AngelScript と C++の連携を容易に実現する機構

2231084　高橋 裕司　　成見研究室

1　背景

1.1　AngelScript

AngelScript (以下 AS) [1] は、静的なオブジェクト指向のスクリプト言語で、C++でコア部分を実装したプログラムの振る舞いを記述するのに使用される．AS の使用例として、ゲーム開発会社におけるゲームプログラムの実装を挙げることができる．社内で使用するゲームのエンジンを C++のコードとして実装し、ゲームの登場人物の挙動を AS のコードとして実装する．こうすることで、登場人物の挙動を変更する際、AS コードの記述を変更するだけで済み、ゲームエンジンの C++コードを書き換えて再コンパイルする必要がない．これにより、ゲームソフトウェアの開発の効率が向上する．

1.2　AngelScript の仮想機械

AS のコードを扱う C++プログラムは、図 1 のように AS の仮想機械 (以下 VM) オブジェクトを内部に持つ．C++プログラムは、この VM を用いて AS のコードを読み込み、VM において AS の関数を呼び出す．以降 AS の VM オブジェクトを、単に VM と呼ぶ．

VM が実行する AS の関数が C++プログラムの関数やオブジェクトを利用する場合、その窓口となる AS の関数やクラスを、VM に追加する必要がある．以下、AS 側の窓口となる AS の関数やクラスを VM に追加し、C++の関数やオブジェクトを AS の関数から利用できるようにすることを、C++の関数やクラスの登録と呼ぶ．

たとえば、図 2 の C++構造体 `Pos` は平面座標を扱うクラスで、座標の平行移動を行うメンバ関数 `translate` を持つ．このクラスを VM オブジェクト `vm` に登録する C++のコードを、図 3 に示す．ここで、簡単のためコードの一部を簡略化してある．図中、AS における名前は赤字で示している．1 行目から 3 行目で C++のクラス `Pos` を AS のクラス `Pos` として登録し、4 行目から 6 行目で C++のメンバ関数 `translate` を AS の関数 `translate` として登録する．

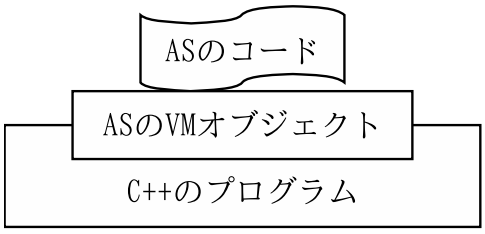

図 1: 振る舞いを AS で実装する C++プログラム

1.3　問題点

図 2 からわかるように、既存の登録処理の記述は煩雑であり、間違いを起こしやすい．その理由は次のとおりである．C++のクラス `Pos` の登録では、その型情報から決まる C++の値と AS のクラス名を C++の登録関数 `RegisterObjectType` に渡す．加えて `translate` の登録では、`translate` の定義と対応する AS の関数宣言を C++の登録関数 `RegisterObjectMethod` に渡す．これらより、各 C++のクラスや関数の登録で、それらの定義と重複する C++の値や AS の記述を用いるため、登録する C++のクラスや関数の数だけ煩雑な登録処理を記述する必要がある．また、AS のクラス名や関数宣言は、登録処理を実装するプログラマが記述するため、AS のクラス名が誤って記入される場合がある．加えて、C++の関数の型を変更した際に、その関数と対応する AS の関数宣言が変更されない場合がある．これらの場合、AS の記述の間違いによって、登録処理が失敗する．

2　目的

C++のクラスや関数を登録する既存の方法では、C++の関数やクラスごとに煩雑な登録処理が必要であるため、登録処理の実装に手間がかかる．また、登録に必要な AS のクラス名や関数宣言を登録処理の実装者が記述するため、AS のクラス名や関数宣言を誤って記述した場合、登録処理が失敗する．

この問題を解決するため本研究では、C++と AS が連携する処理を容易に実現する機構を実装する．

```
1 struct Pos {
2   Pos(int x, int y);
3   ~Pos();
4   void translate(int dx, int dy);
5 };
```

図 2: 平面座標を扱う C++クラス (構造体)Pos の定義

```
1 vm->RegisterObjectType(
2   "Pos", sizeof(Pos),
3   asGetTypeTraits<Pos>() ...);
4 vm->RegisterObjectMethod(
5   "Pos", "void translate(int, int)",
6   asMETHOD(Pos, translate), ...);
```

図 3: Pos とそのメンバ関数の登録処理

3 方針

本研究で実装する機構は、C++の関数やクラスの登録に必要な C++の値や AS の記述を、C++の関数やクラスの定義にある情報を用いて自動的に生成する．このようにする利点は、C++の関数やクラスの定義の変更後に AS の記述を修正する手間を省けることである．

