Kapitel 17
Prä- und Postprozessing

17.1 Hintergrund

Ein Beispiel für die Problematik von Prä- und Postprozessing ist die Umstellung von Programmen und Bibliotheken auf Multi-Threading-Umgebungen. In der Regel werden Methoden nicht „thread-safe“ programmiert, d.h. wird die Funktion von einem Thread aufgerufen, während der Bearbeitung aber von einem anderen Thread unterbrochen, der die gleiche Funktion erneut aufruft, so kommt es zu Inkonsistenzen der Daten, weil Speicherbereiche vom zweiten Thread überschrieben werden, bevor sie an den ersten zurückgegeben werden konnten. Die Standardvorgehensweise in diesem Fall ist die Analyse der Bibliothek und Neuprogrammierung der unsicheren Funktionen, wobei die Schnittstelle nicht verändert werden darf.

Eine elegantere Lösung wäre der automatische Aufruf von Entry- und Exit-Funktionen jeweils bei Aufruf und Verlassen einer Methode, wobei die notwendigen Aktionen wie Setzen und Rücksetzen eines Mutex vorgenommen werden. C++ bietet diese Option nicht im Sprachumfang an. Die folgenden Template-Klassen sollen helfen, diese Probleme zu lösen, wobei die notwendigen Programmänderungen so weit wie möglich auf einige zentrale Datentypenänderungen beschränkt bleiben sollen.

Ein anderes Beispiel ist eine Netzanwendung, in der die einzelnen Objekte auf verschiedenen Maschinen liegen. Prä- und Postprozessing wären hier eine sinnvolle Ergänzung, um beim Eintreten in ein Objekt oder beim Verlassen eines Objektes den anderen Teilnehmern entsprechende Signale zukommen zu lassen. Die Anwendung selbst könnte weiterhin so programmiert werden, als würde alles auf einer Maschine ablaufen.

17.2 Präprozessing

Die Problematik ist leicht zu lösen, wenn wir uns auf Präprozessing beschränken, d.h. nur zu Beginn einer Methode eine bestimmte Funktion ausgeführt werden muss. Wir können hierbei auf die Erfahrungen mit der Zeigervariablenverwaltung zurückgreifen.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_17,

Die vom Präprozessing betroffenen Programmteile seien Objekte bestimmter Klassen. Die Objekte werden als Zeigerobjekte instanziiert und mit der Smart-Pointer-Methode verwaltet. Das Präprozessing sei bei allen Methoden der Klassen auszuführen.

17.2.1 Präprozessing mit spezieller Funktion

Die auszuführende Methode sei vom Typ

```
void function(void)
```

Die Methode wird an das Smart-Pointer-Objekt gebunden, wobei für unterschiedliche Smart-Pointer-Objekt unterschiedliche Praprozessing-Methoden festgelegt werden dürfen. Die Bindung erfolgt im Konstruktor, der Methodenaufruf in den Zugriffsoperatoren. Bei Austausch des vom Smart-Pointer verwalteten Objektes bleibt die Bindung der Präprozessing-Funktion erhalten.

```
template <class T, class P> class PreProcA: public Ptr<T>
{
protected:
    P function;
    PreProcA();

public:
    PreProcA(P fu): function(fu) {};

    inline PreProcA& operator=(T* t)
         { Ptr<T>::operator=(t); return *this; }
    inline T* operator->(void)
         { function(); return  Ptr<T>::ptr; };
    inline const T* operator->(void)const
         { function(); return  Ptr<T>::ptr; };
};//end class
```

