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.....1

■ 目的.....1

■ 用語の規定とシリアライズの基礎.....1

▼ シリアライズ（Serialize）..... 1
▼ デシリアライズ（Deserialize）..... 2
▼ セーブデータ..... 2
▼ セーブ（Save）..... 2
▼ ロード（Load）..... 2
▼ アーカイブ（Archive）..... 3
▼ シリアライズとアーカイブの違い..... 3
▼ コレクター（Collector）..... 3
▼ ディストリビュータ（Distributor）..... 4
▼ 各用語の関係と処理イメージ..... 4

■ 【参考】Boost C++ ライブラリの boost::serialization.....4

▼ Boost C++ライブラリの準備..... 5
▼ シリアライズのサンプル①：基本的な動作..... 5
▼ シリアライズ用テンプレート関数の仕組み..... 7
▼ シリアライズのサンプル②：バイナリアーカイブに変更..... 8
▼ シリアライズのサンプル③：XML アーカイブに変更..... 9
▼ シリアライズのサンプル④：複雑な構造のデータ.....10
▼ シリアライズのサンプル⑤：非侵入型（non-intrusive）.....14
▼ 標準ライブラリのクラスのシリアライズ.....16
▼ シリアライズのサンプル⑥：バージョン互換処理.....16
▼ 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.....17

■ 要件定義.....18

▼ 基本要件.....19
▼ 要求仕様／要件定義.....21
- シリアライズシステムの構成要素.....21
- プログラミングモデルに対する基本要件.....25
- ユーザー定義処理のプログラミングに関する要件.....26
- アーカイブに関する要件.....39

■ 処理仕様.....40
▼ プログラミングイメージ.....40
● シリアライズ／デシリアライズの基本形.....40
● ロード前処理.....41
● ロード後処理.....41
● 通知処理①：セーブデータにないデータ.....42
● 通知処理②：セーブデータにしかないデータ.....42
● コレクターとディストリビュータ.....43
▼ クラス設計.....44
▼ テンプレートオペレータ.....48
▼ 処理フロー：シリアライズ.....50
▼ 処理フロー：デシリアライズ.....51
▼ シリアライズの特殊な仕組み：ネストした構造体.....52
▼ デシリアライズの特殊な仕組み①：データ項目の不一致に対応した読み込み.....53
▼ デシリアライズの特殊な仕組み②：定義順序に依存しない読み込み.....53
▼ デシリアライズの特殊な仕組み③：ネストした構造体.....53
▼ デシリアライズの特殊な仕組み④：セーブデータにしかないデータの委譲.....54
▼ SFINAE を利用したユーザー定義処理実装判定方法.....54
■ データ仕様.....56
▼ シリアライズデータの基本構造.....56
■ 処理実装サンプル：シリアライズの使用サンプル.....58
▼ 【準備】ゲームのデータ管理システムのサンプル.....58
▼ データ管理システムのクラス図.....59
▼ サンプルプログラムについて.....60
▼ データ管理システムのサンプルプログラム：定義部.....60
▼ データ管理システムのサンプルプログラム：テスト処理部.....70
▼ データ管理システムに対するシリアライズ処理.....77
● 準備：フレンド化.....77
● 【任意】準備：バージョン定義.....78
● 準備：セーブデータ専用構造体.....79
● シリアライズ／デシリアライズの定義：汎用クラス.....79
● シリアライズ／デシリアライズの定義：共通データ.....80
● シリアライズ／デシリアライズの定義：アイテムデータ.....80
● シリアライズ／デシリアライズの定義：アビリティデータ.....81
● シリアライズ／デシリアライズの定義：キャラデータ.....81

- シリアライズ／デシリアライズの定義：進行＆フラグデータ 83
- 全体を統括するセーブデータの追加 83
- シリアライズ実行処理 87
- デシリアライズ実行処理 88
- テスト処理 89

▼ システムの依存関係 96
▼ 最適化のために 96

■ 処理実装サンプル：シリアライズの実装 98
▼ インクルードとネームスペース 98
▼ バージョンクラス 98
▼ 型操作クラス 100
▼ シリアライズ関数オブジェクト用テンプレートクラス 114
▼ データ項目管理クラス 118
▼ 処理結果クラス 126
▼ アーカイブ読み書きクラス 129
▼ アーカイブ読み書き用ヘルパークラス 144
▼ アーカイブ形式クラス 148
▼ アーカイブ形式クラス：バイナリ形式 150
▼ アーカイブ形式クラス：テキスト形式 161
▼ アーカイブクラス 168
▼ システムの依存関係 169

■ 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。そのプログラミングスタイルは、Boost C++ ライブラリの boost::serialization を参考にする。

■ 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。
通常、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。
通常、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。

なお、本文中、文脈によってはデシリアライズを意味している。

▼ 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、実際にシリアライズの処理を行い、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。

▼ コレクター（Collector）

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「コレクト」（Collect）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）

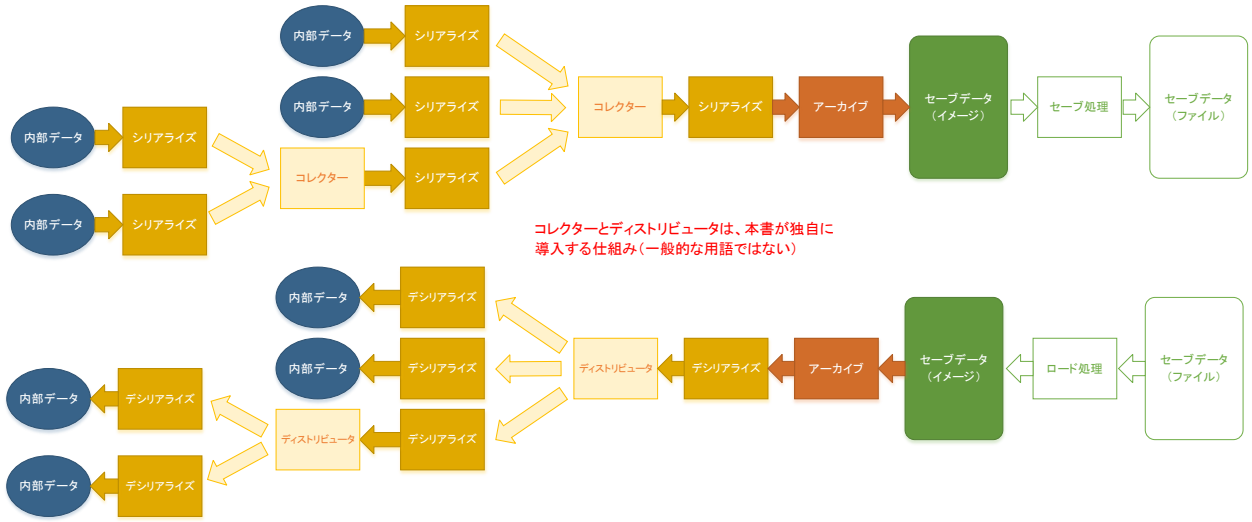
セーブデータの復元のために、読み込んだデータを各所に分散配置する処理を担う仕組みを意味する。また、「分配」（動詞）の意味では「ディストリビュート」（Distribute）という用語も用いる。

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：



「コレクター」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。

■ 【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、「コレクター」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。

▼ 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。

・ ゲームタイトル固有のセーブデータを、形式にとらわれることなく自由に扱うことができるシリアライズ処理とする。

・ バージョンを管理し、セーブデータの構造が変わっても、（できる限り）古いデータの読み込みが可能なシステムとする。
  - もしくは、そのための簡潔な仕組みを提供するシステムとする。

・ プログラマーの労力をできる限り抑えるシステムとする。
  - シリアライズとデシリアライズの処理コードを共通化できるものとする。
  - これはコードの簡略化を意味するが、プログラムサイズを小さくすることを意味するものではない。生産性を優先し、テンプレートによるプログラムサイズの肥大は許容するものとする。

・ シリアライズの際は、ゲームシステム中に分散する多数のデータをひとまとめにしてセーブデータを構成できるものとする。
  - このための任意の処理を、簡潔に定義可能とする。
  - この処理をシリアライズと区別し、データの「収集」を意味して「コレクト」と呼び、それを担う処理を「コレクター」と呼ぶものとする。

