Data Structures Lab

(BBM203 Software Practicum I)

Week 4: Java to C++ Transition Tutorial

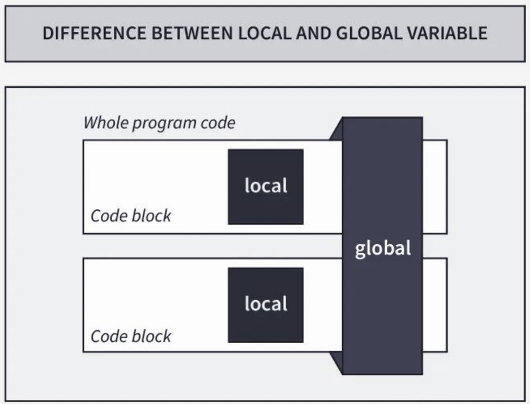
https://web.cs.hacettepe.edu.tr/~bbm201/
https://piazza.com/hacettepe.edu.tr/fall2023/bbm203

Topics

➔ Memory Management

Memory Management

Local vs. Global Storage



Local Storage:

- Variables declared within a function or a block.
- ***Scope***: Limited to the block or function where they are defined.
- ***Lifetime***: Created when the block is entered, destroyed when the block exits.

Advantages:

- **Encapsulation**: Limited visibility, reducing potential conflicts.
- **Efficient memory usage**: Memory is allocated and deallocated dynamically.

Disadvantages:

- **Limited scope**: can potentially lead to code duplication or difficulty in sharing data across different parts of the program.
- **No persistence**: Data is not persistent between function calls, requiring reinitialization each time.
- **Memory overhead**: Memory allocation and deallocation for local variables can impact performance for frequent function calls. Dynamic memory management may add a slight overhead compared to static allocation used for global variables.

Local vs. Global Storage

Local Storage:

- Variables declared within a block.

```
{
    int my_integer; // memory for an integer allocated
    // ... my_integer is used here ...

    Foo my_foo; // memory for instance of class Foo allocated
    // ... my_foo is used here ...
}
```

- The curly braces **{** and **}** mark the beginning and the end of a ***block***.
- When program flow enters the block, memory needed is allocated.
- When the end of the block is reached, this memory is freed up and those variables cease to exist.
- Trying to use the variables after the block is closed will yield ***compile*** errors, just as in Java.

Local vs. Global Storage

Global Storage:

- Variables declared outside of any function, accessible throughout the program.
- ***Scope***: Accessible throughout the program.
- ***Lifetime***: Created when the program starts, destroyed when the program ends.

Advantages:

- **Widespread accessibility**: Useful for sharing data across multiple functions.
- **Persistent data**: Retains its value between function calls.

Disadvantages:

- **Potential for Conflicts**: Increasing the risk of unintentional modifications and naming conflicts, leading to potential bugs and maintenance challenges.
- **Security Risks**: Can be accessed and modified from any part of the program, making it difficult to control access.
- **Difficulty in Debugging**: Identifying the source of an issue related to a global variable can be challenging.
- **Encapsulation Challenges**: may violate the principle of encapsulation by allowing unrestricted access, making it harder to enforce data integrity and controlled access.

Local vs. Global Storage

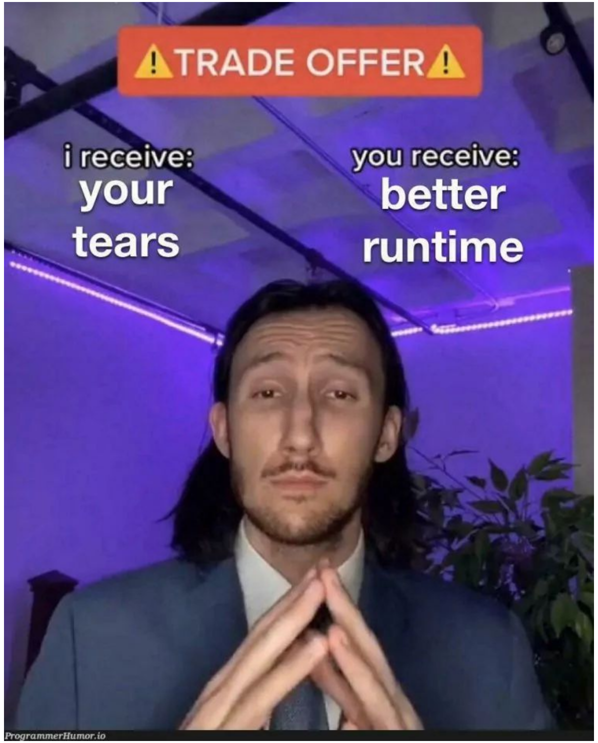

```cpp
#include <iostream>

int globalVariable = 10; // Global variable

void exampleFunction() {
    int localVariable = 20; // Local variable
    std::cout << "Local variable: " << localVariable << std::endl;
    std::cout << "Global variable: " << globalVariable << std::endl;
}

int main() {
    // Access global variable
    std::cout << "Global variable from main: " << globalVariable << std::endl;

    // Attempting to access local variable here would result in a compilation error
    // std::cout << "Trying to access local variable: " << localVariable << std::endl;

    exampleFunction();  // Call the function

    return 0;
}
```