C++の関数やクラスの登録処理の生成は、C++のテンプレートメタプログラミング (以下 TMP) [2] を用いて実現する．TMP は C++の言語機能のみを用いる．また、TMP の実装を容易にするために、既存の TMP ライブラリである Boost の MPL [3] を用いる．

4 テンプレートメタプログラミング

C++の TMP とは、C++のテンプレートを用いてコンパイル時に処理を実行する技法で、C++プログラムのコンパイル時に計算や最適化を行う際に利用される．

本研究で実装する機構は、C++の型に対応する AS の記述を返す TMP の関数 `gen_code_type` を用いる．この関数は、C++のテンプレート構造体を用いて図 4 の通りに定義される．テンプレート引数 `T` で C++の型を受け取り、構造体内で定義される型 `type` が持つ C++の型を返す．1 行目では、`gen_code_type` を表すテンプレート構造体を宣言し、2 行目以降では、`T` が C++の `int`、`void` 型となる場合の `type` の実体を定義する．`type` の実体は、AS の型名文字列を持つ C++のテンプレートクラス `string` である．`string` は、TMP で文字列を扱うためのテンプレートクラスで、コンパイル時に扱われる．

5 本機構の使用例

図 2 の C++のクラス `Pos` とそのメンバ関数の登録処理を、本研究で実装する機構を用いて実装すると、図 5 に示すようになる． 図の 1 行目では `Pos` を `vm` に登録し、2 行目から 3 行目では、テンプレート引数の `Pos` と AS の関数名を用いて、`Pos` のメンバ関数 `translate` を登録する．C++の関数 `RegisterClass`、`RegisterClassMethod` は、本機構が提供する登録関数で、TMP を用いて図 3 の C++関数 `RegisterObjectType`、`RegisterObjectMethod` を内部で呼び出す．このように、本機構が提供する C++の関数を用いることで、図 3 と同等の処理を誤りなく記述することができる．

```
1 template <class T> struct gen_code_type;
2 template <> struct gen_code_type<int> {
3   using type = string<"int">;
4 }
5 template <> struct gen_code_type<void> {
6   using type = string<"void">;
7 }
```

図 4: AS の記述を返すメタ関数 `gen_code_type<T>`

```
1 RegisterClass<Pos>(vm);
2 RegisterClassMethod<Pos, "translate">(
3   vm, &Pos::translate);
```

図 5: 本研究で実装する機構を用いた登録処理の実装例

6 関連研究

Golodetz [4] は、C++の関数の型から AS の関数宣言を生成する機構を提案した．この機構は、C++のテンプレートを用いて C++型に対応する AS の記述を生成する機構を持つが、AS の記述を C++プログラムの実行時に生成する．そのため、C++の関数やクラスの登録時にオーバヘッドが生じる．本研究では、TMP を用いて AS の記述生成を C++プログラムのコンパイル時に行うため、C++の関数やクラスの登録時にオーバヘッドが生じない．

Tanimura ら [5] は、Lua と C の連携を簡潔に実装するためのドメイン特化型言語 LuCa を提案した．LuCa で記述されたコードをコンパイルする際、LuCa のコードを C のコードに変換する必要がある．そのため、LuCa のコードは Tanimura らが実装した C への変換器に依存する．本研究では、TMP を用いて C++と AS の連携を実装するため、連携処理の C++コードは外部ツールに依存しない．

7 現状と今後

現状では、既存の登録処理を、TMP を用いて簡潔に記述できる C++の関数を実装した．

今後は、登録以外の AngelScript と C++の連携処理を簡潔に記述できるようにする．また、実装した機構を既存の登録処理の実装方法と比較し、その性能を評価する．評価では、C++のクラスを登録する処理に必要なコード量と、C++プログラムのコンパイル時間に注目する予定である．

参考文献

[1] Andreas Jönsson. AngelScript - AngelCode.com. https://www.angelcode.com/angelscript/sdk/docs/manual/index.html

[2] Todd Veldhuizen. Template Metaprograms https://www.cs.rpi.edu/~musser/design/blitz/meta-art.html

[3] Aleksey Gurtovoy and David Abrahams. THE BOOST MPL LIBRARY - 1.82.0. https://www.boost.org/doc/libs/1_82_0/libs/mpl/doc/index.html

[4] Stuart Golodetz. “Simplifying the C++/AngelScript Binding Process.”. In *Overload, no. 95*, pp.19–23. Association of C and C++ Users, 2010. https://ora.ox.ac.uk/objects/uuid:6bc7e8c0-f199-4357-8ea3-3e6f224e9de7

[5] Akira Tanimura and Hideya Iwasaki. “Integrating lua into C for embedding lua interpreters in a C application”. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pp.1936–1943. Association for Computing Machinery, 2016. https://dl.acm.org/doi/10.1145/2851613.2851747