セーブデータのためのシリアライズ処理

– 互換性維持とデバッグ効率向上のために –

2014 年 2 月 27 日 初稿

板垣 衛

■ 改訂履歴

稿	改訂日	改訂者	改訂内容
初稿	2014 年 2 月 27 日	板垣 衛	（初稿）

■ 目次

■ 概略

ゲームのセーブデータのためのシリアライズ処理を設計する。
バージョン互換性に強く、かつ、生産性の高いセーブデータ書き込み／読み込みシステムを構築する。

ゲームタイトル固有のセーブデータ構造に特化したプログラミングは必要とするが、その手間を極力抑えるための仕組みを策定する。

■ 目的

本書は、柔軟で生産性の高いシリアライズ処理を作ることにより、ゲーム制作の効率を向上させることを目的とする。

ゲーム制作の過程でセーブデータの構造が変わるのは当然としても、その互換性が保証されることにより、セーブデータ依存の制作作業や QA 作業に支障をきたさずに済む。本書のシステムはそのための汎用処理であり、また、プログラマーの手間もできる限り簡略化する。
さらには、このシステムを通して、作業用のセーブデータの量産を簡単に行えるようにすることも目的とする。

■ 用語の規定とシリアライズの基礎

本書の構成にあたって、まずは用語を規定する。
これにより、本書の前提となるシリアライズの基礎も説明する。
一般的なシリアライズ関係の用語の説明から、本書独自の用語の規定までを含む。

▼ シリアライズ（Serialize）

「シリアライズ」とは、メモリ上のデータを、「後で復元すること」を目的に、保存可能な形式に変換することである。
シリアライズの主な用途は、「ファイルへの保存」、「データ通信」である。
なお、日本語では「直列化」と訳される。また、（主にファイルに保存する意味で）「永続化」と呼ばれることも多い。

本書においては、「ゲームのセーブデータを作成すること」をシリアライズと呼ぶ。
ファイルに直接書き込むことまでを意味せず、ファイルに書き込み可能なイメージをメモリ上に作成することを指すものとする。

▼ デシリアライズ（Deserialize）

「デシリアライズ」とは、シリアライズされたデータをメモリ上に復元することである。
「アンシリアライズ」（Unserialize）と呼ばれることもある。

本書においては、「ゲームのセーブデータからゲームデータを復元すること」をデシリアライズと呼ぶ。
ファイルから直接読み込むことまでを意味せず、メモリ上のファイルイメージを元に復元することを指すものとする。

なお、デシリアライズの際は、純粋にセーブ時の状態を再現するのではなく、可能な限り、現在の構造に適合する形で復元する、バージョン互換維持処理を伴う。

▼ セーブデータ

ゲームの進行状態が記録されたファイルのこと。
シリアライズによって作成されたデータのことであり、文脈によってはメモリ上のファイルイメージを意味するが、基本的には保存されたファイルのことである。

▼ セーブ（Save）

セーブデータを作成すること。
シリアライズからファイルへの保存をひとまとめにした処理を意味する。

▼ ロード（Load）

セーブデータを読み込んで、ゲームの進行状態を復元すること。
ファイルの読み込みからデシリアライズをひとまとめにした処理を意味する。

▼ アーカイブ（Archive）

Boost C++ ライブラリの boost::serialization クラスに基づいてこの用語を用いる。

本来の用語としては、データを「貯蔵」することであり、複数のファイルを一つにまとめるようなことを意味するが、本書においてはシリアライズしたデータの「保存形式」を意味する。

「保存形式」には、大きく分けて「バイナリ」と「テキスト」があり、「バイナリアーカイブ」や「テキストアーカイブ」などと呼ぶ。なお、boost::serialization では標準的に「XML アーカイブ」も扱う。

「アーカイブ」という言葉は基本的には名詞であり、文脈によっては「ファイル」と同義としても用いられるが、「アーカイブする」「アーカイブしたデータ」のように動詞としても用いられる。

▼ シリアライズとアーカイブの違い

処理として「シリアライズ」と「アーカイブ」は厳密には異なる。

シリアライズは保存する情報（とその順序）の指定であり、実際に保存を行うのはアーカイブである。

boost::serializetion では、シリアライズがユーザー定義処理で、アーカイブが汎用処理である。

▼ ギャザーラ（Gatherer）

セーブデータを構成するために、分散するデータを収集する処理を担う仕組みを意味する。また、「収集」（動詞）の意味では「ギャザー」（Gather）という用語も用いる。

