Liebe Free Pascal Gemeinschaft,

Das Free Pascal Developer Team freut sich, endlich ein lang erwartetes Feature ankündigen zu können, genauer gesagt sind es zwei verschiedene, aber sehr verwandte Features: Funktionsreferenzen und Anonyme Funktionen. Diese beiden Features können unabhängig voneinander verwendet werden, entfalten aber ihre größte Kraft, wenn sie zusammen verwendet werden.

Diese Features basieren auf der Arbeit von *Blaise.ru*, vielen Dank dafür und ich hoffe, dass es Ihnen angesichts der aktuellen Situation gut geht.

Im Folgenden werden wir beide Funktionen einzeln beleuchten und dann einen Blick auf ihre gemeinsame Verwendung werfen.

Funktionsreferenzen

Funktionsreferenzen (auch anwendbare Namen sind Prozedurreferenzen und Routinenreferenzen, im Folgenden werden nur Funktionsreferenzen verwendet) sind Typen, die eine Funktion (oder Prozedur oder Routine), Methode, Funktionsvariable (oder Prozedurvariable oder Routinenvariable), Methodenvariable, verschachtelte Funktion (oder verschachtelte Prozedur oder verschachtelte Routine) oder eine anonyme Funktion (oder anonyme Prozedur oder anonyme Routine) als Wert annehmen können. Die Funktionsreferenz kann dann verwendet werden, um die bereitgestellte Funktion aufzurufen, genau wie andere ähnliche Routinezeigertypen. Im Gegensatz zu diesen anderen Typen können ihm fast alle funktionsähnlichen Konstrukte zugewiesen werden (die einzige Ausnahme sind verschachtelte Funktionsvariablen (oder verschachtelte Prozedurvariablen oder verschachtelte Routinevariablen), dazu später mehr) und dann verwendet oder gespeichert werden.

Funktionsreferenzen werden mit dem Modeschalter `FUNCTIONREFERENCES` aktiviert (die folgenden Beispiele gehen davon aus, dass dieser Modeschalter vorhanden ist).

Eine Funktionsreferenz wird wie folgt deklariert:

`REFERENCE TO FUNCTION|PROCEDURE [(`*`argumentlist`*`)][: `*`resulttype`*`;] [`*`directives`*`;]`

Beispiele:

Code: Pascal [Wählen][+]

```
1. Typ
2.   TProcLongInt = Referenz auf procedure(aArg: LongInt); stdcall;
3.   TFuncTObject = Referenz auf function(aArg: TObject): TObject;
```

Wie andere Funktionszeigertypen können auch Funktionsreferenzen als generisch deklariert werden:

Code: Pascal [Wählen][+]

```
1. Typ
2.   generic TGenericProc<T> = Verweis auf procedure(aArg: T);
```

Wie Sie sehen können, können Sie, sobald Funktionsreferenzen aktiviert sind, den Bezeichner "REFERENCE" nicht mehr als Teil einer Alias-Deklaration verwenden, ohne "& " zu benutzen:

Code: Pascal [Wählen][+]

```
1. Typ
2.   someref = reference; // wird fehlschlagen
3.   someref = &reference; // korrekte Korrektur
4.
5. var
6.   somevar: Referenz; // wird fehlschlagen
7.   somevar: &reference; // richtige Korrektur
```

Eine Funktionsreferenzvariable kann dann wie jeder andere Funktionszeigertyp aufgerufen werden:

Code: Pascal [Wählen][+]

```
1. var
```

```
  p: TProcLongInt;
beginnen
  p := @SomeLongIntProc;
  p(42);
Ende.
```

Wenn eine Funktionsreferenz keine Parameter hat, müssen Sie in den FPC/ObjFPC-Modi trotzdem "()" verwenden, wie bei anderen Funktionszeigertypen:

Code: Pascal [Wählen][+]

```
Typ
  TProc = Verweis auf Prozedur;
var
  p: TProc;
beginnen
  p := @SomeProcedure;
  p(); // erforderlich
  p; // dies kann z.B. im Modus Delphi verwendet werden
Ende.
```

Wie andere Funktionszeigertypen können sie auch anonym als Teil einer Variablen- oder Felddeklaration deklariert werden (aber nicht als Teil einer Parameterdeklaration):