・ デシリアライズの際は、必要に応じてデータオブジェクトの追加・更新・削除を適切に行えるものとする。
  - このための任意の処理を、簡潔に定義可能とする。
    - 追加・削除の判断をシステムが自動的に行わない。
    - Boost 版のような自動的な new も行わない。
  - この処理をデシリアライズと区別し、データの「分配」を意味して「ディストリビュート」と呼び、それを担う処理を「ディストリビュータ」と呼ぶものとする。
  - ディストリビュートの際は、復元対象となるデータ、もしくは、復元対象となり得るデータを全て一掃して再構築するものとする。
    - そういう処理を任意に記述できるものとする。
  - デシリアライズ（ディストリビュート）により、稼働中のゲームに影響を与える可能性があるが、それは使う側の問題である。

・ シリアライズは、メモリ上のバッファに対してセーブデータのイメージを作成するものとする。
  - 直接ファイルに出力するようなことはしない。
  - 固定バッファの範囲内で処理するものとする。

  - 途中でバッファオーバーフローした場合、シリアライズは失敗とする。
    - 部分的な成功はなく、一切が失敗とする。

- デシリアライズは、メモリ上のセーブデータイメージから取り込むものとする。
  - 直接ファイルから読み込むようなことはしない。

- シリアライズの際は、元のデータに一切変更を加えないことを保証するものとする。
  - const オブジェクトとして処理し、シリアライズ処理中に直接データを更新するような誤ったコーディングがあったらコンパイルエラーになるものとする。
  - 【要対処】デシリアライズ時のバージョン互換のために、直接データを更新する処理が必要になることもあるが、デシリアライズ処理とシリアライズ処理のコードを共通化するとそれができなくなる。
    - この対処として、デシリアライズ時専用の処理を別メソッドで追加できるものとする。

- シリアライズ処理に変更を加えることなく、バイナリアーカイブとテキストアーカイブを扱えるものとする。
  - 【可能であれば】テキストアーカイブは JSON 形式とする。
  - テキストアーカイブにより、手作業によるセーブデータの量産を可能とし、QA 作業等の効率化に寄与する。

- 【できれば】デシリアライズ時のオプションにより、部分ロードを可能とする。
  - 例えば、プレイヤーキャラの成長状態のデータと、所持アイテムのデータ、ゲームの進行状態（章とフラグ）のデータを、それぞれ別々のセーブデータから読み込んで組み合わせることを可能とする。これにより、デバッグ効率を向上させる。
  - 部分ロードでは、ディストリビュータは部分ロードの対象データのみ（例えば所持アイテム系データのみ）を一掃して再構築するように挙動を変えるものとする。
    - そういう処理を任意に記述できるものとする。
  - 【要対処】部分ロードにより、データの不整合が生じる可能性がある。例えば、所持アイテムを部分ロードすることにより、プレイヤーキャラが装備していた武器が失われ、参照できなくなるなど。デシリアライズおよびディストリビュートの基本的な仕組みとして、このような問題についてはサポートしないものとする。
    - この対処は、ディストリビュータの処理で任意に対応する。
    - 部分ロードであることを判別して整合性の解消を行う。

- セーブデータに要求されることの多い「チェックサム」や「暗号化」は、本システムの対象外とする。
  - そうした処理は、本システムが作成したセーブデータイメージに対して実施するものとする。
  - ロード時は、そうした処理を通過した後のセーブデータイメージを本システムに受け

渡すものとする。

▼ 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。

● シリアライズシステムの構成要素

シリアライズシステムは、下記の要素で構成するものとする。
各要素の基本要件も合わせて示す。

【汎用処理】アーカイブ

シリアライズシステムの本体。

- 汎用クラスであり、ゲームタイトル固有の処理を実装することはない。
- セーブデータ読み書きのためのバッファを受け取って処理する。
- ユーザー定義処理（ゲームタイトル固有の処理）であるシリアライズの実行を担い、セーブデータの作成・復元を実行する。
- アーカイブには複数の種類があり、使用するアーカイブによってセーブデータの保存形式が異なる。
- 下記の基本クラスで構成する。なお、これらのクラスは直接使用しない。
  - ・ アーカイブ書き込みクラス
  - ・ アーカイブ読み込みクラス
  - ・ バイナリ形式アーカイブクラス
  - ・ テキスト形式アーカイブクラス
- 上記の基本クラスを組み合わせて、下記のクラスを構成する。
  - ・ バイナリ形式アーカイブ書き込みクラス
  - ・ バイナリ形式アーカイブ読み込みクラス
  - ・ テキスト形式アーカイブ読み込みクラス
  - ・ テキスト形式アーカイブ読み込みクラス

【ユーザー定義処理】シリアライズ

シリアライズとデシリアライズのための共通処理コード。

- ユーザー定義関数であり、ゲームタイトル固有の処理を扱う。

- テンプレート関数として定義する。
  - ・ これにより、コンパイル時にシリアライズのコードとデシリアライズのコードをそれぞれ生成する。
- 基本的に、データクラス本体にシリアライズ処理を内蔵せず、外部定義関数として構成する。
  - ・ Boost 版でいうところの「非侵入型（non-intrusive)」のみを扱うものとする。
- データクラス一つに対して一つのシリアライズ関数を定義する。
  - ・ セーブデータに必要なオブジェクトの数だけ連鎖的に多数のシリアライズが実行される。
- シリアライズ処理の中では、シリアライズの対象とするデータ項目の指定を行うのが基本。
  - ・ Boost 版と同様。
  - ・ 同じクラス内のメンバーでも、シリアライズ対象に指定されなかった項目はシリアライズされない。
  - ・ 対象項目がクラスの場合、そのクラスに対して定義されたシリアライズ処理が連鎖的に実行される。

【ユーザー定義処理】セーブ専用処理

シリアライズと同じだが、シリアライズ時にしか実行されない。（デシリアライズ時には実行されない）

- この処理を「セーブ」としたのは Boost 版の模倣。

【ユーザー定義処理】ロード専用処理

シリアライズと同じだが、デシリアライズ時にしか実行されない。（シリアライズ時には実行されない）

- この処理を「ロード」としたのは Boost 版の模倣。

【ユーザー定義処理】ロード前処理

デシリアライズの際、セーブデータを読み込む前に実行される処理。

- 読み込み先のインスタンスを生成したりなど、必要に応じてデシリアライズの準備を行う。
- Boost 版では読み込み先のポインタがヌルなら自動的に new するが、それは対応しない。
- Boost 版にはない処理。

【ユーザー定義処理】ロード後処理

デシリアライズの際、セーブデータを読み込んだ後に実行される処理。

- 読み込んだデータに基づいてポインタの参照先を割り当てるなど、事後処理や整合性解消処理などを行う。
- Boost 版にはない処理。

【ユーザー定義処理】ロード結果通知

デシリアライズの結果、セーブデータと現在のデータ構造との不整合を通知する処理。

- シリアライズ対象データとして指定されているが、セーブデータになかったデータ項目を通知。
- セーブデータには存在するが、シリアライズ対象データとして指定されておらず、読み込めなかったデータ項目を通知。
- Boost 版にはない処理。

【ユーザー定義処理】コレクター

シリアライズの際、他のオブジェクトをいっしょにセーブデータに保存するために収集する処理。

- 直接関連性のないオブジェクトをひとまとめにするために用いる。
  - ・ クラスのメンバー変数としてのクラスなら普通のシリアライズで処理できる。
- オブジェクトを集めてシリアライズを実行する処理を任意に記述する。
- コレクターで集めたデータを復元するためのディストリビュータと必ず一対で定義する。
- Boost 版にはない処理。

【ユーザー定義処理】ディストリビュータ

デシリアライズの際、コレクターによってセーブデータに記録されたオブジェクトを分配（復元）するための処理。

- コレクターによって記録されたオブジェクトの情報が通知される。
  - ・ コレクターが記録した情報の数だけ繰り返し呼び出される。
- 対象オブジェクトに見合ったインスタンス生成の処理とそれに対するデシリアライズを任意に記述する。
- 必ずコレクターと一対で定義する。
- Boost 版にはない処理。

【ユーザー定義処理】ディストリビュータ前処理

一連のディストリビュータが実行される前に実行される処理。

- 例えば、プールアロケータを用いるオブジェクトを扱う場合、ディストリビュータではプールからのメモリ割り当てを行うが、その前のプール自体の初期化を、前処理として実施する。