https://pythontutor.com/visualize.html

Local vs. Global Storage

Comparison:

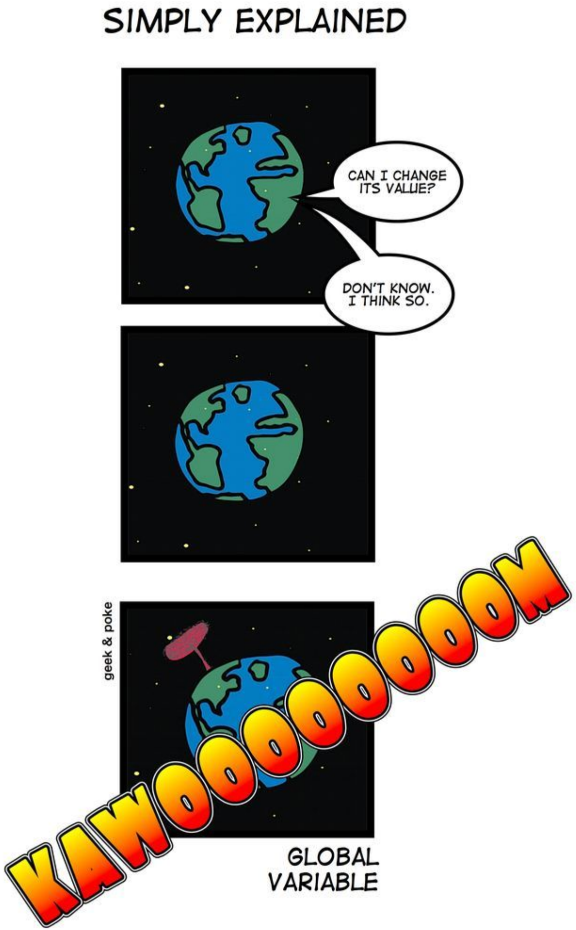
- Local variables are confined to a specific block, promoting encapsulation.
- Global variables can lead to naming conflicts and may be accessed unintentionally.
- Local variables have a controlled scope and lifetime, aiding memory management.
- Global variables can cause potential issues like unintended modifications.



Local vs. Global Storage

Best Practices:

- **Prefer local variables**: Encourage encapsulation and reduce unintended side effects.
- **Minimize global variables**: Limit their use to cases where necessary and clearly document their usage.



Allocating memory with new

In C++, we can request a block of memory in global storage by using **`new`** keyword, and we return the memory by using **`delete`**.

The syntax for the new operator is as follows:

```
new ClassName(params);
```

- On success, a chunk of heap memory that is the size of the object is allocated and a pointer to that memory is returned.
- If the memory can not be allocated, an exception is thrown (**`std::bad_alloc`**).

Allocating memory with new

The following C++ code shows how you can allocate memory and use it later:

[Bar.H]

```
class Bar {
public:
    Bar() { m_a = 0; }
    Bar(int a) { m_a = a; }
    void myFunction(); // this method would be defined elsewhere (e.g. in Bar.C)
protected:
    int m_a;
};
```

[main.C]

```
#include "Bar.H"
int main(int argc, char *argv[]){
    // declare a pointer to Bar; no memory for a Bar instance is allocated now
    // p currently points to garbage
    Bar * p;
    {
        // create a new instance of the class Bar (*p)
        // store pointer to this instance in p
        p = new Bar();
    }

    // since Bar is in global storage, we can still call methods on it
    // this method call will be successful
    p->myFunction();
}
```

Notice that you can still use the object generated by the new statement even if you are outside the block.