Code: Pascal [Wählen][+]

```
var
  f: Verweis auf Funktion: LongInt;

Typ
  TTest = Klasse
    f: Verweis auf das Verfahren;
  Ende;
```

Sie erhalten ihre große Macht von einem Punkt, der ausnahmsweise *nicht* als Implementierungsdetail betrachtet wird: Funktionsreferenzen werden intern als referenzierte Schnittstellen mit einer einzigen Invoke()-Methode der vorgesehenen Signatur deklariert. Die obigen Beispiele sind also in Wirklichkeit so deklariert:

Code: Pascal [Wählen][+]

```
Typ
  TProcLongInt = Schnittstelle(IInterface)
    procedure Invoke(aArg: LongInt); stdcall; overload;
  Ende;

  TFuncTObject = Schnittstelle(IInterface)
    procedure Invoke(aArg: TObject): TObject; overload;
  Ende;

  generisch TGenericProc<T> = Schnittstelle(IInterface)
    procedure Invoke(aArg: T); overload;
  Ende;
```

Dies hat einige Auswirkungen:

- in der RTTI wird dies wie eine normale Schnittstelle erscheinen
- es reagiert auf die $M-Richtlinie wie eine normale Schnittstelle
- es ist ein verwalteter Typ
- er hat *keine* gültige GUID
- sie kann durch eine Klasse implementiert werden
- es kann vererbt werden von

Besonders die letzten beiden Punkte sind wichtig.

Dass die Schnittstelle implementiert werden kann, bedeutet, dass einer Funktionsreferenz viel mehr Funktionalität und Zustand hinzugefügt werden kann:

Code: Pascal [Wählen][+]

```
1. Typ
2.   TFunc = Verweis auf Funktion: LongInt;
3.
4.   TSomeImpl = class(TInterfacedObject, TFunc)
5.     f: LongInt;
6.     function Invoke: LongInt;
7.   Ende;
8.
9.  Funktion TSomeImpl. Aufrufen: LongInt;
10.       beginnen
11.  Ergebnis := f;
12. Ende;
13.
14.       var
15.  t: TSomeImpl;
16.  f: TFunc;
17.       beginnen
18.  t := TSomeImpl. Erstellen;
19.  f := t;
20.  Writeln(f()); // wird 0 schreiben
21.  t. f := 42;
22.  Writeln(f()); // wird 42 schreiben
23.  f := Nil; // es gelten die üblichen Warnungen vor der Vermischung von Klassen und
   Schnittstellen!
24. Ende.
```

Da Funktionsreferenzen keine gültigen GUIDs haben, können Sie jedoch weder `QueryInterface()` noch den `as-Operator` verwenden, um sie abzurufen. Die Verwendung der Low-Level-Interface-bezogenen Funktionen von `TObject` wird jedoch funktionieren.

Eine Schnittstelle, die von einer Funktionsreferenz erbt, wird vom Compiler immer noch als aufrufbar angesehen, so dass sie wie eine gewöhnliche Funktionsreferenz verwendet werden kann, aber Sie können auch zusätzliche Methoden einschließlich Überladungen für Invoke selbst hinzufügen:

Code: Pascal [Wählen][+]

```
1. Typ
2.   TTest = Referenz auf procedure(aArg: TObject);
3.
4.   TTestEx = Schnittstelle(TTest)
5.     function Invoke: TObject; Überladung;
6.   Ende;
7.
8. var
9.   f: TTestEx;
10.  o: TObject;
11.       beginnen
12.  f := TSomeImplEx. Erstellen;
13.  o := f();
14.  f(o);
15. Ende.
```

Dies ist zum Beispiel von Stefan Glienke in seinem Blog beschrieben. Das von ihm verlinkte Beispiel funktioniert jedoch aufgrund der fehlenden Funktionalität in `Rtti.TValue` so nicht.