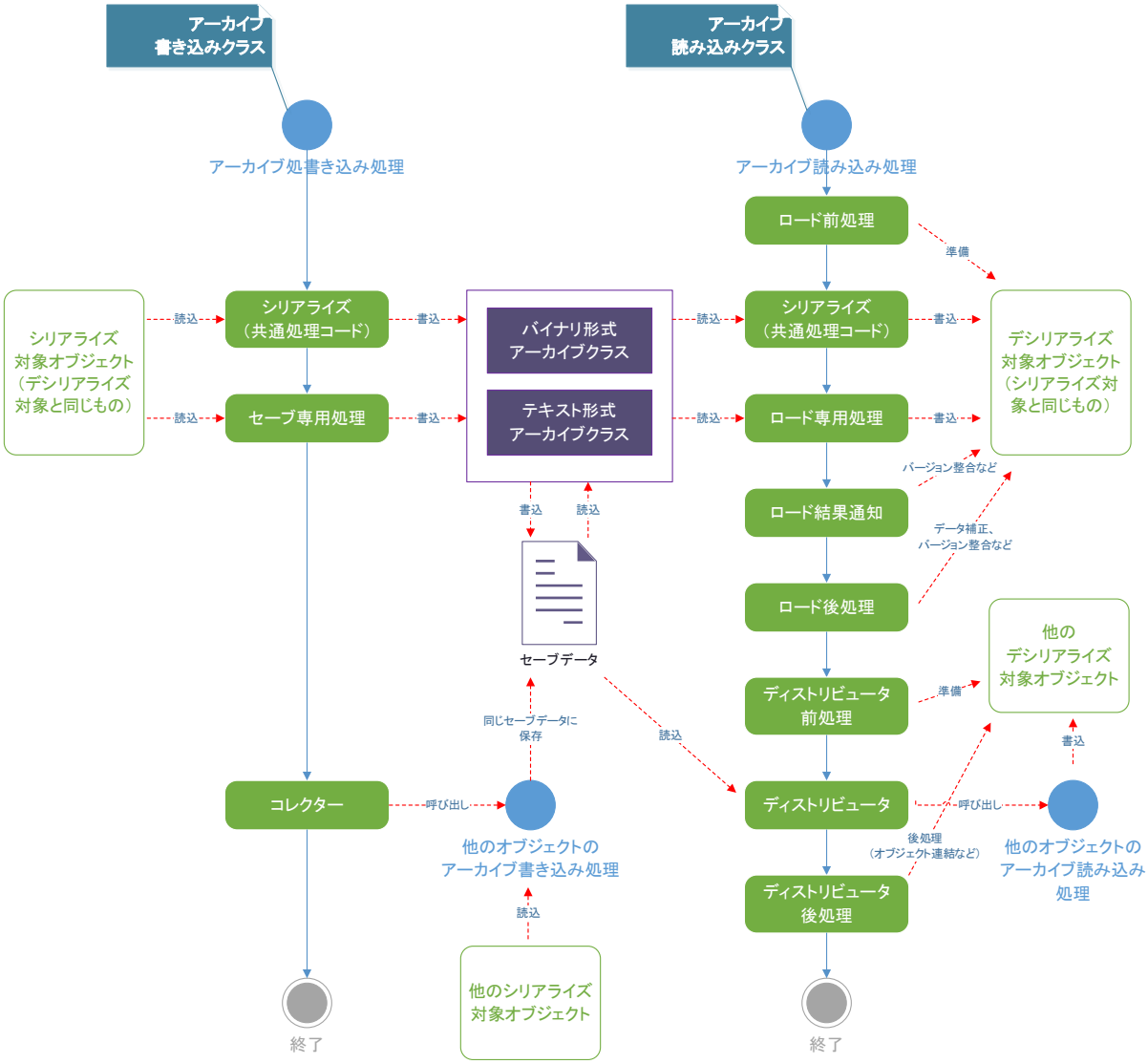
【ユーザー定義処理】ディストリビュータ後処理

一連のディストリビュータが実行された後に実行される処理。

- 例えば、オブジェクト間の関連付けや、部分ロード時の不整合解消などを行う。

構成要素の関係図

以上の構成要素の関係図を示す。



● プログラミングモデルに対する基本要件

- 全般的にテンプレートを活用することで、コンパイル時にユーザー定義処理が共通処理に取り込まれてコード生成されるものとする。
  - これにより、ユーザー定義処理は必要最小限のコーディングで済むものとする。
  - テンプレートクラスの特殊化、および、テンプレート関数のオーバーロードを利用すると、明示的にテンプレート引数で型を与えることなく、共通処理にユーザー定義処理を渡すことができるため、コーディングを簡略化できる。（Boost 版と同様の手法）

- これによるコードサイズの肥大は注意すべきものとなるが、コーディングの利便性の方を重視するものとする。

● ユーザー定義処理のプログラミングに関する要件

- ユーザー定義処理は、Boost 版と同様の簡潔な記述でコーディングできるものとする。
- ユーザー定義処理の種類が多いが、不要なものは定義しなくても良いものとする。
- セーブする対象のクラス（構造体）に、シリアライズ／デシリアライズのための処理を記述しなくてもよいものとする。
  - Boost 版ではクラス内にシリアライズのコードを記述するのが基本だが、それには対応しない。
  - セーブ／ロード処理だけを切り分けてまとめることで、簡潔なコードになるようにする。
- ユーザー定義処理は、Boost 版の非侵入型（non-intrusive）と同様の外部関数のみに対応するものとする。
- ユーザー定義処理は、Boost 版と異なり、テンプレート関数ではなく、テンプレートクラスによる関数オブジェクトで実装するものとする。
  - 【理由】テンプレートクラスを用いると、特殊化（と SFINAE 機構）を利用して、ユーザー定義処理の実装状態を静的に（コンパイル時に）チェックすることができるため。

以下、Boost 版と比べて余計に記述しなければならない部分を赤字で示す。若干構造が異なるぐらいで、コード量には大差ない。

【Boost 版】※非侵入型のテンプレート関数

```
namespace boost{
	namespace serialization{
		template<class Archive>
		void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
		{
			arc & obj.m_a;
			arc & obj.m_b;
			arc & obj.m_c;
		}
	}
}
```

【本システム】※テンプレートクラスによる関数オブジェクト

```
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {//←特殊化のためにテンプレート引数に型を明示する必要がある
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b);
			arc & pair("data3", obj.m_c);
		}
	}
};
```

- バージョンは、Boost 版と同様に、クラス（構造体）ごとに設定可能とする。
- バージョンの扱いは、Boost 版と異なり、メジャーバージョンとマイナーバージョンの組み合わせで定義できるものとする。

【Boost 版】※BOOST_CLASS_VERSION マクロで定義

```
struct USER_DATA
{
	int m_a
	float m_b;
	char m_c[3];
};
BOOST_CLASS_VERSION(USER_DATA, 2);//USER_DATA Ver.2.0
```

【本システム】※SERIALIZE_VERSION_DEF マクロで定義

```
SERIALIZE_VERSION_DEF(USER_DATA, 2, 1);//USER_DATA Ver.2.1
```

- デシリアライズ時には、Boost 版と同様に、セーブデータから読み込んだバージョンがユーザー定義処理に渡されるものとする。
- ユーザー定義処理に渡されるバージョンは、Boost 版と異なり、メジャーバージョンとマイナーバージョンを格納した「バージョン定義クラス」で渡されるものとする。
- ユーザー定義処理に渡されるバージョンは、Boost 版と異なり、セーブデータから読み込んだバージョンだけではなく、現在のバージョンも渡されるものとする。
  - セーブデータのバージョンが現在のバージョンと同じかどうかを素早くチェックするぐらいしか使い道がないが、少しだけ便利になるので。

【Boost 版】

```
template<class Archive>
void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
```

【本システム】

```
template<class Archive>
struct serialize<Archive, USER_DATA> {
	void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
	{
		if(ver < now_ver)//オブジェクト同士を直接比較可能
```

- ユーザー定義処理内では、Boost 版と同様に、アーカイブオブジェクトに対する「&」演算子でシリアライズ対象のデータ項目（メンバー変数）を指定するものとする。
  - テンプレートにより、アーカイブ読み込み時とアーカイブ書き込み時で「&」演算子の振る舞いを変えることができるため、シリアライズとデシリアライズの処理コードを共通化できる。
- シリアライズ対象データ項目を指定する際は、Boost 版と異なり、必ず「名称」を与えるものとする。
  - バイナリデータ時にもデータ項目の識別子として保存する。
  - この「名称」（を CRC にしたもの）を記録することで、本システムの要点である「ディストリビュータ」の処理を成立させることができる。
    - ユーザー定義処理内で、「名称」に対する分配先（復元先）のデータを指定することができる。