Deallocating memory with delete

- In Java, a garbage collector frees the memory automatically when no existing object references it.
- In C++, you have to be much more responsible than that. **Whatever memory you allocate in heap storage, you must explicitly free**, or your program will swell in size and contain **memory leaks**.
- To avoid leaks, you need to keep track of all the dynamic memory you have allocated and free it when you no longer need it:


```
delete p; // memory pointed to by p is deallocated
```

Deallocating memory with delete

- Only objects created using **`new`** should be freed with **`delete`**!
- Instances created in local storage are automatically recycled and should not be deleted explicitly.
- For example, the following code will make your program crash:

```
Bar bar; // bar not created with new
// ... use the instance of Bar ...
delete &bar; // OH NO! bar is in local storage, chaos ensues!
```

- Arrays are deleted differently than single instances.

```
delete [] array_name;
```

My C++ program tries freeing statically allocated local variable memory

OS:

Managing Memory: Classes

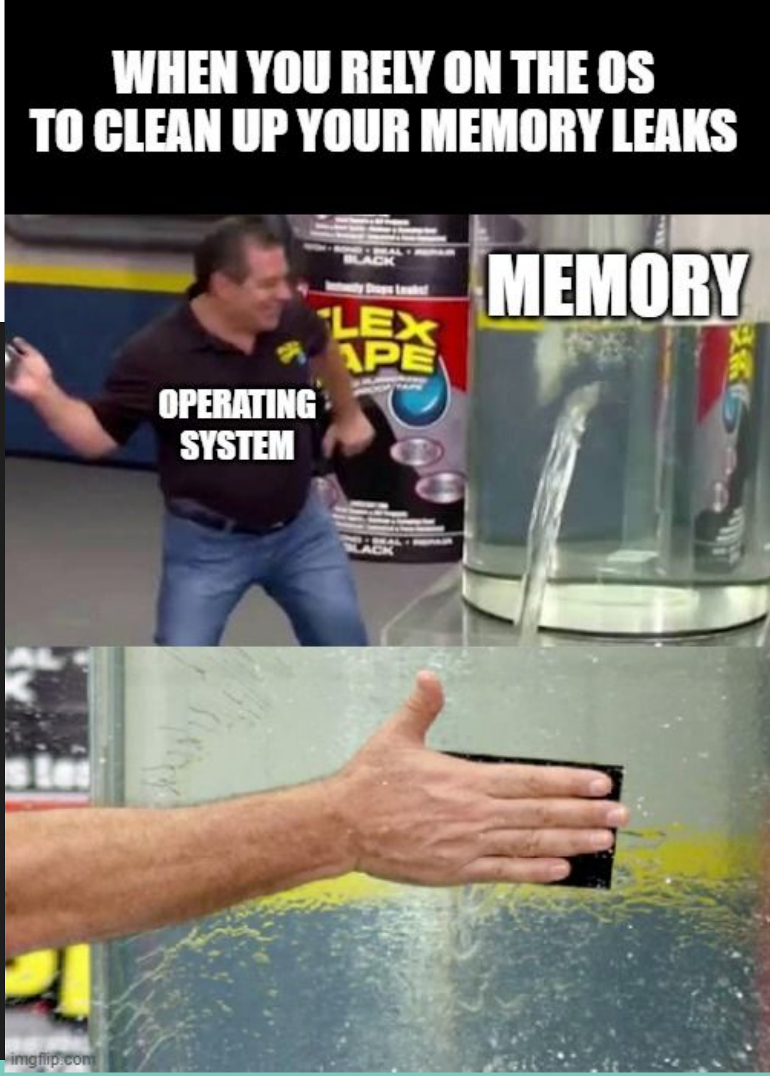
Good vs. Bad Memory Management

- We often write code that has leaks everywhere. However, there is an easy and effective way of avoiding them: **good programming style**.
- Since you can free memory at any time your program is running, the question is when to do it?
- The following is a good rule:
  - **Memory allocated in a constructor should be deallocated in a destructor**, and
  - **Memory allocated in a function should be deallocated before it exits**.