これは、本書が独自に規定する用語である。セーブデータ処理要件に柔軟に対応するための仕組みとして導入する。

なお、これはあくまでもシリアライズの対象データを集めることを意味するものであり、boost::serialization の「非侵入型（non-intrusive）シリアライズ」のような意味ではない。（解説：boost::serialization は、シリアライズ対象クラス本体にシリアライズ処理を記述するのが基本であるが、外部に記述することもできる。標準ライブラリが提供するクラスなど、直接シリアライズ処理を記述できないものをシリアライズするために用意されている仕組み。）

▼ ディストリビュータ（Distributor）

セーブデータの復元のために、読み込んだデータを各所に分散配置する処理を担う仕組

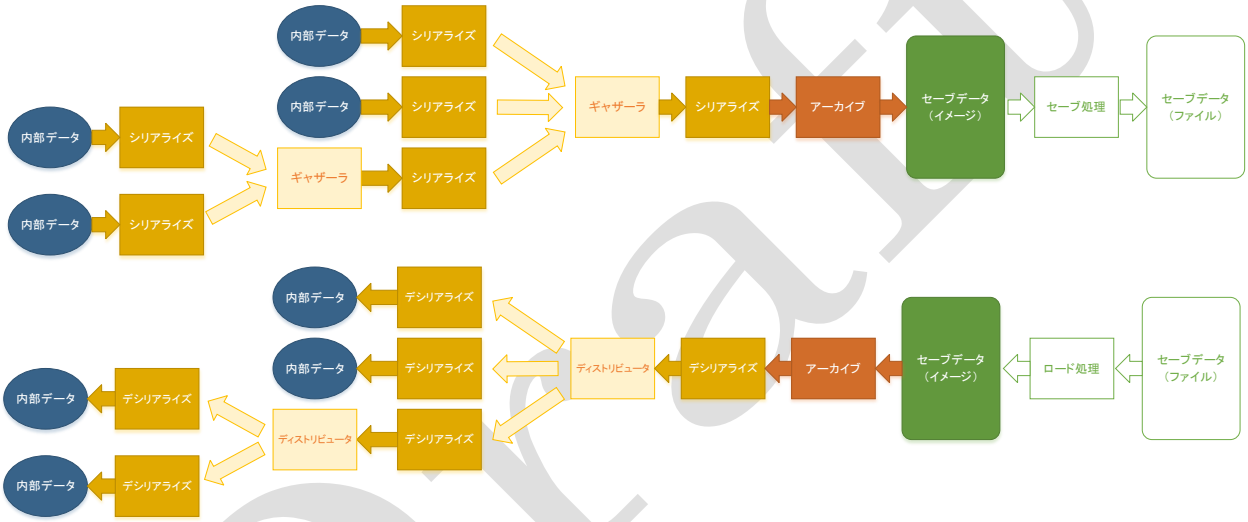
みを意味する。また、「分配」（動詞）の意味では「ディストリビュート」（Distribute）という用語も用いる。

これは、本書が独自に規定する用語である。セーブデータ処理要件に柔軟に対応するための仕組みとして導入する。

▼ 各用語の関係と処理イメージ

上記の各用語の関係を、処理イメージにあてはめると下記のとおりとなる。

用語の関係と基本処理イメージ：



「ギャザーラ」と「ディストリビュータ」により、boost::serialization などの一般的なシリアライズ処理と異なり、直接関連性のない複数のデータをひとまとめに扱うようにする。これにより、ゲームプログラムの自由度とセーブデータの自由度の両方を獲得する。

なお、本書では、セーブデータファイルについてはとくに扱わない。

■ 【参考】Boost C++ ライブラリの boost::serialization

本システムの要件を定義する前に、本システムの参考とする Boost C++ライブラリの boost::serialization を簡単に説明する。

なお、直接 boost::serialization を使用しないのは、「ギャザーラ」と「ディストリビュータ」のような仕組みを導入して、柔軟な処理要件に対応するためである。独自実装することにより、後述する「部分的なロード」のような特殊な処理にも対応し、デバッグ効率を高

めるようにする。

▼ Boost C++ライブラリの準備

Boost C++ライブラリは、ヘッダーをインクルードするだけでそのまま使用できるテンプレートクラスを多く含むが、一部のライブラリはプラットフォーム向けのビルドが必要である。boost::serialization もそのようなライブラリの一つである。

なお、ビルド方法の説明については省略する。

▼ シリアライズのサンプル①：基本的な動作

実際に boost::serialization を使用したサンプルを示す。

単純なクラスをシリアライズしてテキストアーカイブに保存し、またデシリアライズする。「serialize」テンプレート関数が、シリアライズとデシリアライズの処理で共用される。

基本的なシリアライズとデシリアライズのサンプル：

【インクルード】

```
#include <fstream>//ファイル入出力ストリーム
#include <boost/serialization/serialization.hpp>//シリアライズ
#include <boost/archive/text_oarchive.hpp>//テキスト出力アーカイブ
#include <boost/archive/text_iarchive.hpp>//テキスト入力アーカイブ
```