Wie eingangs erwähnt, kann man einer Funktionsreferenz eine verschachtelte Funktion zuweisen, aber nicht eine verschachtelte Funktionsvariable. Dafür gibt es keinen wirklichen technischen Grund, sondern es handelt sich um eine Design-Entscheidung, die darauf beruht, wie sich Funktionsreferenzen verhalten sollen: Es wird angenommen, dass sie über ihren Anwendungsbereich hinaus gültig sind

(dies wird in Kombination mit anonymen Funktionen im dritten Teil deutlicher werden), so dass sie beispielsweise von einer Funktion zurückgegeben oder in einer Klasseninstanz gespeichert werden können und immer noch als gültig gelten können. Eine verschachtelte Funktionsvariable ist jedoch nicht mehr verwendbar, sobald der Funktionsrahmen, aus dem sie abgerufen wurde, verlassen wurde (bei einer verschachtelten Funktion kann der Compiler sie sicher so umwandeln, dass dies kein Problem darstellt, bei einer verschachtelten Funktionsvariable ist dies jedoch nicht möglich).
Man könnte argumentieren, dass dasselbe auch für Methodenzeiger und Methodenvariablen gilt, da sie nicht mehr aufrufbar sind, sobald ihre Klasseninstanz freigegeben ist, aber diese sind in der Object Pascal Welt viel häufiger anzutreffen, während verschachtelte Funktionsvariablen nur sehr selten verwendet werden, so dass die Gefahren der ersteren viel offensichtlicher sind als die der letzteren.
Aus diesem Grund ist die Zuweisung von verschachtelten Funktionsvariablen an Funktionsreferenzen verboten.

Anonyme Funktionen

Anonyme Funktionen (oder Anonyme Prozeduren oder Anonyme Routinen, im Folgenden einfach Anonyme Funktionen) sind Routinen, denen kein Name zugeordnet ist und die in der Mitte eines Codeblocks deklariert werden (zum Beispiel auf der rechten Seite eines Ausdrucks oder als Parameter für einen Funktionsaufruf). Sie können jedoch genauso direkt aufgerufen werden wie eine verschachtelte Funktion (oder verschachtelte Prozedur oder verschachtelte Routine).

Anonyme Funktionen werden mit dem Modeschalter `ANONYMOUSFUNCTIONS` aktiviert (in den folgenden Beispielen wird davon ausgegangen, dass dieser Modeschalter vorhanden ist).

Eine anonyme Funktion wird wie folgt deklariert:

```
FUNCTION|PROCEDURE [(argumentlist)][[resultname]: resulttype;] [Direktiven;]
[[VAR|TYPE|CONSTAbschnitt]|[verschachtelte Routine]]*
BEGIN
[STATEMENTS]
END
```

Wie man sieht, sieht eine anonyme Funktion wie eine reguläre Funktion (oder Prozedur oder Routine) aus, mit den wichtigsten Unterschieden, dass sie keinen Namen hat und nicht durch ein Semikolon abgeschlossen wird (weil sie im Wesentlichen ein Ausdruck ist). Da sie in Modi ohne implizite `RESULTvariable` keinen Namen hat, ist es erlaubt, die Ergebnisvariable explizit zu benennen (auch in Modi mit RESULTvariable), wie es bei Operatorüberladungen der Fall ist.

Es ist möglich, eine anonyme Funktion direkt aufzurufen; in diesem Fall verhält sie sich im Wesentlichen wie eine verschachtelte Funktion.

Wie verschachtelte Funktionen haben anonyme Funktionen Zugriff auf die Symbole (Variablen, Funktionen usw.) des umgebenden Bereichs, einschließlich `Self`, wenn der umgebende Bereich eine Methode ist. Der Zugriff auf ein solches Symbol wird als "Capturing" bezeichnet und ist eines der Kernkonzepte anonymer Funktionen.

Ihre Hauptverwendung ist jedoch die Zuweisung an einen der verschiedenen Funktionszeigertypen: Funktionsvariablen, Methodenvariablen, verschachtelte Funktionsvariablen und Funktionsreferenzen. Allerdings kann nicht jede anonyme Funktion jedem Funktionszeigertyp zugewiesen werden, da dies davon abhängt, welche Symbole (wenn überhaupt) aus dem umgebenden Bereich erfasst werden. Anders als bei nicht-anonymen Funktions- oder Methodenbezeichnern erfolgt diese Zuweisung jedoch *immer ohne* den "@"-Operator, da man mit anonymen Funktionen außer dem Aufruf nicht viel anderes machen kann.
Eine anonyme Funktion, die überhaupt keine Symbole einfängt (außer globalen oder statischen Symbolen), kann allen vier Funktionszeigertypen zugewiesen werden. Wenn die anonyme Funktion `Self` einfängt, kann sie nicht mehr Funktionsvariablen zugewiesen werden, aber immer noch den anderen drei. Und wenn sie ein lokales Symbol aufnimmt, kann sie nur verschachtelten Funktionsvariablen oder Funktionsreferenzen zugewiesen werden.
Im Falle von Funktionsvariablen, Methodenvariablen und verschachtelten Funktionsvariablen verhalten sich anonyme Funktionen genauso wie ihre nicht-anonymen Gegenstücke. Die Unterschiede treten auf, wenn sie mit Funktionsreferenzen verwendet werden, was im nächsten Teil beleuchtet wird.