- なお、Boost 版でも XML アーカイブを扱う場合は名称を必要とするが、それはマクロによって自動的に与えられる。本システムでは、「コレクター」と「ディストリビュータ」（後述）のように、複数の処理で名前を合わせて処理する必要があるため、必ず明示的に名前を指定するものとする。

【Boost 版】

```
template<class Archive>
void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
{
	arc & obj.m_a;
	arc & obj.m_b & obj.m_c;//演算子の連結も可
}
```

【Boost 版の XML アーカイブ時】※マクロを使用して自動的に XML の項目名を与えている

```
template<class Archive>
void serialize(Archive& arc, const USER_DATA& obj,const unsigned int version)
{
	arc & BOOST_SERIALIZATION_NVP(obj.m_a);
	arc & BOOST_SERIALIZATION_NVP(obj.m_b) & BOOST_SERIALIZATION_NVP(obj.m_c);//演算子の連結も可
}
```

【本システム】※pair()関数を使用して名前を与える

```
template<class Archive>
struct serialize<Archive, USER_DATA> {
	void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
	{
		arc & pair("data1", obj.m_a);//「&」演算子でシリアライズ対象データ項目を指定
					//※（pair 関数を使用し、必ず名前とペアで指定）
		arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);//演算子の連結も可
	}
};
```

➢ シリアライズの実行は、Boost 版と同様に、アーカイブオブジェクトとシリアライズ対象オブジェクトを「<<」演算子でつないで実行するものとする。

- この時もやはり「名称」を与える。

【Boost 版】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//ファイル出力ストリーム生成
std::ofstream os("file.bin");
//アーカイブ書き込みオブジェクト生成
boost::archive::binary_oarchive arc(os);
//シリアライズ
arc << user_data;
//ストリームをクローズ
os.close();
```

【Boost 版の XML アーカイブ時】

```
//アーカイブ書き込みオブジェクト生成
boost::archive::xml_oarchive arc(os);
//シリアライズ
arc << BOOST_SERIALIZATION_NVP(user_data);
```

【本システム】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//バッファ
static const std::size_t SAVE_DATA_BUFF_SIZE = 32 *1024;
char save_data_buff[SAVE_DATA_BUFF_SIZE];
```

```
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pair("UserData", user_data);//実行時には名前を与える
```

- デシリアライズの実行は、Boost 版と同様に、アーカイブオブジェクトとシリアライズ対象オブジェクトを「>>」演算子でつないで実行するものとする。
  - この時もやはり「名称」を与える。
  - 以下、シリアライズ時と異なる箇所を赤字で示す。

【Boost 版】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//ファイル入力ストリーム生成
std::ifstream is("file.bin");
//アーカイブ読み込みオブジェクト生成
boost::archive::binary_iarchive arc(is);
//デシリアライズ
arc >> user_data;
//ストリームをクローズ
is.close();
```

【Boost 版の XML アーカイブ時】

```
//アーカイブ読み込みオブジェクト生成
boost::archive::xml_iarchive arc(is);
//デシリアライズ
arc >> BOOST_SERIALIZATION_NVP(user_data);
```

【本システム】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//セーブデータバッファ取得
std::size_t save_data_size = 0;
void* save_data = getLoadedSaveDataFile(&save_data_size);
//バッファ
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pair("UserData", user_data);//実行時には名前を与える
```

- 「<<」演算子および「>>」演算子が指定できるのは、ユーザー定義処理が定義されたクラスもしくは構造体に限定するものとする。
  - テンプレートの特殊化などに頼った処理構造のため、任意のクラス（構造体）として識別する必要がある。
  - int 型や char の配列などのデータを直接シリアライズすることができない。

- Boost 版と同様に、クラスのプライベートメンバーにアクセスするための手段を用意するものとする。
  - Boost 版と異なり、非侵入型の処理に対してフレンド化する手段を提供する。
  - 簡潔に定義できるように、専用マクロを用意する。

【Boost 版】※侵入型（クラス／構造体内に直接シリアライズ処理を実装）のみ対応

```
class USER_DATA
{
private:
	int m_a
	float m_b;
	char m_c[3];
	friend class boost::serialization::access;//フレンド化
	//シリアライズ処理
	template<class Archive>
	void serialize(Archive& arc, const unsigned int version)
	{
		arc & obj.m_a;
		arc & obj.m_b & obj.m_c;
	}
};
```

【本システム】※SERIALIZE_FRIEND マクロで定義

```
class USER_DATA
{
private:
	int m_a
	float m_b;
	char m_c[3];
	FRIEND_SERIALIZE();//フレンド化
};
//シリアライズ処理
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
		}
	};
}
```

- ユーザー定義処理のシリアライズ処理（共通処理コード）ではデータを直接書き換えることを禁止するものとする。
  - データの書き換えは「ロード後処理」を利用するものとする。
  - ロード後処理は、必要な時だけ定義する。

【問題のあるシリアライズ処理】

```
//シリアライズ処理
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {//↓USER_DATA に const が付く
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
			//バージョン整合処理
			if (ver >= CVersion(1, 2))//バージョンが 1.2 以上なら m_d 読み込み（もしくは書き込み）
			{
				arc & pair("data4", obj.m_d);
			}
			else//バージョン 1.2 未満ならセーブデータに m_d がないので直接値を代入
			{
				obj.m_d = 123;//初期値代入
				//	↑コンパイルエラー！
				//※ロードの時だけ実行されることを意図したコードだが、
```

```
                        //  シリアライズとの共通処理内ではデータの更新が許可されない
                }
        }
    };
}
```

↓（変更後）

```
//シリアライズ処理
namespace serial{
    template<class Archive>
    struct serialize<Archive, USER_DATA> {
        void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            arc & pair("data1", obj.m_a);
            arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
            //バージョン整合処理
            if (ver >= CVersion(1, 2))//バージョンが 1.2 以上なら m_d 読み込み（もしくは書き込み）
            {
                arc & pair("data4", obj.m_d);
            }
        }
    };
}
//ロード後処理
namespace serial{
    template<class Archive>
    struct afterLoad<Archive, USER_DATA> {//↓USER_DATA に const が付かない
        void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            //バージョン 1.2 未満ならセーブデータに m_d がないので直接値を代入
            if (ver < CVersion(1, 2))
            {
                obj.m_d = 123;//初期値代入
                //      ↑コンパイル OK
            }
        }
    };
}
```

- 同様に「ロード前処理」でもデータの書き換えを可能とする。
  - 事前のメモリ割り当てなどを行うことが可能。
  - ロード前処理は、必要な時だけ定義する。

【ロード前処理】

```
//ロード前処理
namespace serial{
    template<class Archive>
    struct beforeLoad<Archive, USER_DATA> {//↓USER_DATA に const が付かない
        void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            obj.m_d = new EXTRA_DATA;//データを読み込むための領域を事前に割り当てておく
        }
    };
}
```

- シリアライズ対象に指定されたデータ項目が記録されていないセーブデータを読み込んでも処理がスキップされるだけで、他のデータが問題なく読み込めるものとする。
  - 構造体にメンバーが追加された後で古いセーブデータを読み込む場合など。

【シリアライズ処理】

```
//シリアライズ処理
```

```
namespace serial{
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
			arc & pair("data4", obj.m_d);//←バージョン 1.2 以降に追加されたメンバー
			//	（特殊な処理をせずとも問題なし）
		}
	};
}
```

- シリアライズ対象のデータ項目が記録されていないセーブデータを読み込んだ場合、「通知処理」により、問題が発生したデータ項目が通知されるものとする。
  - プログラム上で新たにデータ項目が追加された後に、古いセーブデータを読み込んだ場合などに起こる。
  - 問題が検出されたデータ項目の数だけ呼び出される。
  - （そのオブジェクトの）一通りのデータ項目の読み込みが済んだ後、「ロード後処理」の前に呼び出される。
  - 通知処理は、必要な時だけ定義する。