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス１
class CTest1
{
public:
	//アクセッサ
	int getData1() const { return m_data1; }//データ１取得
	void setData1(const int value){ m_data1 = value; }//データ１更新
	float getData2() const { return m_data2; }//データ２取得
	void setData2(const float value){ m_data2 = value; }//データ２更新
	char getData3(const int index) const { return m_data3[index]; }//データ３取得
	void setData3(const int index, const int value){ m_data3[index] = value; }//データ３更新
private:
	//シリアライズ用テンプレート関数
	//※シリアライズとデシリアライズの共通処理
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ１ ※アーカイブに & 演算子でデータを渡す
		arc & m_data2 & m_data3;//データ２ & データ３ ※連続してデータを渡すことも可能
	}
private:
	//フィールド
	int m_data1;//データ１：int 型
	float m_data2;//データ２：float 型
	char m_data3[3];//データ３：配列型
};
```

```
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest1, 99);//バージョン設定
BOOST_CLASS_TRACKING(CTest1, boost::serialization::track_never);//VC++でエラー C4308 を回避するための設定
```

【シリアライズ】

```
//--------------------
//シリアライズテスト１：テキストアーカイブ
void serializeTest1()
{
	printf("--------------------¥n");
	printf("シリアライズ：テキストアーカイブ¥n");
	//オブジェクト生成
	CTest1 obj;
	//テストデータをセット
	obj.setData1(123);
	obj.setData2(4.56f);
	obj.setData3(0, 7);
	obj.setData3(1, 8);
	obj.setData3(2, 9);
	//実行前の状態表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
	//シリアライズ
	std::ofstream stream("test1.txt");//ファイル出力ストリーム生成
	boost::archive::text_oarchive arc(stream);//出力アーカイブ生成：テキストアーカイブ
	arc << obj;//シリアライズ
	stream.close();//ストリームをクローズ
}
```

【デシリアライズ】

```
//--------------------
//デシリアライズテスト１：テキストアーカイブ
void deserializeTest1()
{
	printf("--------------------¥n");
	printf("デシリアライズ：テキストアーカイブ¥n");
	//オブジェクト生成
	CTest1 obj;
	//デシリアライズ
	std::ifstream stream("test1.txt");//ファイル入力ストリーム生成
	boost::archive::text_iarchive arc(stream);//入力アーカイブ生成：テキストアーカイブ
	arc >> obj;//デシリアライズ
	stream.close();//ストリームをクローズ
	//結果を表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト１：テキストアーカイブ
void test1()
{
	//シリアライズ
	serializeTest1();
	//デシリアライズ
	deserializeTest1();
}
```

↓（実行結果）

```
--------------------
シリアライズ：テキストアーカイブ
```

```
data1=123
data2=4.56
data3={7, 8, 9}
serialize:version=99    ←シリアライズ実行（バージョンはクラスに設定されているもの）
--------------------
デシリアライズ：テキストアーカイブ
serialize:version=99    ←デシリアライズ実行　※シリアライズと同じ関数（バージョンはファイルから読み込まれたもの）
data1=123               ←正しくデータが復元されている
data2=4.56              ←（同上）
data3={7, 8, 9}         ←（同上）
```

【test1.txt】※シリアライズの結果、出力されたファイル

```
22 serialization::archive 10 0 99 123 4.5599999 3 7 8 9
```

▼ シリアライズ用テンプレート関数の仕組み

シリアライズ用テンプレート関数は、シリアライズ処理とデシリアライズ処理の両方に兼用する。このテンプレート関数は、与えられたアーカイブクラスによって実体化されるため、アーカイブクラスの「operator&()」の実装に応じて、読み込み処理にも書き込み処理にも対応することができる。

様々な型に対応するために、このオペレータ自体もテンプレート関数となっている。

ライブラリのコードを一部抜粋：

【boost/archive/detail/interface_oarchive.hpp】※一部抜粋

```
template<class Archive>
class interface_oarchive
{
    …（略）…
    // the << operator
    template<class T>
    Archive & operator<<(T & t){
        this->This()->save_override(t, 0);
        return * this->This();
    }
    // the & operator
    template<class T>
    Archive & operator&(T & t){
        return *(this->This()) << const_cast<const T &>(t);
    }
};
```

【boost/archive/detail/interface_iarchive.hpp】※一部抜粋

