Qt - How a QObject with a parent on the stack might get deleted twice?

Asked 4 years, 5 months ago Active 4 years, 5 months ago Viewed 2k times

1

1

Can someone explain the following please?

> Stack or Heap? In general, a QObject without a parent should be created on the stack or defined as an subobject of another class. A QObject with a parent should not be on the stack because then it might get deleted twice accidentally. All QObjects created on the heap should either have a parent, or be managed somehow by another object.

Source: **LINK**

c++ qt qobject

asked May 22 '16 at 11:07

Michael Heidelberg
860 8 23

1 Answer

Active | Oldest | Votes

12

Sorry, downvoting the accepted answer. Both versions as currently explained are wrong, and especially the conclusion is very wrong.

Parent/child for memory management

Amongst other things, Qt uses the parent/child concept to manage memory. When an object is parented to another one, then

- **deleting the parent will also delete (via `operator delete`) all of its children.** Of course, this works recursively;
- **deleting a child will un-parent it, so that the parent will not attempt a double deletion.** `deleteLater` is not required for this to work -- *any* deletion will un-parent it.

This allows you to build a tree of `QObject`s via repeated dynamic allocations through `operator new`, and not having the problem of having to manually delete all the allocated objects. Just give them parents, and you'll only have to delete the root of the tree. You can also delete a child (i.e. a subtree) at any time, and that will do the right thing™.

```
class MyWidget : public QWidget // a QObject subclass
{
    Q_OBJECT
public:
    MyWidget(QWidget *parent = nullptr);

    // default destructor is fine!

private:
    // raw pointers:
    // we won't own these objects through these pointers.
    // we just need them to access the pointees
    QTimer *m_timer;
    QPushButton *m_button;
};

void MyWidget::MyWidget(QWidget *parent) : QWidget(parent)
{
    // don't need to save the pointer to this child. because reasons
    auto lineEdit = new QLineEdit(this);
    auto validator = new QIntValidator(lineEdit); // a nephew

    // but let's save the pointers to these children
    m_timer = new QTimer(this);
    m_button = new QPushButton(this);
    // ...
}
```

The default destructor will properly delete the entire tree, although we allocated the children objects through calls to `operator new`, and we didn't even bother to save in members the pointers to some children.

QObjects on the stack

You are allowed (and in certain contexts it's actually a good idea) to give parents to objects allocated on the stack.

The typical example is of `QDialog` subclasses:

```
void MyWidget::showOptionsDialog()
{
    // OptionsDialog is a QDialog subclass;
    // create an instance as a child of "this" object
    OptionsDialog d(this);

    // exec the dialog (i.e. show it as a modal dialog)
    conts auto result = d.exec();

    if (result == QDialog::Accept) {
        // apply the options
    }

    // d gets destroyed here
    // => it will remove itself as a child of this
}
```

The purpose of passing `this` as the dialog's parent is to allow the dialog to be centered

explained in `QDialog` docs. Also, ultimately, `d` does only need to live in that function, so it's a good idea to declare it as an automatic variable (i.e. allocated on the stack).

There you go: you've got a `QObject`, allocated on the stack, with a parent.

So what's the danger of `QObject`s on the stack? Consider this code:

```
QObject *parent = new QObject;
QObject child(parent);
delete parent;
```

As explained before, `parent` here will attempt to call `operator delete` on `child`, an object which was *not* allocated using `new` (it's on the stack). That's illegal (and a likely crash).

Obviously, nobody writes code like this, but consider the dialog example above again. What if, during the call `d.exec()`, we manage somehow to delete `this`, that is, the dialog's parent? That could happen for a variety of reasons very, very difficult to track down -- for instance, data arrives on a socket which causes widgets in your UI to change, creating some and destroying others. Ultimately, you would delete a stack variable, crashing (and having an extremely hard time trying to reproduce the crash).

Hence the suggestion to *avoid* creating code like that in the first place. It's not illegal, it may work, but it may also not work, and nobody likes fragile code.

edited May 22 '16 at 16:17

answered May 22 '16 at 13:50

peppe

19k 3 43 61

Thank you for the correct answer, peppe. I wrote too quickly without enough caffeine in my system. I'm slapping my forehead. – anonymous May 22 '16 at 16:07