【保存先の指定があるが、セーブデータになくロードできなかったデータ項目の通知処理】

```
//保存先の指定があるが、セーブデータになくロードできなかったデータ項目の通知
namespace serial{
	template<class Archive>
	struct noticeUnloadedItem<Archive, USER_DATA> {//↓USER_DATA に const が付かない
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
			, const CItemBase& unloaded_item)
		{	//↑読み込めなかったデータ項目が渡される
			//data4 はバージョン 1.2 未満のセーブデータには存在しない
			if (unloaded_item == "data4")//引数 unloaded_item で問題の項目をチェック
			{
				obj.m_d = 123;//初期値代入
			}
			//バージョンで比較して処理をせずとも、確実な整合処理を行うことができる
		}
	};
}
```

- シリアライズ対象のデータ項目ではない、セーブデータにだけ存在するデータ項目があっても問題なく動作し、専用の「通知」処理で通知も行われるものとする。
  - プログラム上でデータ項目を削除、もしくは改名した後に、古いセーブデータを読み込んだ場合などに起こる。
  - 問題が検出されたデータ項目の数だけ呼び出される。
  - 読み込みの途中、問題が見つかったタイミングで呼び出されるため、正しい読み込み先を指定し直して再度読み込みすることができる。
  - 通知処理は、必要な時だけ定義する。

【セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理】

```
//セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理
namespace serial{
	template<class Archive>
```

```
	struct noticeUnrecognizedItem<Archive, USER_DATA> {//↓USER_DATA に const が付かない
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
                                                    , const CItemBase& unrecognized_item)
		{                                           //↑読み込めなかったデータ項目が渡される
			//data_x は現在 data5 に改名
			if (unrecognized_item == "data_x")//引数 unrecognized_item で問題の項目をチェック
			{
				arc & pair("data5", obj.m_e5);//読み込み先を指定し直して再実行
				//「&」演算子で改めてシリアライズ対象項目を指定し直すことで
				//データの読み込み先を委譲する

				//この関数内では obj の値を更新してもよい
				obj.m_preblemItem = unrecognized_item.m_nameCrc;//問題があった項目を記録
			}
		}
	};
}
```

- プログラム上の構造体のメンバーの順序とそれをシリアライズ対象項目として指定する順序、それと、（古い）セーブデータ上のデータ項目の順序は、それぞれ違っていても問題なく読み込みできるものとする。
  - プログラム上で構造体のメンバーの順序を変更しても不整合が起こらないものとする。
  - このための任意の処理や設定をユーザー定義する必要はないものとする。
    - 順序が異なるデータ項目をシステムに伝える手続きは不要。
  - データ項目名に基づいて自動的に整合性の解消が行われるものとする。
  - なお、書き込みはシリライズ対象に指定したデータ項目の順に行われるものとする。

- プログラム上の構造体のメンバーの型や配列のサイズが変更されても、（古い）セーブデータを問題なく読み込みできるものとする。
  - この変更によって発生する問題は居所的なものであり、それ以外のデータは正しく読み込めるものとする。
  - このための任意の処理や設定をユーザー定義する必要はないものとする。
    - 型や配列要素数が異なるデータ項目をシステムに伝える手続きは不要。
  - 問題のあったデータも、読み込み可能な範囲で読み込むものとする。
    - 例えば、int → short に変わった項目は、2 バイトの読み込みを行う。
    - また、例えば、short → int に変わった項目は、2 バイトの読み込み、残りバイトを 0 で埋める。
    - リトルエンディアンなら前詰め、ビッグエンディアンなら後詰めで処理する。
    - 配列の要素ごとにこの変換を行う。
    - 配列の要素数が縮小されている場合は、余りを読み捨てる。
    - 配列の要素数が拡大されている場合は、残りを無視する。（0 で初期化したりもしない）
    - オーバーフローによるデータ破壊、セーブデータの読み込み位置のずれといった、他のデータにも影響が及ぶことはないようにする。

➢ ユーザー定義処理の「コレクター」と「ディストリビュータ」を定義することにより、対象オブジェクトと直接関連性のないデータを一つのセーブデータとしてまとめて扱えるものとする。

- コレクターとディストリビュータは、必ずセットで定義する。
- コレクターとディストリビュータは、必要な時だけ定義する。
- コレクターの処理で、複数のデータのシリアライズを行うと、ディストリビュータはそのデータの数だけ呼び出される。
- ディストリビュータには、「ディストリビュート前処理」と「ディストリビュート後処理」があり、それぞれ一連のディストリビュート処理の前後に 1 回ずつ実行される。
  - デシリアライズ前のデータの削除や、デシリアライズ後のデータ整合処理などに用いる。

【コレクター】

```
//コレクター
template<class Arc>
struct collector<Arc, USER_DATA> {
	void operator()(Arc& arc, const USER_DATA& obj, const CVersion& ver)
	{
		//USER_DATA の後に連続して下記のデータもいっしょにシリアライズする

		//インベントリのデータを収集
		CSingleton<CInventory> inventory;
		for (auto item_data : *inventory)//ループ処理で全要素を取得
		{
			//アイテムデータを一つずつシリアライズ
			arc << pair("item", *item_data);
		}
		//進行＆フラグデータをシリアライズ
		CSingleton<CPhaseAndFlags> phase_and_flags;
		arc << pair("phase_and_flags", *phase_and_flags);
	}
};
```

【ディストリビュート前処理】

```
//ディストリビュート前処理
template<class Arc>
struct beforeDistribute<Arc, USER_SAVE_DATA> {//↓USER_DATA に const が付かない
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
			//↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
			//  渡さてくる
	{
		//データを読み込む前に、読み込み先を一旦空にする

		//インベントリデータクリア
		CSingleton<CInventory> inventory;
		inventory.destroy();
		//フェーズ＆進行データクリア
		CSingleton<CPhaseAndFlags> phase_and_flags;
		phase_and_flags.destroy();
	}
};
```

【ディストリビュータ】

```
//ディストリビュータ
template<class Arc>
struct distributor<Arc, USER_DATA> {//↓USER_DATA に const が付かない
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
```

```
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
                                                                const CItemBase& target_item)
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡される（ディストリビュート前処理と同じ
                                                                //↑対象データ項目が渡される
    {
        //USER_DATA の後に連続して下記のデータもいっしょにデシリアライズされる

        //コレクターでシリアライズしたデータの数だけ繰り返し実行されるので、
        //データ項目名を判定して一つずつ処理する
        //※無視したらデータは読み込まれずに次のデータに進む

        if (target_item == "item")//インベントリデータか？
        {
            //インベントリデータ復元
            CSingleton<CInventory> inventory;
            ITEM_DATA item_data;
            arc >> pair("item", item_data);//デシリアライズ
            inventory->regist(item_data);//インベントリにデータを登録
        }
        else if (target_item == "phase_and_flags")//進行データか？
        {
            //フェーズ＆進行データ復縁
            CSingleton<CPhaseAndFlags> phase_and_flags;
            arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
        }
    }
};
```

【ディストリビュート後処理】

```
//ディストリビュート後処理
template<class Arc>
struct afterDistribute<Arc, USER_DATA> {//↓USER_DATA に const が付かない
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡される（ディストリビュート前処理と同じ
    {
        //全ての読み込みが完了
        //インベントリデータが更新されたので、キャラが所持する武器データの参照を再設定する
        CSingleton<CCharaList> chara_list;
        for (auto& chara_data : *chara_list)//ループ処理で全キャラを取得
        {
            chara_data->attachItems();//アイテムを参照し直す
        }
    }
};
```

- セーブデータの部分ロードは、ディストリビュータを工夫することによって実現可能とする。
  - 例えば、「インベントリデータは『セーブデータ A』から読み込み、進行データは『セーブデータ B』から読み込んで組み合わせる」といったことができると、QA や制作のためのセーブデータの組み合わせパターンを用意する手間を軽減できる。
  - このような処理は、ここまでに示したシステムの応用で十分対応可能となる。

【部分ロード実行】

```
//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//部分ロード設定
setLoadTarget("phase_and_flags");//部分ロード指定：進行データのみをロード
```

```
resetLoadTarget();//部分ロードを解除して全体ロードする場合
//デシリアライズ
arc >> serial::pair("UserData", user_data);
```

【部分ロードのための処理】※シリアライズの仕組みとは直接関係のない処理

```
//部分ロード設定
crc32_t s_loadTarget;//ロード対象データ項目名 (CRC)
void setLoadTarget(const char* name)//ロード対象データ項目名をセット
{
	s_loadTarget = calcCRC32(name));
}
void resetLoadTarget()//ロード対象データ項目をリセット
{
	s_loadTarget = 0;
}
bool isLoadTarget(const char* name)//ロード対象データ項目か？
{
	return s_loadTarget == 0 || s_loadTarget == calcCRC32(name_crc);
}
```