```
template<class Archive>
class interface_iarchive
{
    …（略）…
    // the >> operator
    template<class T>
    Archive & operator>>(T & t){
        this->This()->load_override(t, 0);
        return * this->This();
    }
    // the & operator
    template<class T>
    Archive & operator&(T & t){
        return *(this->This()) >> t;
    }
};
```

▼ シリアライズのサンプル②：バイナリアーカイブに変更

サンプル①をバイナリアーカイブに変更したサンプルを示す。
シリアライズ対象クラスには変更なし。

バイナリアーカイブを使用したシリアライズとデシリアライズのサンプル：

【インクルード】

```
#include <boost/archive/text_oarchive.hpp>//バイナリ出力アーカイブ
#include <boost/archive/text_iarchive.hpp>//バイナリ入力アーカイブ
```

【シリアライズ】

```
//--------------------
//シリアライズテスト２：バイナリアーカイブ
void serializeTest2()
{
	printf("--------------------¥n");
	printf("シリアライズ：バイナリアーカイブ¥n");
	…（略：テスト１と同じ）…
	std::ofstream stream("test2.bin");//ファイル出力ストリーム生成
	boost::archive::binary_oarchive arc(stream);//出力アーカイブ生成：バイナリアーカイブ
	…（略：テスト１と同じ）…
}
```

【デシリアライズ】

```
//--------------------
//デシリアライズテスト２：バイナリアーカイブ
void deserializeTest2()
{
	printf("--------------------¥n");
	printf("デシリアライズ：バイナリアーカイブ¥n");
	…（略：テスト１と同じ）…
	std::ifstream stream("test2.bin");//ファイル入力ストリーム生成
	boost::archive::binary_iarchive arc(stream);//入力アーカイブ生成：バイナリアーカイブ
	…（略：テスト１と同じ）…
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト２：バイナリアーカイブ
void test2()
{
	//シリアライズ
	serializeTest2();
	//デシリアライズ
	deserializeTest2();
}
```

↓（実行結果）　※テキストアーカイブと同じ結果

```
--------------------
シリアライズ：バイナリアーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
serialize:version=99
--------------------
デシリアライズ：バイナリアーカイブ
serialize:version=99
data1=123
data2=4.56
data3={7, 8, 9}
```

【test2.bin】※シリアライズの結果、出力されたファイル

```
00000000  16 00 00 00 73 65 72 69  61 6C 69 7A 61 74 69 6F  ....serializatio
00000010  6E 3A 3A 61 72 63 68 69  76 65 0D 0A 00 04 04 04  n::archive......
00000020  08 01 00 00 00 00 63 00  00 00 7B 00 00 00 85 EB  ......c...{.....
00000030  91 40 03 00 00 00 07 08  09                       .@.......
```

▼ シリアライズのサンプル③：XML アーカイブに変更

サンプル①を XML アーカイブに変更したサンプルを示す。

XML はシリアライズの際に「データ名」が必要なため、シリアライズ処理にも変更を加える。「BOOST_SERIALIZABLE_NPV()」というマクロを使用してデータ項目を指定することで、データ名も自動的に指定される。

XML アーカイブを使用したシリアライズとデシリアライズのサンプル：

【インクルード】

```
#include <boost/archive/xml_oarchive.hpp>//XML 出力アーカイブ
#include <boost/archive/xml_iarchive.hpp>//XML 入力アーカイブ
```

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス３
class CTest3
{
	…（略：テスト１と同じ）…
	void serialize(Archive& arc, const unsigned int version)
	{
		…（略：テスト１と同じ）…
		arc & BOOST_SERIALIZATION_NVP(m_data1);//データ１ ※アーカイブに & 演算子でデータを渡す
		arc & BOOST_SERIALIZATION_NVP(m_data2) & BOOST_SERIALIZATION_NVP(m_data3);
                                                    //データ２& データ３ ※連続してデータを渡すことも可能
	}
	…（略：テスト１と同じ）…
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest3, 99);//バージョン設定
```

【シリアライズ】

```
//--------------------
//シリアライズテスト３：XML アーカイブ
void serializeTest3()
{
	printf("--------------------¥n");
	printf("シリアライズ：XML アーカイブ¥n");
	…（略：テスト１と同じ）…
	std::ofstream stream("test3.xml");//ファイル出力ストリーム生成
	boost::archive::xml_oarchive arc(stream);//出力アーカイブ生成：XML アーカイブ
	arc << BOOST_SERIALIZATION_NVP(obj);//シリアライズ
	…（略：テスト１と同じ）…
}
```

【デシリアライズ】

```
//--------------------
//デシリアライズテスト３：XML アーカイブ
void deserializeTest3()
{
	printf("--------------------¥n");
	printf("デシリアライズ：XML アーカイブ¥n");
	…（略：テスト１と同じ）…
```

```
	std::ifstream stream("test3.xml");//ファイル入力ストリーム生成
	boost::archive::xml_iarchive arc(stream);//入力アーカイブ生成：XML アーカイブ
	arc >> BOOST_SERIALIZATION_NVP(obj);//デシリアライズ
	…（略：テスト１と同じ）…
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト３：XML アーカイブ
void test3()
{
	//シリアライズ
	serializeTest3();
	//デシリアライズ
	deserializeTest3();
}
```

