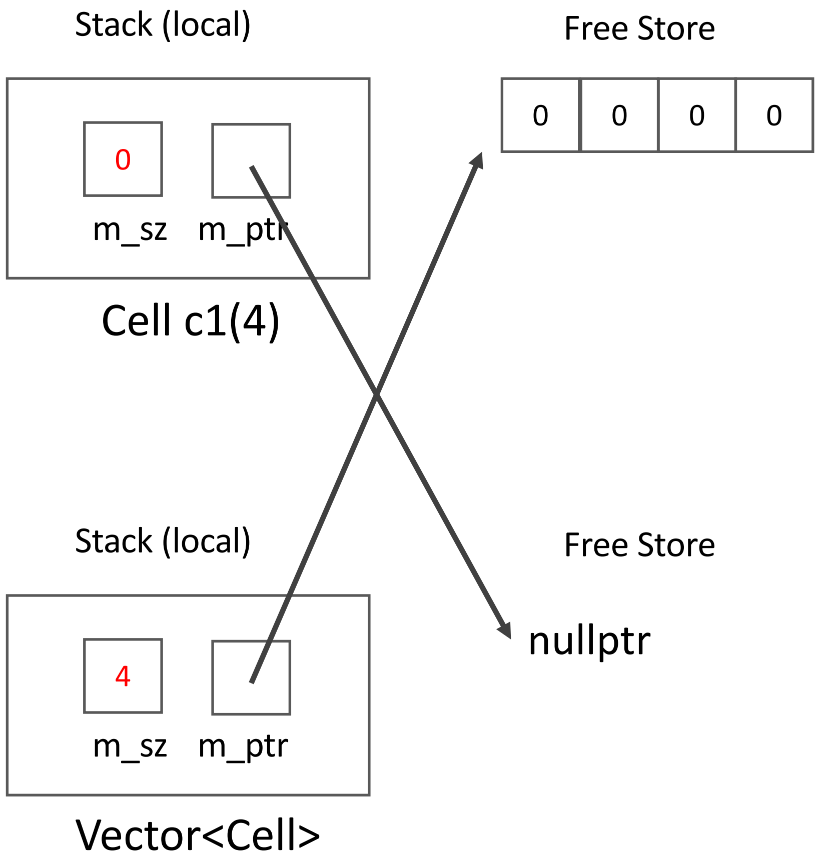
Move Constructor – temp. class instances



Vector.push_back(Cell {4})

1. Swap member variables between vector element and temp. obj
    - Temp. obj will be destructed automatically (destructor)

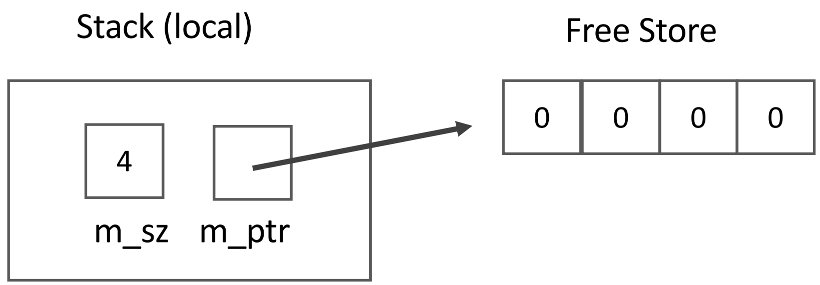
Move Constructor – temp. class instances



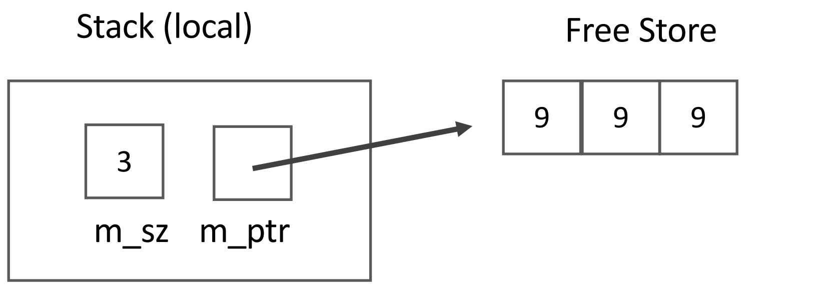
Vector.push_back(Cell {4})

1. Swap member variables between vector element and temp. obj
   - Temp. obj will be destructed automatically (destructor)