17.2.2 Präprozessing durch Objektmethode

Der Präprozessing-Prozess kann weitere Einstellungen erfordern oder Statusinformationen benötigen, die auch in anderen Programmteilen berücksichtigt werden können. Anstelle einer einzelnen Funktion empfiehlt sich in diesem Fall die Instanziierung eines Präprozessing-Objektes, das im Konstruktoraufruf der Smart-Pointer-Klasse mitsamt einer Klassenmethode des Typs `void function(void)` übergeben wird. Um mehrere Smart-Pointer-Objekten durch das gleiche Präprozessingobjekt managen zu lassen, wird eine Referenz auf das Objekt übernommen. Zusätzlich Klassenmethoden der Smart-Pointer-Klasse erlauben den Zugriff auf das Präprozessingobjekt

```
template <class T, class P> class PreProcB: public Ptr<T>
{
protected:
    typedef void (P::*function)();
    function f;
    P& po;
    PreProcB();

public:
    PreProcB(P& o, function fu): po(o), f(fu) {};

    PreProcB& operator=(T* t)
       { Ptr<T>::operator=(t); return *this; }
    inline T* operator->(void)
       { (po.*f)(); return  Ptr<T>::ptr; };
    inline const T* operator->(void)const
       { (po.*f)(); return  Ptr<T>::ptr; };

    inline P&       p_obj()      { return po; }
    inline P const& p_obj()const { return po;}
};//end class
```

Der Zugriff auf das Präprozessingobjekt über das Smart-Pointer-Objekt erlaubt auch die Instanziierung weiterer Smart-Pointer-Objekte in Programmbereichen, in denen auf das Präprozessingobjekt nicht zugegriffen werden kann.

Es muss programmtechnisch allerdings dafür gesorgt werden, dass das Präprozessingobjekt während der gesamten Programmlaufzeit nicht zerstört wird, es muss aber nicht für eine Linkerauflösung des Objektnamens in der gesamten Anwendung gesorgt werden.

17.2.3 Präprozessing mit Singleton-Objekt

Die bisher vorgestellte Methodik hat zur Folge, dass alle Konstruktoraufrufe von Smart-Pointer-Objekten geändert werden müssen. Das ist zwar schon ein deutlicher Vorteil gegenüber einer gesamten Neuprogrammierung, birgt aber immer noch Fehlerquellen in sich, da vergessene Umbenennungen in der Regel nicht auffallen.

Sind die Smart-Pointer allerdings ihrerseits durch `typedef` typisiert und wird nur ein Singleton-Präprozessing-Objekt benötigt, so kann das Präprozessing im Rahmen der Typisierung vorgenommen werden. Eine Hilfsklasse dient zunächst zur Typisierung der für das Präprozessing eingesetzten Klassenmethode:

```
template <class T> struct MemberFunctionPointer {
    typedef void (T::*function)(void);
};//end struct
```

Mit Hilfe dieser Typisierung kann nun die Präprozessingklasse und die zugehörenden Klassenmethode in einer Template-Vereinbarung untergebracht werden:

```
template <class T, class P,
          typename MemberFunctionPointer<P>::function f>
    class PreProcC: public Ptr<T> {
protected:
    static inline P& P_obj(){ static P obj; return obj;}
public:
    PreProcC(){};


    PreProcC& operator=(T* t)
        { Ptr<T>::operator=(t); return *this; }
    inline T* operator->(void){
        (p_obj().*f)(); return  Ptr<T>::ptr; };
    inline const T* operator->(void)const{
        (p_obj().*f)(); return  Ptr<T>::ptr; };


    inline P&       p_obj()      { return P_obj(); }
    inline P const& p_obj()const { return P_obj();}
};//end class
```

Die Hilfsklasse erlaubt nämlich die Übergabe der Methodenadresse als Nicht-Klassenparameter im Template-Aufruf (ähnlich einem `int`-Parameter). Das Präprozessingobjekt wird in der üblichen Form als Singleton-Objekt implementiert.

17.2.4 Varianten, Kritik

Das Präprozessingmodell ist recht einfach zu implementieren. Unter bestimmten Voraussetzungen – globale Typisierung, Singleton-Objekt – kann sogar eine Implementierung durch Änderung an einer Stelle einer größeren Anwendung fehlerfrei gelingen. Weitere Varianten, beispielsweise eine einzelne Methode wie in 2.1, die jedoch mit der Methodik von 2.3 typisiert wird, oder individuelle Präprozessingobjekte für jedes einzelne Smart-Pointer-Objekt, lassen sich leicht aus den vorgestellten Varianten nach Bedarf generieren.