↓（実行結果）　※テキストアーカイブと同じ結果

```
--------------------
シリアライズ：XML アーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
serialize:version=99
--------------------
デシリアライズ：XML アーカイブ
serialize:version=99
data1=123
data2=4.56
data3={7, 8, 9}
```

【test3.xml】※シリアライズの結果、出力されたファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<obj class_id="0" tracking_level="0" version="99">	←変数名がそのまま XML の項目名になっている
	<m_data1>123</m_data1>			←メンバー名がそのまま XML の項目名になっている
	<m_data2>4.5599999</m_data2>
	<m_data3>
		<count>3</count>
		<item>7</item>
		<item>8</item>
		<item>9</item>
	</m_data3>
</obj>
```

▼ シリアライズのサンプル④：複雑な構造のデータ

シリアライズ対象メンバーに構造体を含む場合、全ての構造体に「serialize」テンプレート関数を実装することで連鎖的にシリアライズできる。

また、ポインタ変数は復元時に自動的にメモリ確保される。

さらに、継承したクラスの場合、子クラスのシリアライズ関数の中から親クラスのシリアライズ関数を呼び出すことも可能。（継承のサンプルは省略する）

複雑な構造のデータのシリアライズとデシリアライズのサンプル：

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス４
```

```
//内部クラス
class CTest4sub
{
public:
	//アクセッサ
	int getVal1() const { return m_val1; }//値 1 取得
	void setVal1(const int value){ m_val1 = value; }//値 1 更新
	int getVal2() const { return m_val2; }//値 2 取得
	void setVal2(const int value){ m_val2 = value; }//値 2 更新
private:
	//シリアライズ用テンプレート関数
	//※シリアライズとデシリアライズの共通処理
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("sub::serialize:version=%d¥n", version);//バージョンを表示
		arc & BOOST_SERIALIZATION_NVP(m_val1)//値 1
		    & BOOST_SERIALIZATION_NVP(m_val2);//値 2
	}
private:
	//フィールド
	int m_val1;//値 1
	int m_val2;//値 2
};
//データクラス
class CTest4
{
public:
	//アクセッサ
	int getData1() const { return m_data1; }//データ 1 取得
	void setData1(const int value){ m_data1 = value; }//データ 1 更新
	float getData2() const { return m_data2; }//データ 2 取得
	void setData2(const float value){ m_data2 = value; }//データ 2 更新
	char getData3(const int index) const { return m_data3[index]; }//データ 3 取得
	void setData3(const int index, const int value){ m_data3[index] = value; }//データ 3 更新
	CTest4sub& getData4(){ return m_data4; }//データ 4 取得
	CTest4sub& getData5(const int index){ return m_data5[index]; }//データ 5 取得
	CTest4sub* getData6(){ return m_data6; }//データ 6 取得
	void setData6(CTest4sub* obj){ m_data6 = obj; }//データ 6 更新
	CTest4sub* getData7(){ return m_data7; }//データ 7 取得
	void setData7(CTest4sub* obj){ m_data7 = obj; }//データ 7 更新
private:
	//シリアライズ用テンプレート関数
	//※シリアライズとデシリアライズの共通処理
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & BOOST_SERIALIZATION_NVP(m_data1)//データ 1
		    & BOOST_SERIALIZATION_NVP(m_data2)//データ 2
		    & BOOST_SERIALIZATION_NVP(m_data3)//データ 3
		    & BOOST_SERIALIZATION_NVP(m_data4)//データ 4
		    & BOOST_SERIALIZATION_NVP(m_data5)//データ 5
		    & BOOST_SERIALIZATION_NVP(m_data6)//データ 6
		    & BOOST_SERIALIZATION_NVP(m_data7);//データ 7
	}
private:
	//フィールド
	int m_data1;//データ 1：int 型
	float m_data2;//データ 2：float 型
	char m_data3[3];//データ 3：配列型
```

```
	CTest4sub m_data4;//データ４：構造体
	CTest4sub m_data5[2];//データ５：構造体の配列
	CTest4sub* m_data6;//データ６：構造体のポインタ
	CTest4sub* m_data7;//データ７：構造体のポインタ（ヌル時の動作確認用）
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest4, 99);//バージョン設定
BOOST_CLASS_VERSION(CTest4sub, 100);//バージョン設定
```