【ディストリビュート前処理】※部分ロード処理を追加

```
//ディストリビュート前処理
template<class Arc>
struct beforeDistribute<Arc, USER_DATA> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
	{
		//データを読み込む前に、読み込み先を一旦空にする
		//※部分ロード時は、部分ロードの対象データのみを破棄

		if(isLoadTarget("item"))
		{
			//インベントリデータクリア
			CSingleton<CInventory> inventory;
			inventory.destroy();
		}
		if(isLoadTarget("phase_and_flags"))
		{
			//フェーズ＆進行データクリア
			CSingleton<CPhaseAndFlags> phase_and_flags;
			phase_and_flags.destroy();
		}
	}
};
```

【ディストリビュータ】※部分ロード処理を追加

```
//ディストリビュータ
template<class Arc>
struct distributor<Arc, USER_DATA> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
				const CItemBase& target_item)
	{
		//USER_DATA の後に連続して下記のデータもいっしょにデシリアライズされる

		if (target_item == "item" && isLoadTarget(target_item))//インベントリデータか？
		{
			//インベントリデータ復元
			CSingleton<CInventory> inventory;
			ITEM_DATA item_data;
			arc >> pair("item", item_data);//デシリアライズ
			inventory->regist(item_data);//インベントリにデータを登録
		}
		else if (target_item == "phase_and_flags" && isLoadTarget(target_item))//進行データか？
		{
```

```
                //フェーズ＆進行データ復縁
                CSingleton<CPhaseAndFlags> phase_and_flags;
                arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
            }
        }
};
```

【ディストリビュート後処理】※部分ロード処理を追加

```
//ディストリビュート後処理
template<class Arc>
struct afterDistribute<Arc, USER_DATA> {
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
    {
        //全ての読み込みが完了
        //インベントリデータが更新されているなら、キャラが所持する武器データの参照を再設定する
        if(isLoadTarget("item"))
        {
            CSingleton<CCharaList> chara_list;
            for (auto& chara_data : *chara_list)//ループ処理で全キャラを取得
            {
                chara_data->attachItems();//アイテムを参照し直す
            }
        }
    }
};
```

- 細かい互換性維持を捨てて、シリアライズ対象項目をひとくくりに扱う手段も用意するものとする。
  - ここまでの要件で示したとおり、すべてのシリアライズ対象項目を「&」演算子で指定さえすれば、バージョン互換性に優れた利便性の高いセーブデータシステムにすることができるが、反面、コードサイズの肥大化やセーブデータサイズの肥大化、処理の低速化が懸念される。
  - この対処として、データ項目の型を無視してバイトデータのかたまりとして記録する手段を用意するものとする。

【構造体まるごとをバイトデータのかたまりとして扱う】

```
struct USER_DATA
{
    int m_a
    float m_b;
    char m_c[3];
    struct SUB
    {
        int x;
        float y;
        short z[5];
    } m_sub;//サブ構造体のメンバー
};

template<class Archive>
struct serialize<Archive, USER_DATA> {
    void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
    {
        arc & pair("data1", obj.m_a);
        arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
        arc & pairBin("sub", obj.m_sub);//構造体まるごとをバイトデータとして扱う
        //※本来はサブ構造体を１項目ずつ指定するか、
        //  struct serialize<Archive, USER_DATA::SUB> を定義する
    }
};
```

- ただし、この「pairBin」の仕組みが使えるのは「&」演算子に対してのみ。「<<」演算子、および、「>>」演算子は、その仕組み上「型」が指定されている必要があるため、バイトデータ扱いにできない。（通常でも、これらの演算子には、ユーザー定義処理に対応したクラスもしくは構造体しか指定できない）

【>>演算子、<<演算子に pairBin の指定は不可】

```
//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pairBin("UserData", user_data);//←NG

//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pairBin("UserData", user_data);//←NG
```

➢ （主にテキストアーカイブのために）char 型の配列と文字列を区別してシリアライズするための仕組みを用意するものとする。

- これにより、テキストアーカイブ時に、char 型の配列は数値で扱われ、文字列は文字列として扱われるようにする。
- ポインタ型の変数も扱えるようにする。
- ポインタ型の場合、配列の要素数を自動的に読み取ることができないので、要素数を明示する必要がある。

【文字列を扱う】

```
struct USER_DATA
{
	char m_data[10];//数値データの配列
	char m_str[10];//文字列データ
	char* m_dataP;//数値データ（ポインタ型）
	char* m_strP;//文字列データ（ポインタ型）
};

template<class Archive>
struct serialize<Archive, USER_DATA> {
	void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
	{
		arc & pair("data", obj.m_data);//数値の配列（通常通りの指定方法）
		arc & pairStr("str", obj.m_str);//文字列
		arc & pairArray("data_arr", obj.m_dataP, 10);//ポインタを数値の配列として扱う
		//arc & pair( “data_p, obj_m_dataP);//この場合は配列ではなく char 型の値一つ（のポインタ）
                                                //として扱う
		arc & pairStr("str_p", obj.m_strP);//文字列ポインタ（文字列長は自動的に求める／ヌル時は 0）
		//※ポインタ型は、デシリアライズ時には先に領域が割り当てられている必要がある
	}
};
```

➢ （主にセーブデータの起点として）コレクターとディストリビュータだけを使用したい場合、クラスのインスタンスを用意しなくてもシリアライズ／デシリアライズを実行できるものとする。

- セーブデータに多数のデータを集めることだけを目的としたクラスの場合、それ自体はシリアライズする必要がない場合がある。

【インスタンスのないクラスでシリアライズ／デシリアライズ】

```
//セーブデータ用クラス
class SAVE_DATA_TOP{};//中身なし

//コレクター
template<class Arc>
struct collector<Arc, SAVE_DATA_TOP> {
	void operator()(Arc& arc, const USER_DATA& obj, const CVersion& ver)
	{
		//多数のシリアライズ処理
		//…（略）…
	}
};
//ディストリビュータ
template<class Arc>
struct distributor<Arc, SAVE_DATA_TOP> {
	void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
			const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
			                                                const CItemBase& target_item)
	{
		//多数のデシリアライズ処理
		//…（略）…
	}
};

//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pair<SAVE_DATA_TOP>("SaveData");//←インスタンス不要（クラス名と名前だけ指定）

//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pair<SAVE_DATA_TOP>("SaveData";//←インスタンス不要（クラス名と名前だけ指定）
```

● アーカイブに関する要件

アーカイブクラスがシリアライズ／デシリアライズ処理の本体。

- アーカイブクラスもまたテンプレートで構成する。
  - 基本の「アーカイブ読み込み／書き込みクラス」に対して、「アーカイブ形式クラス」を与えることでインスタンス化するものとする。

- 「アーカイブ形式クラス」は、後からの追加が可能なものとする。
  - アーカイブ読み込み／書き込みクラスは、特定のアーカイブ形式クラスに特化しない、（できるだけ）汎用的な処理構造とする。
    - 例えば、バイナリ形式の読み込み処理の場合、順次データを読み込みながら処理をするため、バッファのカレントポインタを操作するが、テキスト形式は事前のパースが必要な可能性がある。もし、共通処理中にバッファのシークがあると、バイナリ形式以外の形式が扱えなくなる可能性がある。読み込んだセーブデータイメージをどのように扱うかは、完全にアーカイブ形式クラスに任せ、抽象化したインターフェースでデータの入出力を求める構造とする。

■ 処理仕様

▼ プログラミングイメージ

プログラミングのイメージは要件定義に示したとおり。

事前にユーザー定義処理を定義し、シリアライズしたいオブジェクトをアーカイブオブジェクトに渡すだけで良い。

以下、要件定義の内容の繰り返しになるが、シリアライズ／デシリアライズに必要なプログラミング要素を簡単に列挙する。詳しい説明については要件定義を参照。

● シリアライズ／デシリアライズの基本形

最低限シリアライズに必要な要素。

【クラス／構造体の定義とフレンド宣言】※フレンド宣言は private メンバーへのアクセスが必要な時のみ

```
//シリアライズ対象のクラス／構造体
struct USER_DATA
{
	int m_a
	float m_b;
	char m_c[3];
	struct SUB
	{
		int m_x;
		float m_y;
		short m_z[5];
	} m_sub;//サブ構造体のメンバー
	char m_data[10];//数値データの配列
	char m_str[10];//文字列データ
	char* m_dataP;//数値データ（ポインタ型）
	char* m_strP;//文字列データ（ポインタ型）
	FRIEND_SERIALIZE();//【必要があれば】フレンド化設定（private メンバーにアクセスする場合のみ）
};
```