Das Modell führt allerdings ein Präprozessing für alle Methoden der in den Smart-Pointern verwalteten Objekte aus, auch wenn dies nicht notwendig sein sollte. Betroffen sind auch direkte Attributzugriffe von öffentlich zugänglichen Attributen. Eine Rückkopplung zwischen Praprozessingobjekt und Smart-Pointer-Objekt ist auf allgemeiner Basis nicht möglich.

Das Präprozessing wird auch nur bei direkten Funktionsaufrufen durchgeführt. Enthält das Programm beispielsweise Code der Form

```
void foo(A const& a) {
   ...
   a.f();
   ...
}

void main(){
   PreProc<A,b> p(..);
   ...
   foo(*p);
   ...
```

so wird das Smart-Pointer-Objekt ohne den Smart-Pointer an die Funktion übergeben, d.h. in der Funktion aufgrufene Funktionen des Parameterobjektes werden ohne Präprozessing ausgeführt. Es ist zwar möglich, weitere Operatoren in der Smart-Pointer-Klasse neu zu definieren, so dass zumindest bei Parameterübergabe das Präprozessing durchgeführt wird. Die Frage ist jedoch, ob das funktionsmäßig genügt; wir haben aus diesem Grunde die Erweiterungen nicht durchgeführt.

Wie die Operatorendefinitionen zeigen, wird bei einem Objektzugriff zunächst die Präprozessingmethode ausgeführt und anschließend eine Referenz auf das innere Objekt an das rufenden Programm übergeben. Ein Postprozessing ist auf diese Weise nicht implementierbar, da nicht mehr in den Operator zurückgekehrt wird.

17.3 Prä- und Postprozessing

Soll Prä- und Postprozessing durchgeführt werden, so bleibt kein anderer Weg als den Funktionsaufruf vollständig in einer anderen Funktion zu kapseln, was sich sinngemäß folgendermaßen formulieren lässt:

```
RT foo(P p) { ... }
...
RT t = foo(p); ...

=======>

RT wrap(RT (*foo)(P) , P p){
   pre();
   RT temp=(*foo)(p);
```

```
   post();
   return temp;
}//end function
...
RT t = wrap(&foo,p);
```

Wir werden die Implementation für den Aufruf von Klassenmethoden mit Prä- und Postprozessing entwickeln. Hierbei ist zu berücksichtigen, dass Methoden mit oder ohne (`void`) Rückgabewert und mit unterschiedlichen Parameteranzahlen implementiert sein können.

17.3.1 Prä- und Postprozessing-Methodenverwaltung

Wie aus dem Funktionsbeispiel hervorgeht, lässt sich die Aufgabe nicht mehr zentral lösen, sondern jede Programmzeile, die in das Prä- und Postprozessing eingebezogen werden muss, ist neu zu programmieren. Die Template-Varianten bei Template-Methoden sind etwas begrenzt, weshalb wir das Funktionsbeispiel in Klassenaufrufe, so genannte Funktoren, umsetzen. Andererseits erlauben Methode oft die Vermeidung der genauen Spezifizierung der Typen; man kann die Auflösung dem Compiler überlassen. Template-Methoden werden also zur Erzeugung der Funktorobjekte zum Einsatz kommen.

Verschiedene Funktoren sind für den Compiler jedoch auch verschiedene Datentypen, so dass auch die Prä- und Postprozessing-Methoden an jeder Aufrufstelle spezifiziert werden müssten. Aus diesem Grunde führen wir zunächst eine zentrale Verwaltung der Methoden ein. Methodenpaare werden auf einer zentralen Vektor-Speicherstruktur hinterlegt.

```
typedef void(*vfunc)();
static vector<pair<vfunc,vfunc> >
prozessing_function_store;
```

Im Startteil der Anwendung muss dieser Vektor mit Methodenpaare versorgt werden; dies kann nicht zur Compilezeit geschehen.

```
void pre() { ... }
void post(){ ... }
...
prozessing_function_store.push_back(
                   pair<vfunc,vfunc>(&pre,&post));
```

Auf diese Weise können verschiedene Prä- und Postprozessing-Methodengruppen definiert werden. Der Zugriff erfolgt über den Index im Container, d.h. auf die

Reihenfolge bei der Initialisierung muss geachtet werden (*eine leider kaum zu umgehende Fehlerquelle*).