【シリアライズ】※扱うデータ項目が増えている以外はとくに変わった処理はなし

```
//--------------------
//シリアライズテスト４：複雑な構造のデータ（XML アーカイブ）
void serializeTest4()
{
	printf("--------------------¥n");
	printf("シリアライズ：複雑な構造のデータ（XML アーカイブ）¥n");
	//オブジェクト生成
	CTest4 obj;
	//テストデータをセット
	obj.setData1(123);
	obj.setData2(4.56f);
	obj.setData3(0, 7);
	obj.setData3(1, 8);
	obj.setData3(2, 9);
	obj.getData4().setVal1(10);
	obj.getData4().setVal2(11);
	obj.getData5(0).setVal1(12);
	obj.getData5(0).setVal2(13);
	obj.getData5(1).setVal1(14);
	obj.getData5(1).setVal2(15);
	obj.setData6(new CTest4sub());//ポインタ変数にはメモリ確保したオブジェクトをセット
	obj.getData6()->setVal1(16);
	obj.getData6()->setVal2(17);
	obj.setData7(nullptr);//ヌルポインタをセット
	//実行前の状態表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
	printf("data4={val1=%d, val2=%d}¥n", obj.getData4().getVal1(), obj.getData4().getVal2());
	printf("data5[0]={val1=%d, val2=%d}¥n", obj.getData5(0).getVal1(), obj.getData5(0).getVal2());
	printf("data5[1]={val1=%d, val2=%d}¥n", obj.getData5(1).getVal1(), obj.getData5(1).getVal2());
	printf("data6={val1=%d, val2=%d}¥n", obj.getData6()->getVal1(), obj.getData6()->getVal2());
	printf("data7=0x%p¥n", obj.getData7());
	//シリアライズ
	std::ofstream stream("test4.xml");//ファイル出力ストリーム生成
	boost::archive::xml_oarchive arc(stream);//出力アーカイブ生成：XML アーカイブ
	arc << BOOST_SERIALIZATION_NVP(obj);//シリアライズ
	stream.close();//ストリームをクローズ
}
```

【デシリアライズ】※扱うデータ項目が増えている以外はとくに変わった処理はなし

```
//--------------------
//デシリアライズテスト４：複雑な構造のデータ（XML アーカイブ）
void deserializeTest4()
{
	printf("--------------------¥n");
	printf("デシリアライズ：複雑な構造のデータ（XML アーカイブ）¥n");
	//オブジェクト生成
	CTest4 obj;
	//デシリアライズ
	std::ifstream stream("test4.xml");//ファイル入力ストリーム生成
	boost::archive::xml_iarchive arc(stream);//入力アーカイブ生成：XML アーカイブ
	arc >> BOOST_SERIALIZATION_NVP(obj);//デシリアライズ
	stream.close();//ストリームをクローズ
```

```
	//結果を表示
	printf("data1=%d¥n", obj.getData1());
	printf("data2=%.2f¥n", obj.getData2());
	printf("data3={%d, %d, %d}¥n", obj.getData3(0), obj.getData3(1), obj.getData3(2));
	printf("data4={val1=%d, val2=%d}¥n", obj.getData4().getVal1(), obj.getData4().getVal2());
	printf("data5[0]={val1=%d, val2=%d}¥n", obj.getData5(0).getVal1(), obj.getData5(0).getVal2());
	printf("data5[1]={val1=%d, val2=%d}¥n", obj.getData5(1).getVal1(), obj.getData5(1).getVal2());
	printf("data6={val1=%d, val2=%d}¥n", obj.getData6()->getVal1(), obj.getData6()->getVal2());
	printf("data7=0x%p¥n", obj.getData7());
}
```

【テスト】

```
//--------------------
//シリアライズ＆デシリアライズテスト４：複雑な構造のデータ（XML アーカイブ）
void test4()
{
	//シリアライズ
	serializeTest4();
	//デシリアライズ
	deserializeTest4();
}
```

↓（実行結果）

```
--------------------
シリアライズ：複雑な構造のデータ（XML アーカイブ）
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}
data5[0]={val1=12, val2=13}
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}
data7=0x00000000
serialize:version=99
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
--------------------
デシリアライズ：複雑な構造のデータ（XML アーカイブ）
serialize:version=99
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
sub::serialize:version=100
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}	←内部のオブジェクトもきちんと復元している
data5[0]={val1=12, val2=13}	←配列も大丈夫
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}	←ポインタも大丈夫（自動的に new されている）
data7=0x00000000	←ヌルポインタも再現
```

【test4.xml】※シリアライズの結果、出力されたファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<obj class_id="0" tracking_level="0" version="99">
	<m_data1>123</m_data1>
	<m_data2>4.5599999</m_data2>
	<m_data3>
		<count>3</count>
		<item>7</item>
		<item>8</item>
```