【クラス／構造体のバージョン設定】※バージョン設定の必要がある時のみ

```
SERIALIZE_VERSION_DEF(USER_DATA, 2, 1);//【必要があれば】USER_DATA Ver.2.1
```

【シリアライズ／デシリアライズ共通処理を定義】

```
//シリアライズ＆デシリアライズ共通処理
namespace serial {
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);//「&」演算子でシリアライズ対象データ項目を指定
							//※（pair 関数を使用し、必ず名前とペアで指定）
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);//演算子の連結も可
			arc & pairBin("sub", obj.m_sub);//構造体まるごとをバイトデータとして扱う
			arc & pair("data", obj.m_data);//数値の配列（通常通りの指定方法）
			arc & pairStr("str", obj.m_str);//文字列
			arc & pairArray("data_arr", obj.m_dataP, 10);//ポインタを数値の配列として扱う
			//arc & pair( "data_p, obj_m_dataP);//この場合は配列ではなく char 型の値一つ（のポインタ）として扱う
			arc & pairStr("str_p", obj.m_strP);//文字列ポインタ（文字列長は自動的に求める／ヌル時は 0）
		}
	};
```

```
};
```

【シリアライズ実行】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//バッファ
static const std::size_t SAVE_DATA_BUFF_SIZE = 32 *1024;
char save_data_buff[SAVE_DATA_BUFF_SIZE];
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ書き込みオブジェクト生成
serial::COBinaryArchive arc(save_data_buff, SAVE_DATA_BUFF_SIZE, work_buff, WORK_BUFF_SIZE);
//シリアライズ
arc << serial::pair("UserData", user_data);//実行時には名前を与える
```

【デシリアライズ実行】

```
//アーカイブ対象オブジェクト取得
USER_DATA& user_data = getUserData();
//セーブデータバッファ取得
std::size_t save_data_size = 0;
void* save_data = getLoadedSaveDataFile(&save_data_size);
//バッファ
static const std::size_t WORK_BUFF_SIZE = 32 *1024;
char work_buff[WORK_BUFF_SIZE];
//アーカイブ読み込みオブジェクト生成
serial::CIBinaryArchive arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);
//デシリアライズ
arc >> serial::pair("UserData", user_data); //実行時には名前を与える
```

● ロード前処理

デシリアライズの際、データを読み込む準備を行いたい場合、「ロード前処理」を定義する。

【ロード前処理】

```
//ロード前処理
namespace serial{
	template<class Archive>
	struct beforeLoad<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			obj.m_dataP = new char[10];//ロード前にメモリ割り当て
			obj.m_strP = new char[10 + 1];//ロード前にメモリ割り当て
		}
	};
}
```

● ロード後処理

デシリアライズの際、バージョン判定や読み込んだデータの状態などに基づいてデータを更新したい場合、「ロード後処理」を定義する。

【ロード後処理】

```
//ロード後処理
namespace serial{
	template<class Archive>
```

```
	struct afterLoad<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			//バージョン1.2未満ならセーブデータに m_b がないので直接値を代入
			if (ver < CVersion(1, 2))//バージョン判定
			{
				obj.m_b = 1.23f;//初期値代入
			}

			//読み込みの結果、 m_strP が空文字列なら nullptr にする
			if(obj.m_strP[0] == '¥0')
			{
				delete obj.m_strP;//削除
				obj.m_strP = nullptr;
			}
		}
	};
}
```

● 通知処理①：セーブデータにないデータ

「明示的なバージョン判定に頼らず、古いセーブデータに存在しないデータがあったら初期値をセットしたい」という処理を行うには、そのようなデータの通知を受け取る処理を定義する。

【セーブデータに存在しないデータ項目の通知】

```
//保存先の指定があるが、セーブデータになくロードできなかったデータ項目の通知
namespace serial{
	template<class Archive>
	struct noticeUnloadedItem<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
			, const CItemBase& unloaded_item)
		{	//↑読み込めなかったデータ項目が渡される
			//data2(m_b)はバージョン1.2未満のセーブデータには存在しない
			if (unloaded_item == "data2")//引数 unloaded_item で問題の項目をチェック
			{
				obj.m_b = 123;//初期値代入
			}
		}
	};
}
```

● 通知処理②：セーブデータにしかないデータ

「データ名を変更したが、古いセーブデータの古い名前で読み込みたい」という処理を行うには、そのようなデータの通知を受け取る処理を定義する。

【セーブデータにしか存在しないデータ項目の通知】

```
//セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理
namespace serial{
	template<class Archive>
	struct noticeUnrecognizedItem<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
			, const CItemBase& unrecognized_item)
		{	//↑読み込めなかったデータ項目が渡される
			//data_x は現在 data3(m_c)に改名
			if (unrecognized_item == "data_x")//引数 unrecognized_item で問題の項目をチェック
			{
```

```
                        arc & pair("data3", obj.m_c);//読み込み先を指定し直して再実行
                        //「&」演算子で改めてシリアライズ対象項目を指定し直すことで
                        //データの読み込み先を委譲する

                        //この関数内ではobjの値を更新してもよい
                        obj.m_preblemItem = unrecognized_item.m_nameCrc;//問題があった項目を記録
                }
        }
    };
}
```

● コレクターとディストリビュータ

クラス（構造体）のメンバーではないデータをセーブデータにひとまとめにするには、「コレクター」と「ディストリビュータ」をセットで定義する。

【コレクター】

```
//コレクター
template<class Arc>
struct collector<Arc, USER_DATA> {
    void operator()(Arc& arc, const USER_DATA& obj, const CVersion& ver)
    {
        //USER_DATAの後に連続して下記のデータもいっしょにシリアライズする

        //インベントリのデータを収集
        CSingleton<CInventory> inventory;
        for (auto item_data : *inventory)//ループ処理で全要素を取得
        {
            //アイテムデータを一つずつシリアライズ
            arc << pair("item", *item_data);
        }
        //進行＆フラグデータをシリアライズ
        CSingleton<CPhaseAndFlags> phase_and_flags;
        arc << pair("phase_and_flags", *phase_and_flags);
    }
};
```

【ディストリビュート前処理】

```
//ディストリビュート前処理
template<class Arc>
struct beforeDistribute<Arc, USER_DATA> {
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
                //↑objが配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡さてくる
    {
        //データを読み込む前に、読み込み先を一旦空にする

        //インベントリデータクリア
        CSingleton<CInventory> inventory;
        inventory.destroy();
        //フェーズ＆進行データクリア
        CSingleton<CPhaseAndFlags> phase_and_flags;
        phase_and_flags.destroy();
    }
};
```

【ディストリビュータ】

```
//ディストリビュータ
template<class Arc>
struct distributor<Arc, USER_DATA> {//↓USER_DATAにconstが付かない
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
```

```
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
                                                                const CItemBase& target_item)
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡される（ディストリビュート前処理と同じ
                                                                //↑対象データ項目が渡される
    {
        //USER_DATA の後に連続して下記のデータもいっしょにデシリアライズされる