Doch zunächst einige Beispiele:

Code: Pascal [Wählen][+]

```
1. Typ
2.   TFunc = Funktion: LongInt;
3.
4. var
```

```
5.    p: TVorgang;
6.    f: TFunc;
7.    n: TNotifyEvent;
8. beginnen
9.    procedure(const aArg: String)
10.   beginnen
11.     Writeln(aArg);
12.   end('Hallo Welt');
13.
14.   p := Verfahren
15.        beginnen
16.              Writeln('Foobar');
17.           Ende;
18.   p();
19.
20.   n := procedure(aSender: TObject);
21.        beginnen
22.              Writeln(HexStr(Pointer(aSender));
23.           Ende;
24.   n(Nil);
25.
26.   f := Funktion MyRes : LongInt;
27.        beginnen
28.              MyRes := 42;
29.           Ende;
30.   Writeln(f());
31. Ende.
```

Anonyme Funktionsreferenzen

Wie bereits erwähnt, entfalten die beiden neuen Funktionen ihre größte Wirkung, wenn sie miteinander kombiniert werden: Wie eine verschachtelte Funktion kann eine anonyme Funktion auf Symbole aus dem umgebenden Bereich zugreifen, aber im Gegensatz zu verschachtelten Funktionen kann eine anonyme Funktion, die einer Funktionsreferenz zugewiesen wurde, den Bereich *verlassen*, in dem sie deklariert wurde, und nimmt dann die erfassten Symbole mit.
Zu diesem Zweck wird jede Variable oder jeder Parameter, der von einer anonymen Funktion erfasst wird, Teil der implizit erzeugten Objektinstanz (die als undurchsichtig zu betrachten ist), die der Funktionsreferenz zugewiesen wird, anstatt zur ursprünglichen Funktion zu gehören. Die ursprüngliche Funktion referenziert diese Symbole dann unter Verwendung der Objektinstanz anstelle ihres Stackframes. Dies hat zur Folge, dass Änderungen an den Symoblen in allen anonymen Funktionen, die dieses Symbol erfassen, berücksichtigt werden.

Zum Beispiel:

Code: Pascal [Wählen][+]

```
1. Typ
2.    TProc = Verweis auf Prozedur;
3.
4.  Verfahren Test;
5. var
6.    i: LongInt;
7.    p: TProc;
8. beginnen
9.    i := 42;
10.   p := Verfahren
11.        beginnen
12.              Writeln(i);
13.           Ende;
14.
15.   p(); // wird 42 drucken
16.
17.   i := 21;
18.
19.   p(); // druckt 21
20. Ende;
```

Diese Änderungen bleiben auch über Aufrufe und verschiedene anonyme Funktionen hinweg bestehen, solange sie die gleichen Symbole erfassen:

Code: Pascal [Wählen][+]

```
1. Typ
2.   TProc = Verweis auf Prozedur;
3.
4. Verfahren Test;
5. var
6.   i: LongInt;
7.   p1, p2: TProc;
8. beginnen
9.   i := 42;
10.  p1 := Verfahren
11.        beginnen
12.              Writeln(i);
13.                  i := i * 2;
14.           Ende;
15.
16.  p1(); // wird 42 gedruckt
17.
18.  p2 := Verfahren
19.        beginnen
20.                  Writeln(i);
21.               Ende;
22.
23.  p1(); // wird 84 gedruckt
24.  p2(); // wird 168 gedruckt
25. Ende;
```

Die Lebensdauer von verwalteten Typen, die von anonymen Funktionsreferenzen erfasst werden, wird entsprechend gehandhabt (sie bleiben so lange am Leben, wie mindestens eine anonyme Funktion, die sie erfasst hat, ebenfalls am Leben ist), allerdings ist besondere Vorsicht bei der manuellen Speicherverwaltung geboten:

Code: Pascal [Wählen][+]

```
1. Typ
2.   TProc = Verweis auf Prozedur;
3.
4. Funktion Test: TProc;
5. var
6.   o: TObject;
7. beginnen
8.   o := TObject. Erstellen;
9.   Ergebnis := Verfahren
10.           beginnen
11.                       Writeln(o. ClassName);
12.                   Ende;
13.  o. Frei;
14. Ende;
```

Der Aufruf der Funktionsreferenz, die von Test zurückgegeben wird, führt im Wesentlichen zu use-after-free. Und "o" überhaupt nicht freizugeben, führt zu einem Speicherleck.