Anmerkung: wir haben hier ein einfaches Methodenmodell gewählt. Der Leser kann für seine Bedürfnisse natürlich auf ähnliche Art auch Objektmodelle mit oder ohne Singletons implementieren.

17.3.2 Methodentypisierung

Methoden können mit oder ohne Übergabeparameter und mit oder ohne Rückgabewert implementiert sein. Mittels einiger Template-Spezialisierungen lassen sich die Methoden eindeutig typisieren:

```
template <class T, class RT=void, class P1=void, class
P2=void>
struct FuncPtr {
    typedef RT (T::*function)(P1,P2);
};//end struct
template <class T, class RT, class P1>
struct FuncPtr<T,RT,P1,void> {
    typedef RT (T::*function)(P1);
};//end struct

template <class T, class RT>
struct FuncPtr<T,RT,void,void> {
    typedef RT (T::*function)();
};//end struct

template <class T>
struct FuncPtr<T,void,void,void> {
    typedef void (T::*function)();
};//end struct
```

Wir haben dies hier für zunächst maximal zwei Übergabeparameter durchgeführt. Sollten mehr Parameter notendig sein, kann die Klassendefinition beliebig nach „oben“ fortgesetzt werden. Bei Übergabe einer beliebigen Funktion als Nicht-Klassen-Template-Parameter wie oben in `PreProcC` kann über diese Typisierung nun der exakte Typ ermittelt und für die passende Spezialisierung von Zugriffsmethoden ermittelt werden.

Ein Problem stellt hierbei aber noch der Datentyp `void` dar. Einen Instanziierungsversuch in der Art

```
void p;
```

beantwortet der Compiler durch harsche Fehlermeldungen. Um „leere“ Instanziierungen, d.h. Methoden, die solche Instanzen definieren, jedoch nicht implementiert werden, kommen wir aber nicht herum. Zur Abhilfe definieren wir noch die Klasse

```
template <class U> struct IsType {
    enum { is_type=true};
    typedef U type;
};//end struct

template <> struct IsType<void> {
private:
    struct void_replace_type {};
public:
    enum {is_type=false };
    typedef void_replace_type type;
};//end struct
```

Bei jedem beliebigen Typ außer `void` stellt uns diese Klassen den Typ nochmals zur Verfügung und gibt über die Boolsche Variabel bekannt, dass es sich um einen instanziierbaren Typ handelt, bei `void` stellt sie einen instanziierbaren Ersatztyp zur Verfügung und notiert das in der logischen Variabeln.

17.3.3 Die Funktorklasse(n)

Mit diesem Handwerkszeug ausgerüstet können nun die Funktorklassen eingerichtet werden. Hier zunächst die Implementation der Basisklasse:

```
template <class T, class RT, class P1, class P2,
          int fn=0, bool b=true>
class PrePostProc {
private:
    typedef typename FuncPtr<T,RT,P1,P2>::function function;
    T* t;
    function func;
public:
    PrePostProc(T* obj, function fu): t(obj),func(fu){}
    RT operator()(){
      if(prozessing_function_store.size()>fn)
           prozessing_function_store.at(fn).first();
        RT tmp= (t->*func)();
        if(prozessing_function_store.size()>fn)
            prozessing_function_store.at(fn).second();
        return tmp;}
    ...
    RT operator()(typename IsType<P1>::type p1,
                  typename IsType<P2>::type p2){
        if(prozessing_function_store.size()>fn)
            prozessing_function_store.at(fn).first();
```

```
        RT tmp= (t->*func)(p1,p2);
        if(prozessing_function_store.size()>fn)
            prozessing_function_store.at(fn).second();
        return tmp;}
};//end class
```

