Scientific Programming with C++

DYNAMIC MEMORY MANAGEMENT AND EXCEPTION HANDLING

Learning objectives

- After this lecture and related assignments, you will…
  - have an idea of how C++ uses memory storage
  - know when and how to to dynamically allocate memory
  - know how to safely allocate memory
  - be able to prevent some exceptions from crashing the program

Memory allocation in C++

- Most variables are allocated to either stack or heap
- Stack
  - Temporary memory storage
  - Memory is deallocated (released) automatically
  - Faster for memory allocation than heap
  - Less storage space than heap
- Heap
  - Permanent memory storage
  - Memory is allocated manually
    - Dynamic allocation
    - Data must also be deallocated manually
    - Unsafe
  - Used (safely) by some std functionalities like std::vector
  - Memory leaks possible

```
// stack allocation
int age = 54;
```

```
/* heap allocation:
- the pointer itself is on the stack
(variable that holds a memory address)
- the value 54 is on the heap (the integer
that is pointed to) */
int* age = new int(54);
```

```
/* the container itself is on the stack, but
its elements are on the heap */
std::vector<int> ages = { 54 };
```

How to use dynamic allocation

- Operators: new, delete, delete[]
- Allocate memory on the heap with "new"
  - Returns the address of the allocated memory

```
int* age = new int(54);
```

  - Can also be used to allocate an array of memory

```
int* ages = new int[3];
```

- Deallocate (release) memory on the heap with "delete"
  - Use "delete[]" to deallocate an array

```
delete age;
delete[] ages;
```

When to (manually) use the heap?

- You need a lot of memory (large objects)
  - Heap has more storage space than stack
- You want the data to be accessible outside the current scope
  - Otherwise, the pointer will point to a deallocated part of stack
- If there is no need to allocate on the heap, don't

Example: data is accessible outside the current scope

```
// note that this function passes pointers by reference; this allows us to modify the original pointers
void stackAndHeap(int*& a, int*& b) {
    // allocate integer 28 on the heap and have a point to it
    a = new int(28);
    // allocate integer 26 on the stack and have b point to it
    int num = 26;
    b = &num;
}

int main()
{
    int* age;
    int* anotherAge;
    stackAndHeap(age, anotherAge);
    std::cout << *age << std::endl; // valid, because the value pointed to is on the heap; accessible anywhere
    std::cout << *anotherAge << std::endl; // undefined behaviour, because the value pointed to is deallocated
    // anotherAge is a called dangling pointer and the memory address may now contain something else
    delete age; // always remember to deallocate dynamically allocated memory!
    anotherAge = nullptr; // this will "neutralize" the dangling pointer
}
```

Smart pointers

- Stack-allocated objects that wrap pointers
  - Deallocated automatically when it goes out of scope
- You can write your own or use existing ones
  - auto_ptr
    - Only one pointer can point to the same entity
    - Old version of unique_ptr
  - unique_ptr
    - Only one pointer can point to the same entity
    - New version of auto_ptr
  - shared_ptr
    - More than one pointer can point to the same entity
    - Counts how many pointers point to the same object with a reference counter
  - weak_ptr
    - Like shared_ptr, but doesn't participate in reference counting

Example: unique_ptr

```
// create two unique pointers, one pointing to a boolean and one null
std::unique_ptr<bool> booleanPtr(new bool(true));
std::unique_ptr<bool> anotherPtr;
//anotherPtr = booleanPtr; // not allowed, because unique pointers cannot share an object
anotherPtr = std::move(booleanPtr); // we can instead change "ownership" of the object with move()
```

Example: shared_ptr

```cpp
// create two shared pointers, one pointing to a boolean and one null
std::shared_ptr<bool> booleanPtr(new bool(true));
std::shared_ptr<bool> anotherPtr;

// count how many pointers share the object pointed to by each pointer
std::cout << booleanPtr.use_count() << std::endl;
std::cout << anotherPtr.use_count() << std::endl;

anotherPtr = booleanPtr; // allowed, because shared pointers can share an object
// count again
std::cout << booleanPtr.use_count() << std::endl;
std::cout << anotherPtr.use_count() << std::endl;
```

```
1
0
2
2
```

Avoiding memory leaks

- Avoid using raw pointers for dynamic allocation
  - Use smart pointers instead
- Modern C++ functionalities are typically safer than classic C
  - std::string versus char*
- For every “new”, there must be a “delete”
  - And the program must always be able to reach that delete
- Remember these also in class definitions
  - Any dynamically allocated attributes must be deleted

Example: memory leak

```
// return a pointer to a heap-allocated double
double* allocateValueOnHeap(double value) {
    return new double(value);
}

// demonstrate how memory leak can occur even if you use delete
int main() {

    // create a double pointer
    double *number = nullptr;

    // allocate different values on the heap and point to them
    number = allocateValueOnHeap(5.5);
    double* a = number;
    number = allocateValueOnHeap(-2.1);
    double* b = number;
    number = allocateValueOnHeap(0.3);
    double* c = number;

    // deallocate the latest value, BUT this will still leave all the others floating in heap space
    delete number;
    std::cout << *a << ", " << *b << ", " << *c;
}
```

5.5, -2.1, -1.45682e+144

Exception handling

- Exception is an error that occurs during execution
  - Program (usually) stops functioning
- Keywords of exception handling: try, catch, throw
- Exceptions can be “caught” to define what the program should do when one occurs
  - Allows the program to recover and continue execution
- You can also tell the program to “throw” an exception when a condition is met
  - Ensures that the program stops running unless the exception is handled or at least informs the user

Catching and throwing exceptions

- Enclose the code you want to evaluate for exceptions with “try”
  - Must be followed by catch() and an enclosing block
    - Defines what happens if the code enclosed with “try” throws an exception
    - Catch has a parameter of type std::exception that can be used to print an explanatory string with what()
- Throw exceptions with throw()
  - “throw” can also be used to rethrow exceptions that are being caught

```
if (idx >= 0)
    return vec[idx];
else
    throw std::invalid_argument("index given was negative, index cannot be negative");
```

Example: throw and try … catch

```
// if index is negative, throw exception; otherwise, return value from vector by index
int getByIndex(const std::vector<int>& vec, int idx) {
    if (idx >= 0)
        return vec[idx];
    else
        throw std::invalid_argument("index given was negative, index cannot be negative");
}

int main() {
    std::vector<int> numbers = { 2,6,14 };
    // try encloses the statements we are checking for exceptions
    try {
        std::cout << getByIndex(numbers, -3);
    } // catch defines what we do if we encounter an exception
    catch (std::exception& e) { // in this case, we print what the exception said
        std::cout << "Exception occurred, message: " << e.what() << std::endl;
    }
    // now let's try the same with a valid index (no exception will occur)
    try {
        std::cout << getByIndex(numbers, 2);
    }
    catch (std::exception& e) {
        std::cout << "Exception occurred, message: " << e.what() << std::endl;
    }
}
```

Exception occurred, message: index given was negative, index cannot be negative
14

Summary

- Use stack for data that is not needed outside the current scope
- Use heap for data that is needed beyond the current scope
- Data on stack is deallocated when it goes out of scope
- Data on heap persists if you manually allocated it
- Use pointers only when necessary (polymorphism, working with other libraries)
  - Prefer smart pointers whenever possible
- **Always** follow “new” with “delete”
- Try to fix the cause of the exception before using exception handling