Managing Memory: Classes

Bad Memory Management Example:

```cpp
#include "Bar.H"

class Foo {
private:
    Bar* m_barPtr;
public:
    Foo() {}
    ~Foo() {}
    void funcA() {
        m_barPtr = new Bar; // Some memory allocated
    }

    void funcB() {
        // use object *m_barPtr
    }

    void funcC() {
        // ...
        delete m_barPtr; // This memory is freed up here
    }
 };
```



Managing Memory: Classes

Bad Memory Management Example - some code that uses the **Foo** class:

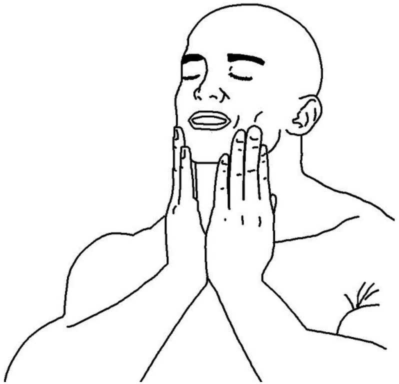
```
{
    Foo myFoo; // create local instance of
               // Foo
    myFoo.funcA(); // memory for *m_barPtr is
                   // allocated

    // ...
    myFoo.funcB();
    // ...
    myFoo.funcB();
    // ...

    myFoo.funcC(); // memory for *m_barPtr is
                   // deallocated
 }
```

Code that does not leak any memory

```
{
    Foo myFoo;
    //...
    myFoo.funcB(); // oops, bus error in funcB()

    myFoo.funcA(); // memory for *m_barPtr is allocated

    myFoo.funcA(); // memory leak, you lose track of the
                   // memory previously pointed to by
                   // m_barPtr when new instance stored
    //...
    myFoo.funcB();

 }   // memory leak! memory pointed to by m_barPtr in myFoo

     // is never deallocated
```

Code that uses the Foo class improperly

Managing Memory: Classes



Good Memory Management Example:

```
#include "Bar.H"

class Foo {
private:
    Bar* m_barPtr;
public:
    Foo() { m_barPtr = new Bar; }
    ~Foo() { delete m_barPtr; }
    void funcA() {}

    void funcB() {
        // use object *m_barPtr
    }

    void funcC() {
        // ...
    }
};
```

- Memory is always allocated in the constructor at the time a **Foo** object is allocated.
- The memory is automatically deleted when **myFoo** is deleted or goes out of scope.
- Using the constructor above, it is impossible not to allocate the memory before we call **funcB**, nor is it possible to forget to delete the memory, since the destructor is automatically called.

Pointers, references, and instances

- **Dynamically allocated memory** using the **new** keyword should be deallocated manually using the **delete** keyword.
- **Local variables** will be automatically deallocated once they are **out of scope**, regardless of whether your program has a reference to them anywhere or not.

Reminders!


```
Counter * counter_dynamic = new Counter(6);
int number;
std::cout << "Enter the array size: ";
std::cin >> number;
Counter * counter_array_dynamic = new
Counter[number];

// After you are DONE with those variables.
delete counter_dynamic;
delete [] counter_array_dynamic;
```

Pointers, references, and instances

- When you have **dynamically allocated variables** as **member variables** inside a **class**, you should deallocate the memory for those variables in the **class destructor**.

Reminders!

- This way, dynamically allocated memory is **managed correctly** and deallocated when the object goes out of scope or is explicitly destroyed.


```cpp
class MyClass {
public:
    int* dynamic_array;

    MyClass (int size) {
        // Dynamically allocate memory
        dynamic_array= new int[size];
    }
    ~MyClass () {
        // Deallocate memory in the destructor
        delete[] dynamic_array;
    }
};
```

Parameters

- In C++, parameters can be passed to functions in several ways: by **value**, by **reference**, or by **pointer**.
- The default behavior in C++ is to **pass by value (pass by copy)**. This will cause a copy of the object to be passed to the function.
- The counter **will not increase**!

```cpp
void increment_by_copy(Counter counter){
    counter++;
}

int main() {
    Counter counter_1(0);
    cout << "initial: " << counter_1 << endl;
    increment_by_copy(counter_1);
    cout << "after increment: " << counter_1 << endl;
    return 0;
}
```

```
> initial: 0
> after increment: 0
```

Parameters

- To be able to alter the value of a local variable **in a different function**, we should either use **pass by pointer** or **pass by reference**.

