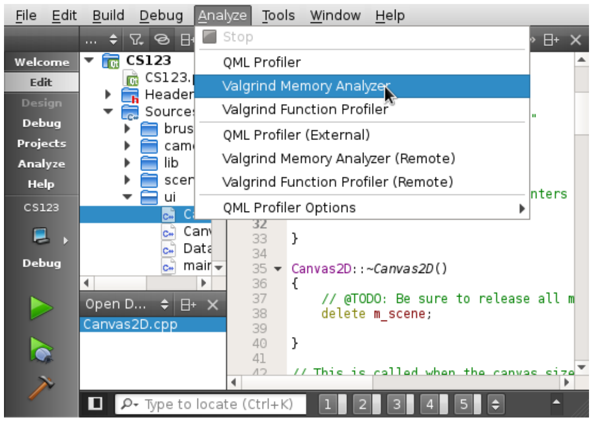
Detecting and Preventing Memory Leaks

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.

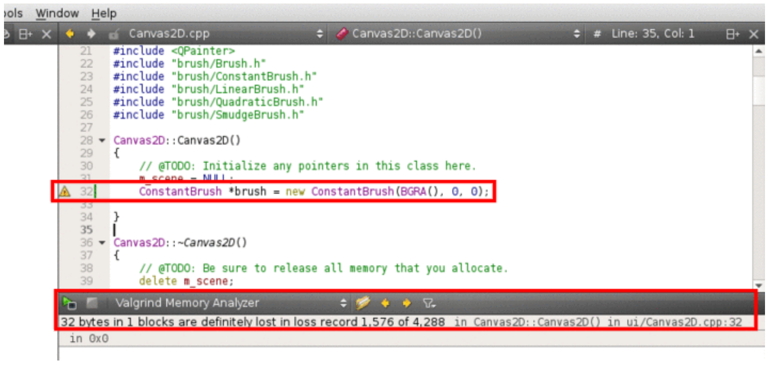
1 Valgrind Memory Analyzer

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.



```
}

Canvas2D::~Canvas2D()
{
    // @TODO: Be sure to release all m
    delete m_scene;

}

// This is called when the canvas size
```

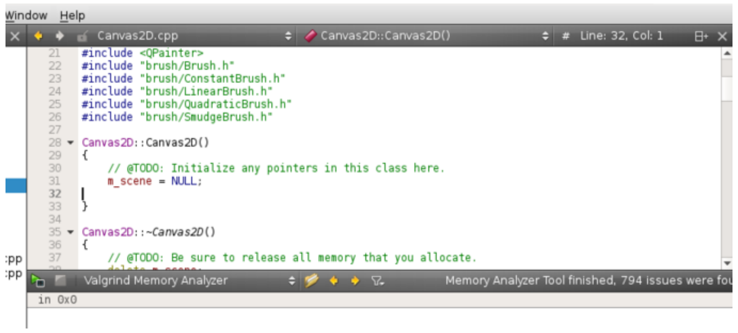
This will run your program, keeping track of the memory that is allocated and freed. After the program has finished running, you will see a message indicating any leaks that occurred. In the example below, we never deleted the ConstantBrush, so 32 bytes of memory were leaked. The message points us to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



```
#include <QPainter>
#include "brush/Brush.h"
#include "brush/ConstantBrush.h"
#include "brush/LinearBrush.h"
#include "brush/QuadraticBrush.h"
#include "brush/SmudgeBrush.h"

Canvas2D::Canvas2D()
{
    // @TODO: Initialize any pointers in this class here.
    m_scene = NULL;
    ConstantBrush *brush = new ConstantBrush(BGRA(), 0, 0);

}

Canvas2D::~Canvas2D()
{
    // @TODO: Be sure to release all memory that you allocate.
    delete m_scene;
```

```
32 bytes in 1 blocks are definitely lost in loss record 1,576 of 4,288  in Canvas2D::Canvas2D() in ui/Canvas2D.cpp:32
  in 0x0
```

If no memory was leaked during the execution of your program, you will not see any messages displayed in that section, as shown below.



```
#include <QPainter>
#include "brush/Brush.h"
#include "brush/ConstantBrush.h"
#include "brush/LinearBrush.h"
#include "brush/QuadraticBrush.h"
#include "brush/SmudgeBrush.h"

Canvas2D::Canvas2D()
{
    // @TODO: Initialize any pointers in this class here.
    m_scene = NULL;

}

Canvas2D::~Canvas2D()
{
    // @TODO: Be sure to release all memory that you allocate.
```

```
Memory Analyzer Tool finished, 794 issues were fou
  in 0x0
```

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.

```
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. You shouldn't need to worry about this if you're using smart pointers! Remember, don't allocate dynamic memory to a raw pointer.

Another source of common errors is arrays/vectors. As usual, using vectors over arrays whenever possible will solve many of your problems. You might run into the issue where you access or write to indices past the length of the vector. If you see unexplained segfaults, check your indices!