```
            <item>9</item>
      </m_data3>
      <m_data4 class_id="1" tracking_level="1" version="100" object_id="_0">
            <m_val1>10</m_val1>
            <m_val2>11</m_val2>
      </m_data4>
      <m_data5>
            <count>2</count>
            <item object_id="_1">
                  <m_val1>12</m_val1>
                  <m_val2>13</m_val2>
            </item>
            <item object_id="_2">
                  <m_val1>14</m_val1>
                  <m_val2>15</m_val2>
            </item>
      </m_data5>
      <m_data6 class_id_reference="1" object_id="_3">
            <m_val1>16</m_val1>
            <m_val2>17</m_val2>
      </m_data6>
      <m_data7 class_id="-1"></m_data7>
</obj>
```

▼ シリアライズのサンプル⑤：非侵入型（non-intrusive）

上記のように、内部で保持するオブジェクトをシリアライズするためには、手間をかけてそれぞれのクラスにシリアライズ処理を実装しなければならない。しかし、これがライブラリのクラスであった場合、独自にシリアライズ処理を追加することができない。このような場合、シリアライズ処理をクラス内の関数ではなく、外部関数として実装することで対応できる。これを「非侵入型」（non-intrusive）と呼ぶ。

非侵入型シリアライズ関数のプロトタイプ：

```
namespace boost{
      namespace serialization{//このネームスペースである必要がある（オーバーロード関数扱いになり自動的に呼び出される）
            template <class Archive>
            void serialize(Archive& ar, TYPE& obj, const unsigned int version)
            {
                  ar & obj.***
                     & obj.***
                     & obj.***;
            }
      }
}
```

サンプル④の「CTest4sub」を非侵入型に変更したサンプル：

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス４
//内部クラス
class CTest4sub
{
      …（略）…
private:
#if 0//元の処理を無効化
      //シリアライズ用テンプレート関数
      //※シリアライズとデシリアライズの共通処理
```

```
	//※private スコープにした上でフレンドクラスを設定する
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("sub::serialize:version=%d¥n", version);//バージョンを表示
		arc & BOOST_SERIALIZATION_NVP(m_val1)//値 1
		    & BOOST_SERIALIZATION_NVP(m_val2);//値 2
	}
private:
#else
public://外部関数から扱えるのはパブリックなメンバーに限られる
#endif
	//フィールド
	int m_val1;//値 1
	int m_val2;//値 2
};
```

【非侵入型シリアライズ関数】

```
namespace boost{
	namespace serialization{
		template <class Archive>
		void serialize(Archive& ar, CTest4sub& sub, const unsigned int version)
		{
			printf("[non-intrusive]sub::serialize:version=%d¥n", version);//バージョンを表示
			arc & BOOST_SERIALIZATION_NVP(m_val1)//値 1
			    & BOOST_SERIALIZATION_NVP(m_val2);//値 2
		}
	}
}
```

↓（実行結果）

```
--------------------
シリアライズ：複雑な構造のデータ（XML アーカイブ）
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}
data5[0]={val1=12, val2=13}
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}
data7=0x00000000
serialize:version=99
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
--------------------
デシリアライズ：複雑な構造のデータ（XML アーカイブ）
serialize:version=99
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
[non-intrusive]sub::serialize:version=100
data1=123
data2=4.56
data3={7, 8, 9}
data4={val1=10, val2=11}
data5[0]={val1=12, val2=13}
data5[1]={val1=14, val2=15}
data6={val1=16, val2=17}
data7=0x00000000
```

【test4.xml】※シリアライズの結果、出力されたファイル

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
```

```
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<obj class_id="0" tracking_level="0" version="99">
	<m_data1>123</m_data1>
	<m_data2>4.5599999</m_data2>
	<m_data3>
		<count>3</count>
		<item>7</item>
		<item>8</item>
		<item>9</item>
	</m_data3>
	<m_data4 class_id="1" tracking_level="1" version="100" object_id="_0">
		<sub.m_val1>10</sub.m_val1>
		<sub.m_val2>11</sub.m_val2>
	</m_data4>
	<m_data5>
		<count>2</count>
		<item object_id="_1">
			<sub.m_val1>12</sub.m_val1>
			<sub.m_val2>13</sub.m_val2>
		</item>
		<item object_id="_2">
			<sub.m_val1>14</sub.m_val1>
			<sub.m_val2>15</sub.m_val2>
		</item>
	</m_data5>
	<m_data6 class_id_reference="1" object_id="_3">
		<sub.m_val1>16</sub.m_val1>
		<sub.m_val2>17</sub.m_val2>
	</m_data6>
	<m_data7 class_id="-1"></m_data7>
</obj>
```

▼ 標準ライブラリのクラスのシリアライズ

「std::string」や「std::vector」などの標準ライブラリのデータをシリアライズすために、Boost C++ライブラリはそれらのシリアライズ用処理を多数用意している。下記のファイルをインクルードすれば使える。

標準ライブラリ用シリアライズ処理のインクルード：※主な例