Kompatibilität

Die beiden Funktionen sind im Großen und Ganzen kompatibel zu Delphi's Anonymen Methoden. FPC erlaubt jedoch die Zuweisung von anonymen Funktionen an verschiedene Funktionszeigertypen, während Delphi sie auf Funktionsreferenzen beschränkt.
Außerdem behandelt FPC die Zuweisung von Funktions-, Methoden- und verschachtelten Funktionsvariablen an Funktionsvariablen etwas anders. Nehmen Sie den folgenden Code:

Code: Pascal [Wählen][+]

```
1. Verfahren Foo;
2. beginnen
3.   Writeln('Foo');
4. Ende;
5.
6. Verfahren Bar;
7. beginnen
8.   Writeln('Bar');
9. Ende;
10.
11. Verfahren Test;
12.       var
13.   p: Verweis auf das Verfahren;
14.   p2: Verfahren;
15.       beginnen
16.   p2 := Foo;
17.   p := p2;
18.   p();
19.   p2 := Bar;
20.   p();
21. Ende;
```

Delphi erzeugt im Wesentlichen Folgendes:

Code: Pascal [Wählen][+]

```
1. Verfahren Test;
2. var
3.   p: Verweis auf das Verfahren;
4.   p2: Verfahren;
5. beginnen
6.   p2 := Foo;
7.   p := Verfahren
8.        beginnen
9.             p2();
10.           Ende;
11.   p();
12.   p2 := Bar;
13.   p();
14. Ende;
```

Das Ergebnis ist die folgende Ausgabe:

Code: [Wählen]

```
Foo
Bar
```

FPC erzeugt jedoch Folgendes:

Code: Pascal [Wählen][+]

```
1. Verfahren Test;
2. var
3.   p: Verweis auf das Verfahren;
4.   p2, tmp: Verfahren;
5. beginnen
6.   p2 := Foo;
7.   tmp := p2;
8.   p := Verfahren
9.        beginnen
10.            tmp();
11.           Ende;
12.   p();
```

```
13.   p2 := Bar;
14.   p();
15. Ende;
```

Das Ergebnis ist die folgende Ausgabe:

Code: [Wählen]

Foo

Foo

Dies ist konsistenter mit der Zuweisung von anderen Funktionszeigertypen zu Funktionszeigertypen.

Das Feature Funktionsreferenzen ist auf allen Plattformen verfügbar, die das Feature Klassen zur Verfügung haben (also im Wesentlichen alles außer AVR) und Anonyme Funktionen selbst sind auf allen Plattformen verfügbar (mit Ausnahme der Zuweisung zu Funktionsreferenzen auf Plattformen, wo diese fehlen). Ja, das schließt Plattformen wie DOS ein, wo Direktiven wie "*far*" und "*near*" entsprechend gehandhabt werden (was bedeutet, dass diese bei der Zuweisung ebenfalls kompatibel sein müssen).

Da diese beiden Features ziemlich kompliziert sind, könnte es noch eine Menge Bugs geben, also bitte ich euch, sie nach Herzenslust zu testen und gefundene Bugs in den Issues auf GitLab zu melden, damit wir so viele wie möglich vor der nächsten Hauptversion (die noch nicht geplant ist, also keine Sorge) beheben können.

Weitere RTL-Verbesserungen wie die Deklaration von *TProc<>* oder das Hinzufügen einer *TThread.Queue()*, die eine Funktionsreferenz entgegennimmt, werden in naher Zukunft kommen, nachdem die Grundlagen auf der Compilerseite fertig sind. Vielleicht können wir jetzt auch Portierungen von Bibliotheken wie Spring4D und OmniThreadLibrary in Angriff nehmen. Es gibt auch die Idee, eine Syntax einzuführen, die steuert, ob Symbole per Referenz (wie derzeit) oder per Wert erfasst werden.

Viel Spaß!