Cutting the Call Stack

The problem

Many programmers know this as the "delete this" problem. You should avoid calling "delete this" unless you know the consequences and are prepared to handle them. The problem is that you are effectively killing the object while it is running i.e. pulling the rug under its feet. Any subsequent operations will be executing "dead" code and any attempts to modify member variables will crash the program. It is possible to get away with it by not modifying the member variables, but nevertheless its not considered very good practice.

You can encounter this problem even without explicitly calling "delete this". A very common situation is a plug-in or some long running background task that is owned and observed by another object. An object creates a "worker" object to perform the background task and sets itself as the observer for task completion. When the task is complete, the worker calls a notification function in the observer where the observer deletes the worker. "Delete this" has not been called, but the situation is the same. After the observer has deleted the worker, the notification function has to return to the already dead worker from where it was called.

Example of the problem

File: worker.h

```
#ifndef CWORKER_H
#define CWORKER_H

#include <e32base.h>

class MWorkerObserver
    {
public:
    virtual void WorkerDone() = 0;
    };

class CWorker : public CActive
    {
public:

    CWorker( MWorkerObserver& aObserver );
    ~CWorker();
    void Start();

private:

    void DoCancel();
    void RunL();

private: // data

    MWorkerObserver& iObserver;
    TBuf<30>         iMember;

    };

#endif // CWORKER_H
```

File: workerowner.cpp

```
void CWorkerOwner::DoSomethingL()
    {
    iWorker = new (ELeave) CWorker( *this );
    iWorker->Start();
    }

void CWorkerOwner::WorkerDone()
    {
    delete iWorker;
    iWorker = NULL;
    }
```

File: worker.cpp

```
#include "worker.h"

CWorker::CWorker( MWorkerObserver& aObserver )
    : CActive( CActive::EPriorityStandard )
    , iObserver( aObserver )
    {
    CActiveScheduler::Add( this );
    }

CWorker::~CWorker()
    {
    Cancel();
    }

void CWorker::Start()
    {
    iStatus = KRequestPending;
    TRequestStatus* stat = &iStatus;
```


(C) Copyright Nokia 2013. All rights reserved.

```
    User::RequestComplete( stat, KErrNone );
    SetActive();
    }

void CWorker::DoCancel()
    {
    }

void CWorker::RunL()
    {
    // Notify the observer. Worker gets deleted here
    iObserver.WorkerDone();

    // Attempt to modify member data and crash
    iMember.Copy( _L("test") );
    }
```

Solution 1: Burying your head in the sand

One simple solution is to ignore the problem and simply make a hasty exit from the worker when it becomes a zombie.

Only the modified parts of the code are shown here. Only the worker requires changes from the initial example.

File: worker.h

```
private: // data

    ...
    TBool*           iDeleted;
```

File: worker.cpp

```
CWorker::~CWorker()
    {
    Cancel();
    // Flip the flag to know we are dying
    // check for iDeleted != NULL because
    // destructor could be called without reaching RunL
    if (iDeleted) *iDeleted = ETrue;
    }

void CWorker::RunL()
    {
    // Declare a flag to tell us if we have died
    TBool deleted = EFalse;
    iDeleted = &deleted;

    // Notify the observer. Worker gets deleted here
    iObserver.WorkerDone();

    if ( deleted )
        {
        // We're dead, get out fast
        return;
        }
     else
        {
        // We need to set iDeleted to Null just to ensure
        //if destructor is called, it will be pointing to the
        // last address which is a junk address now
         iDeleted = NULL;
         }

    // Attempt to modify member data and crash
    iMember.Copy( _L("test") );
    }
```

Solution 2: Cut the Call Stack With CIdle

The second widely used solution is to use **CIdle** in the worker owner to cut the call stack.

Only modified parts are shown here. Only the worker owner requires changes from the initial example.

File: workerowner.cpp

```
void CWorkerOwner::WorkerDone()
    {
    TRAPD( err, iCallStackCutter = CIdle::NewL( CActive::EPriorityStandard ) );
    if ( !err )
        {
        iCallStackCutter->Start( TCallBack( DeleteWorker, this ) );
        }
    }

// Static callback function to do the deletion from another call stack
TInt CWorkerOwner::DeleteWorker( TAny* aSelf )
    {
    CWorkerOwner* self = static_cast<CWorkerOwner*>( aSelf );
    delete self->iWorker;
    self->iWorker = NULL;

    return EFalse; // Not called again
    }
```


(C) Copyright Nokia 2013. All rights reserved.

Solution 3: Call Arnie

The basic idea for the third solution doesn't much differ from the idea behind the second one. The inconvenience in the second solution is that if you have a big project and many situations like this, you need one static function and **CIdle** for each deletion. To get rid of the static functions and **CIdle**s you can make an object whose job is to kill other objects. Making it a singleton would make it easily accessible to all classes in the project.

The terminator keeps a list of pointers to pointers so it can delete the object and null the pointer from the owner. It is a high priority active object to make the deletion to happen as soon as possible, although that isn't very important.

Only the worker owner from the initial example requires changes to call the terminator.

File: terminator.h

```
#ifndef CTERMINATOR_H
#define CTERMINATOR_H

#include <e32base.h>

class CTerminator : public CActive
    {
public:

    CTerminator();
    ~CTerminator();

    template<typename T>
    inline TInt HastaLaVistaBaby( T*& aVictim );

private:

    void DoCancel();
    void RunL();

    TInt AddToHitlist( CBase** aVictim );
    void IllBeBack();

private: // data

    // Victims. Pointers to pointers
    RPointerArray<CBase*>   iHitlist;

    };

template<typename T>
inline TInt CTerminator::HastaLaVistaBaby( T*& aVictim )
    {
    CBase** victim = reinterpret_cast<CBase**>( &aVictim );
    return AddToHitlist( victim );
    }

#endif // CTERMINATOR_H
```

File: terminator.cpp

```
#include "terminator.h"

CTerminator::CTerminator()
    : CActive( CActive::EPriorityHigh )
    {
    CActiveScheduler::Add( this );
    }

CTerminator::~CTerminator()
    {
    Cancel();
    iHitlist.Close();
    }

TInt CTerminator::AddToHitlist( CBase** aVictim )
    {
    TInt err = iHitlist.Append( aVictim );
    if ( !err )
        {
        IllBeBack();
        }
    return err;
    }

// Completes request to return immediately from another call stack
void CTerminator::IllBeBack()
    {
    iStatus = KRequestPending;
    TRequestStatus* stat = &iStatus;
    User::RequestComplete( stat, KErrNone );
    SetActive();
    }

void CTerminator::DoCancel()
    {
    }

void CTerminator::RunL()
    {
    for ( TInt i = 0; i < iHitlist.Count(); ++i )
        {
        CBase** victim = iHitlist[i];

        // Dead ...
        delete *victim;

        // ... and buried
        *victim = NULL;
        }
    iHitlist.Reset();
    }
```


(C) Copyright Nokia 2013. All rights reserved.

File: workerowner.cpp

```
void CWorkerOwner::ConstructL()
    {
    ...
    iArnie = new (ELeave) CTerminator;
    }

void CHelloWorldBasicAppUi::WorkerDone()
    {
    iArnie->HastaLaVistaBaby( iWorker );
    }
```


(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.