Die Funktorklasse ist für Objekte mit Klassenmethoden und virtuelle Vererbung ausgelegt. Die Klassenmethoden können in dieser Variante wieder bis zu zwei Übergabeparameter aufweisen. Eine Anweisung

```
T* obj = new T();
...
obj->methode();
```

wird ersetzt durch

```
class TT: T {...
T* obj = new TT();
...
PrePostProc<T,int,void,void,0,true>
Functor(obj,&T::method);
Functor();
```

Durch den Operator `operator->*` im Funktor-Operator `operator()` wird vererbungsrichtig die Methode `TT::method` aufgerufen, sofern die Methode als `virtual` deklariert ist.
Das Prä- und Postprozessingmethodenpaar wird durch den Templateparameter `fn` spezifiert. Die Reaktion auf nicht definierte Paare kann beispielsweise auch durch das Werfen von Ausnahmen ersetzt werden.

Der ursprüngliche Methodenaufruf wird durch eine Funktorinstanziierung mit anschließendem Aufruf des Funktor mit dem `operator()` substituiert. Diesen Operator müssen wir entsprechend der unterschiedlichen Anzahlen der Übergabeparameter mehrfach überschreiben, wobei die Funktionstypisierung nun den exakten Aufruf der Klassenmethoden erlaubt. Die Typisierung von `void` mit der Hilfsklasse `IsType` vermeidet Compilerfehlermeldungen im ersten Compilerdurchlauf bei den Operatorvarianten mit mehreren Aufrufparametern; da diese Methode im zweiten Durchlauf nun auch nicht übersetzt werden, erhalten wir ausführbaren Code ohne Fehlermeldung.

Ein Problem bleibt hierbei allerdings der Rückgabetyp `void`, der bei der Deklaration der temporären Variablen zu Fehlermeldungen führt. Hier hilft nur eine Spezialisierung der gesamten Klasse weiter, die diese Variablen nicht besitzt:

```
template <class T, class RT, class P1, class P2, int fn>
class PrePostProc<T,RT,P1,P2,fn,false> {
   ...
```

```
   void operator()()
{  if(prozessing_function_store.size()>fn)
      prozessing_function_store.at(fn).first();
   (t->*func)();
   if(prozessing_function_store.size()>fn)
      prozessing_function_store.at(fn).second();}
   ...
};//end struct
```

Die Steuerung, welche Klasse vom Compiler instanziiert wird, erfolgt über den Boolschen Templateparameter, den wir nun noch passen von der Hilfsklasse `IsType` abnehmen müssen.

17.3.4 Instanziierungsmethoden

Die Instanziierung der Funktoren ist in dieser Form ziemlich umständlich, da sich der Programmierer um die richtige Besetzung der Templateparameter kümmern muss. Funktionstemplates können diese jedoch automatisch ermitteln, weshalb wir die Funktorinstanziierung mehreren überladenen Methoden überlassen:

```
template <int fn, class T, class RT>
PrePostProc<T,RT,void,void,void,fn,IsType<RT>::is_type>
    make_fptr(T* obj, RT (T::*f)()){
      return PrePostProc<T,RT,void,void,void,
                         fn,IsType<RT>::is_type>(obj,f);
}//end function
...
template <int fn, class T, class RT, class P1, class P2>
PrePostProc<T,RT,P1,P2,void,fn,IsType<RT>::is_type>
   make_fptr(T* obj, RT (T::*f)(P1,P2)){
      return PrePostProc<T,RT,P1,P2,void,
                         fn,IsType<RT>::is_type>(obj,f);
}//end function
```

Die Arbeit im Programm verkürzt sich damit von

```
  obj->method(p);
```

auf

```
  make_fptr<0>(obj,&T::method)(p);
```

Das lässt sich in größeren Anwendungen aber vermutlich auch durch entsprechende Parser erledigen, was die Fehleranfälligkeit gering hält.