- The value of the counter **will increase**!

```
void increment_by_pointer(Counter * counter){
    (*counter)++;
}

int main() {
    Counter counter_2(11);
    cout << "initial: " << counter_2 << endl;
    increment_by_pointer(&counter_2);
    cout << "after increment: " << counter_2 << endl;
    return 0;
}
```

```
void increment_by_reference(Counter & counter){
    counter++;
}

int main() {
    Counter counter_3(11);
    cout << "initial: " << counter_3 << endl;
    increment_by_reference(counter_3);
    cout << "after increment: " << counter_3 << endl;
    return 0;
}
```

```
> initial: 11
> after increment: 12
```

Returning Pointers

- While returning a pointer from a function, you should be careful **not to return the address of a local variable**.

- Returning a pointer to a local variable can lead to **undefined behavior**, as the local variable's memory may be deallocated once the function exits, **making the pointer invalid**.

This is a common source of bugs and segfaults!

```
Counter* make_counter() {
    Counter counter(0);
    return &counter;
}
int main() {
    Counter * my_counter = make_counter();
    (*my_counter)++;
    cout << *my_counter << endl;
    return 0;
}
```

```
counter_ptr.cpp:10:12: warning: address of
local variable ‘counter’ returned
[-Wreturn-local-addr]
   10 |     return &counter;
      |            ^~~~~~~~
counter_ptr.cpp:9:13: note: declared here
    9 |     Counter counter(0);
      |             ^~~~~~~
```

g++ warning

```
==267== Invalid read of size 4
==267==    at 0x109367: Counter::operator++(int)
(Counter.cpp:20)
==267==    by 0x10924B: main (counter_ptr.cpp:15)
==267==  Address 0x0 is not stack'd, malloc'd or
(recently) free'd
```

Runtime error caught by valgrind

Returning References

- Returning a reference to a local variable from a function is equally problematic.
- Again, the memory for the reference will be deallocated once the function exits, leaving you with an **invalid reference**.

This is a common source of bugs and segfaults!

```
Counter& make_counter(){
    Counter counter(0);
    return counter;
}
int main() {
    Counter my_counter = make_counter();
    my_counter++;
    cout << my_counter << endl;
    return 0;
}
```

```
counter_ptr.cpp:10:12: warning: address of
local variable ‘counter’ returned
[-Wreturn-local-addr]
   10 |     return counter;
      |            ^~~~~~~~
counter_ptr.cpp:9:13: note: declared here
    9 |     Counter counter(0);
      |             ^~~~~~~
```

g++ warning

```
==276== Invalid read of size 4
==276==    at 0x10934E: Counter::Counter(Counter&)
(Counter.cpp:16)
==276==    by 0x109241: main (counter_ptr.cpp:14)
==276==  Address 0x0 is not stack'd, malloc'd or
(recently) free'd
```

Runtime error caught by valgrind

Pointer to Heap

- Returning a pointer to a **dynamically allocated variable** from a function is **perfectly fine!**
- Just don't forget to **deallocate** the variable using **the delete keyword**.
- Here is the **valgrind output** we all want to see:

```
Counter* make_counter(){
    Counter * counter = new Counter;
    return counter;
}

int main() {
    Counter * my_counter = make_counter();
    (*my_counter)++;
    cout << *my_counter << endl;
    delete my_counter;
    return 0;
}
```

```
==296== All heap blocks were freed -- no leaks are possible
==296==
==296== For lists of detected and suppressed errors, rerun with: -s
==296== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Returning an Instance

- If you don’t need to use a pointer, you also can return **an instance** of the object from a function.
- Note that, here, the **my_counter** variable will end up on the **local scope** of the **main function**.
- Again, the **valgrind output** shows all is well:

```
Counter make_counter(){
    Counter counter(0);
    return counter;
}

int main() {
    Counter my_counter = make_counter();
    my_counter++;
    cout << my_counter << endl;
    return 0;
}
```

```
==50== All heap blocks were freed -- no leaks are possible
==50==
==50== For lists of detected and suppressed errors, rerun with: -s
==50== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```


REMINDER
Delete the pointer when you're done
DONT FORGET THE Pointer
Hey man, don't forget to delete the pointer again!
Remember the pointer
I left a memory leak in the release again
CHARTOONS
FOR LACK OF A BETTER COMIC
JACOB ANDREWS
ProgrammerHumor.io

More Useful Resources For Practice:

- https://www.w3resource.com/cpp-exercises/basic/index.php
- https://www.w3schools.com/cpp/cpp_exercises.asp
- https://www.hackerrank.com/domains/cpp
- https://algoleague.com/

Questions?