```
#include <boost/serialization/string.hpp>//std::string 用
#include <boost/serialization/vector.hpp>//std::vector 用
#include <boost/serialization/list.hpp>//std::list 用
#include <boost/serialization/map.hpp>//std::map 用
#include <boost/serialization/set.hpp>//std::set 用
#include <boost/serialization/deque.hpp>//std::deque 用
```

▼ シリアライズのサンプル⑥：バージョン互換処理

例えば、新しくデータ項目が追加された後で、その項目が存在しない古いデータを読み込む場合がある。

そのようなデータを読み込むためのバージョン互換処理は、シリアライズ用テンプレート関数に渡ってくるバージョンをチェックして、処理を振り分ければよい。

シリアライズテンプレート関数のバージョン互換処理サンプル：

【シリアライズ対象クラス】

```
//--------------------
//シリアライズ対象クラス１
class CTest1
{
	…（略）…
	//シリアライズ用テンプレート関数
	friend class boost::serialization::access;
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ１
		arc & m_data2 & m_data3;//データ２ & データ３
		if(version >= 101)
		{
			//Ver.101 以上の場合
			arc & m_data4;//データ４
		}
		else
		{
			//Ver.101 以前の場合
			//※古いバージョンが渡ってくるのはデシリアライズの時しかありえないので、
			//  直接値を更新してしまっても問題ない
			m_data4 = 99;//データ４
		}
	}
private:
	//フィールド
	int m_data1;//データ１：int 型
	float m_data2;//データ２：float 型
	char m_data3[3];//データ３：配列型
	int m_data4;//データ４　※Ver.101 より追加
};
//--------------------
//シリアライズ設定
BOOST_CLASS_VERSION(CTest1, 101);//バージョン設定
```

▼ シリアライズのサンプル⑦：セーブ処理とロード処理を分ける

複雑なバージョン互換処理が必要な場合、セーブ処理とロード処理を分けて扱うことですっきりした処理構造にすることができる。セーブ処理にはバージョン互換処理が必要ないためである。

「BOOST_SERIALIZATION_SPLIT_MEMBER()」というマクロをクラス内に配置すると、セーブ処理とロード処理を分けて扱うクラスとなる。その場合、「serialize()」関数の代わりに「save()」関数と「save()」関数を定義する。

基本的なシリアライズとデシリアライズのサンプル：

```
#include <boost/serialization/split_member.hpp>//BOOST_SERIALIZATION_SPLIT_MEMBER 用　※このインクルードが必要

//--------------------
//シリアライズ対象クラス１
class CTest1
{
	…（略）…
```

```
	//シリアライズ用テンプレート関数
	friend class boost::serialization::access;
#if 0
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		printf("serialize:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ 1
		arc & m_data2 & m_data3;//データ 2 & データ 3
	}
#else
	BOOST_SERIALIZATION_SPLIT_MEMBER();
	template<class Archive>
	void save(Archive& arc, const unsigned int version) const
	{
		printf("save:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ 1
		arc & m_data2 & m_data3;//データ 2 & データ 3
	}
	template<class Archive>
	void load(Archive& arc, const unsigned int version)
	{
		printf("load:version=%d¥n", version);//バージョンを表示
		arc & m_data1;//データ 1
		arc & m_data2 & m_data3;//データ 2 & データ 3
		if(version >= 101)
		{
			//Ver.101 以上の場合
			arc & m_data4;//データ 4
		}
		else
		{
			//Ver.101 以前の場合
			m_data4 = 99;//データ 4
		}
	}
#endif
	…（略）…
```

↓（実行結果）

```
--------------------
シリアライズ：テキストアーカイブ
data1=123
data2=4.56
data3={7, 8, 9}
save:version=99
--------------------
デシリアライズ：テキストアーカイブ
load:version=99
data1=123
data2=4.56
data3={7, 8, 9}
```

■ 要件定義

以上を踏まえて、ゲームのセーブデータのためのシリアライズ処理に対する要件を定義する。

▼ 基本要件

本書が扱うシステムの基本要件は下記のとおり。

・ する。

▼ 要求仕様／要件定義

以下、本書が扱うシステムの要件を定義する。なお、要件として不確定の要求仕様も併記する。

・

■ 処理仕様

■■以上■■

■ 索引

索引項目が見つかりません。

セーブデータのためのシリアライズ処理

以　上