Copy Constructor – given existing class instances (c1 and c2) - Refactor



Cell c1(4)



Cell c2(3,9)

c2 = c1

Steps:

1. ~~Free memory previously allocated by c2~~
2. ~~Allocate NEW memory according to c1's~~
3. ~~Copy content of c1 to c2~~
4. Create temp. obj out of c1
5. Swap c2 and temp.obj

Cell c3 = c1

Steps:

1. ~~Free memory previously allocated by c2~~
2. Allocate NEW memory according to c1's
3. Copy content of c1 to c3

Framework

```cpp
void create(int sz) {
    m_ptr = new int[sz];
    for (int i {0}; i < m_sz; i++)
        this->m_ptr[i] = m_sz;  // for tracking
}
```

```cpp
void free() {
    if (m_ptr == nullptr) return;
    cout << "destroy " << m_sz << endl;
    m_sz = 0;
    delete [] m_ptr;
    m_ptr = nullptr;
}
```

```cpp
void copy(const Cell & src) {
    for (int i {0}; i < m_sz; i++)
        this->m_ptr[i] = src.m_ptr[i];
}
```

```cpp
void swapFields(Cell & from) noexcept {

    std::swap(m_sz, from.m_sz);
    std::swap(m_ptr, from.m_ptr);
}
```

Refactor – Normal Constructor

```
// normal
Cell(int sz): m_sz(sz) {
    cout << "normal " << sz << endl;
    create(sz);
}
```

Refactor – Copy Constructor

```
// copy constructor (delegate to normal constructor)
Cell(const Cell & src): Cell(src.m_sz) {
    cout << "copy " << src.m_sz << endl;
    copy(src);
}

// copy assignment (not need to skip if this == &rhs)
Cell & operator=(const Cell & rhs) {
    cout << "copy= " << rhs.m_sz << endl;
    Cell tmp {rhs}; // call copy constructor
    swapFields(tmp);
    return *this;
}
```

Refactor – Move Constructor

```
// move constructor
Cell(Cell && src) noexcept {
    cout << "move " << src.m_sz << endl;
    swapFields(src);
}

// move assignment
Cell & operator=(Cell && rhs) noexcept {
    cout << "move= " << rhs.m_sz << endl;
    swapFields(rhs);
    return *this;
}
```

Demo

```cpp
Cell createCell() {
    return Cell {11};
}

void test(vector<Cell> vec, int capacity) {

    cout << "Reserve Capacity=" << capacity << endl;
    vec.reserve(capacity);

    for (int i {100}; i < 103; ++i) {
        cout << "Loop=" << i << endl;
        vec.push_back( Cell {i} );
        cout << endl;
    }

    Cell c1 {10};
    cout << c1 << endl;

    c1 = createCell();
    cout << c1 << endl;

    Cell c2 {12};
    cout << c2 << endl;

    c2 = c1;
    cout << c2 << endl;

    Cell c3 = c2;
    cout << c3 << endl;

}
```

```
Reserve Capacity=10
Loop=100
normal 100
move 100

Loop=101
normal 101
move 101

Loop=102
normal 102
move 102

normal 10
10 10 10 10 10 10 10 10 10 10

normal 11
move= 11
destroy 10
11 11 11 11 11 11 11 11 11 11 11

normal 12
12 12 12 12 12 12 12 12 12 12 12 12

copy= 11
normal 11
copy 11
destroy 12
11 11 11 11 11 11 11 11 11 11 11

normal 11
copy 11
11 11 11 11 11 11 11 11 11 11 11
```

```cpp
Cell createCell() {
    return Cell {11};
}

void test(vector<Cell> vec, int capacity) {

    cout << "Reserve Capacity=" << capacity << endl;
    vec.reserve(capacity);

    for (int i {100}; i < 103; ++i) {
        cout << "Loop=" << i << endl;
        vec.push_back( Cell {i} );
        cout << endl;
    }

    Cell c1 {10};
    cout << c1 << endl;

    c1 = createCell();
    cout << c1 << endl;

    Cell c2 {12};
    cout << c2 << endl;

    c2 = c1;
    cout << c2 << endl;

    Cell c3 = c2;
    cout << c3 << endl;

}
```

```
Reserve Capacity=0
Loop=100
normal 100
move 100

Loop=101
normal 101
move 101
move 100

Loop=102
normal 102
move 102
move 101
move 100
```