Detecting and Preventing Memory Leaks

CS123

Preventing and detecting memory leaks is a crucial part of coding in C++. This doc describes the Valgrind Memory Analyzer, a built-in tool in QtCreator that automatically detects memory leaks, as well as some design patterns to keep in mind in order to prevent memory leaks from ever happening.

1 Valgrind Memory Analyzer

Valgrind is a great tool for automatically detecting memory leaks, and it is built in to QtCreator. To use it, select Analyze > Valgrind Memory Analyzer.

This will run your program, keeping track of the memory that is allocated and freed. After the program has finished running, you will see a message indicating any leaks that occurred. In the example below, we never deleted the ConstantBrush, so 32 bytes of memory were leaked. The message points us to the exact line where this memory was allocated.

If no memory was leaked during the execution of your program, you will not see any messages displayed in that section, as shown below.
Note that if no messages are displayed, it does not necessarily mean that your program is leak-free. It only means that no leaks occurred during that run of the program. For example, the code below only deletes the object if myBoolean is true. If myBoolean happened to be true when running Valgrind Memory Analyzer, no leak messages would show up, but your program can still leak memory if myBoolean is false.

```cpp
MyObject *object = new MyObject();
if (myBoolean) {
    delete object;
}
```

Valgrind can be a very useful tool for detecting memory leaks, but it will not find them if you don’t explicitly cause a certain branch of code to be executed while using it. As a result, you want to make sure you design your program such that memory leaks can be prevented in the first place.

## 2 Preventing Memory Leaks

The rule of thumb for creating objects in C++ is that every new should correspond to exactly one delete. Certain design patterns make it easier to keep track of this correspondence than others. For example, take a look at the following code, which involves creating an array of integers.

```cpp
int* makeArray(int size, int value) {
    int *array = new int[size];
    for (int i = 0; i < size; i++) {
        array[i] = value;
    }
    return array;
}
int main(int argc, char** argv) {
    int *array = makeArray(5, 5);
    // Do something with the array...
    delete[] array;
    return 0;
}
```

This program leaks memory because we “new”ed an array in the makeArray function, but never deleted it after we were finished using it. However, it is not obvious just by looking at the main function that the array needs to be deleted, because the “new” operation is hidden inside makeArray. This design pattern is not ideal because it requires callers of makeArray to adhere to a specific protocol in order to use the function properly (i.e. delete the returned array when you’re done with it).

Now, take a look at the alternative code below.

```cpp
void fillInArray(int *array, int size, int value) {
    for (int i = 0; i < size, i++) {
        array[i] = value;
    }
}
int main(int argc, char** argv) {
    int *array = new int[5];
    fillInArray(array, 5, 5);
    // Do something with the array...
    delete[] array;
    return 0;
}
```
In this example, the fillInArray does not handle the creation of the array, just filling it in with 5's. With this design, it is obvious just by looking at the main function that the array needs to be deleted in the main function. By making sure the “new” and the “delete” are in logically corresponding places, we have avoided the need to know the details of what fillInArray is doing. Each function is responsible for its own memory.

(Another advantage to this design is it makes our helper function have a more explicit and stand-alone purpose. Rather than being a “create AND fill in array” method, it is simply a “fill in array” method.)

Hopefully keeping this design pattern in mind will help you prevent memory leaks in the future!