        //コレクターでシリアライズしたデータの数だけ繰り返し実行されるので、
        //データ項目名を判定して一つずつ処理する
        //※無視したらデータは読み込まれずに次のデータに進む

        if (target_item == "item")//インベントリデータか？
        {
            //インベントリデータ復元
            CSingleton<CInventory> inventory;
            ITEM_DATA item_data;
            arc >> pair("item", item_data);//デシリアライズ
            inventory->regist(item_data);//インベントリにデータを登録
        }
        else if (target_item == "phase_and_flags")//進行データか？
        {
            //フェーズ＆進行データ復縁
            CSingleton<CPhaseAndFlags> phase_and_flags;
            arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
        }
    }
};
```

【ディストリビュート後処理】

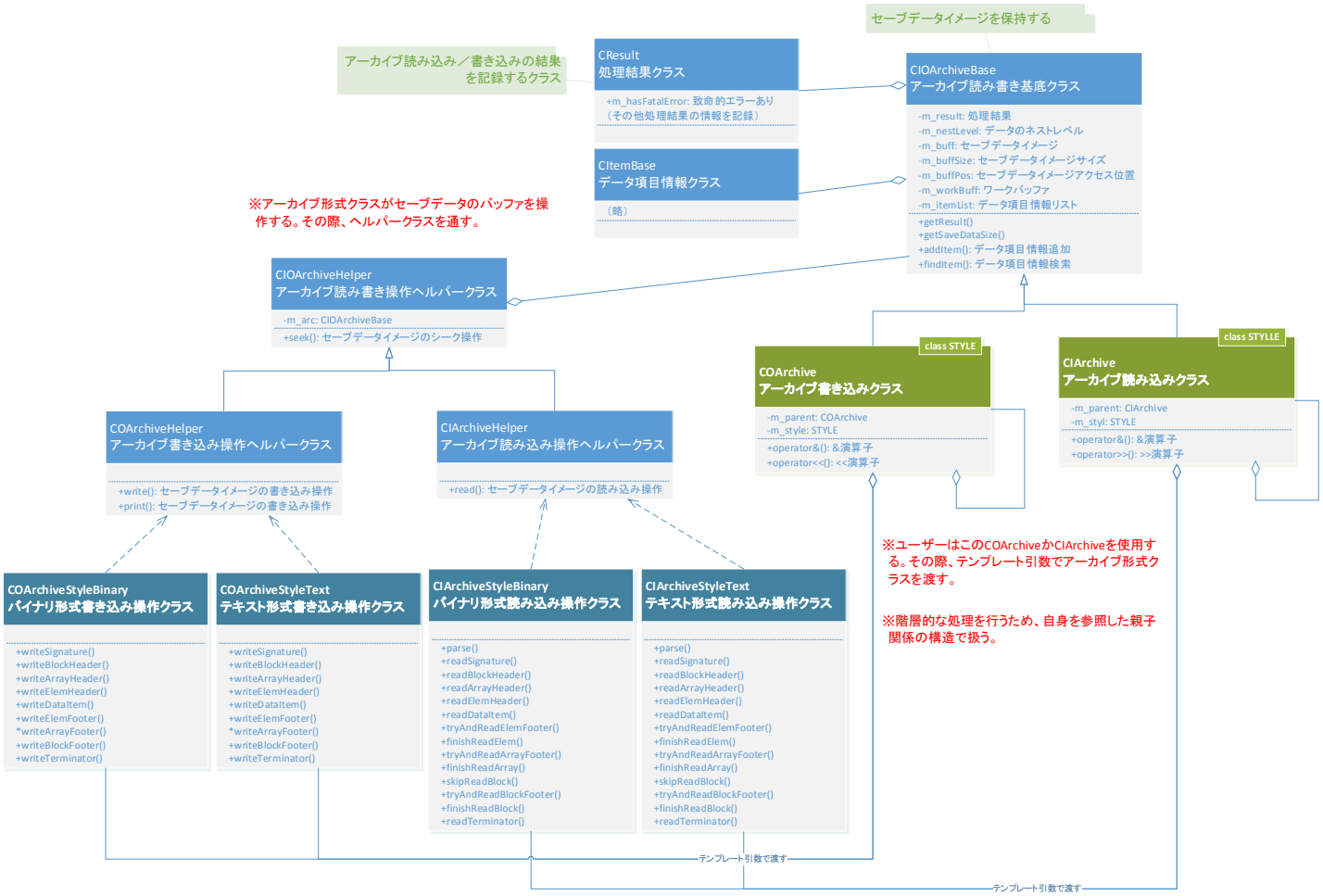
```
//ディストリビュート後処理
template<class Arc>
struct afterDistribute<Arc, USER_DATA> {//↓USER_DATA に const が付かない
    void operator()(Arc& arc, USER_DATA& obj, const std::size_t array_num_on_save_data,
                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
                //↑obj が配列だった場合、セーブデータ上の要素数（前者）と実際に読み込んだ要素数（後者）が
                //  渡される（ディストリビュート前処理と同じ
    {
        //全ての読み込みが完了
        //インベントリデータが更新されたので、キャラが所持する武器データの参照を再設定する
        CSingleton<CCharaList> chara_list;
        for (auto& chara_data : *chara_list)//ループ処理で全キャラを取得
        {
            chara_data->attachItems();//アイテムを参照し直す
        }
    }
};
```

なお、これらの処理を工夫することで、データの部分ロードに対応することができる。例えば、「『セーブデータ A』をロード後、インベントリだけ『セーブデータ B』の状態にする」といった制御が可能。詳しい方法は要件定義で説明。

▼ クラス設計

本システムのクラス図を示す。

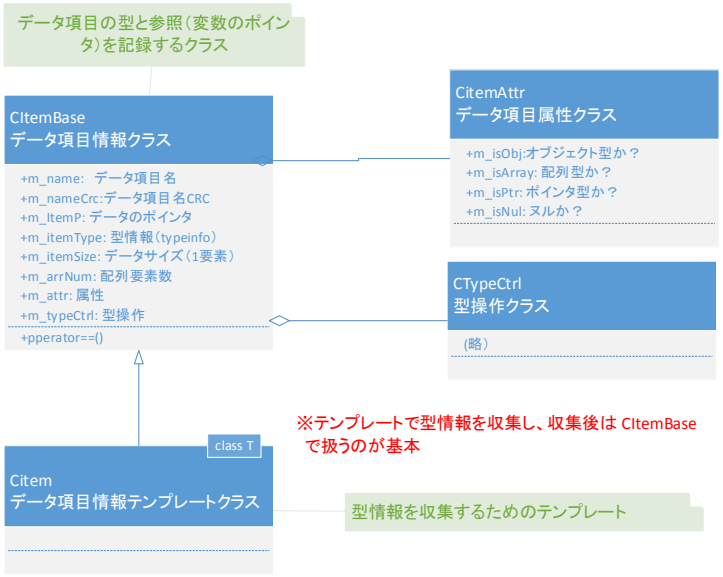
アーカイブクラス：


セーブデータイメージを保持する
アーカイブ読み込み／書き込みの結果を記録するクラス
CResult
処理結果クラス
+m_hasFatalError: 致命的エラーあり（その他処理結果の情報を記録）
CIOArchiveBase
アーカイブ読み書き基底クラス
-m_result: 処理結果
-m_nestLevel: データのネストレベル
-m_buff: セーブデータイメージ
-m_buffSize: セーブデータイメージサイズ
-m_buffPos: セーブデータイメージアクセス位置
-m_workBuff: ワークバッファ
-m_itemList: データ項目情報リスト
+getResult()
+getSaveDataSize()
+addItem(): データ項目情報追加
+findItem(): データ項目情報検索
CItemBase
データ項目情報クラス
（略）
※アーカイブ形式クラスがセーブデータのバッファを操作する。その際、ヘルパークラスを通す。
CIOArchiveHelper
アーカイブ読み書き操作ヘルパークラス
-m_arc: CIOArchiveBase
+seek(): セーブデータイメージのシーク操作
class STYLE
COArchive
アーカイブ書き込みクラス
-m_parent: COArchive
-m_style: STYLE
+operator&(): &演算子
+operator<<(): <<演算子
class STYLLE
CIArchive
アーカイブ読み込みクラス
-m_parent: CIArchive
-m_styl: STYLE
+operator&(): &演算子
+operator>>(): >>演算子
COArchiveHelper
アーカイブ書き込み操作ヘルパークラス
+write(): セーブデータイメージの書き込み操作
+print(): セーブデータイメージの書き込み操作
CIArchiveHelper
アーカイブ読み込み操作ヘルパークラス
+read(): セーブデータイメージの読み込み操作
※ユーザーはこのCOArchiveかCIArchiveを使用する。その際、テンプレート引数でアーカイブ形式クラスを渡す。
※階層的な処理を行うため、自身を参照した親子関係の構造で扱う。
COArchiveStyleBinary
バイナリ形式書き込み操作クラス
+writeSignature()
+writeBlockHeader()
+writeArrayHeader()
+writeElemHeader()
+writeDataItem()
+writeElemFooter()
*writeArrayFooter()
+writeBlockFooter()
+writeTerminator()
COArchiveStyleText
テキスト形式書き込み操作クラス
+writeSignature()
+writeBlockHeader()
+writeArrayHeader()
+writeElemHeader()
+writeDataItem()
+writeElemFooter()
*writeArrayFooter()
+writeBlockFooter()
+writeTerminator()
CIArchiveStyleBinary
バイナリ形式読み込み操作クラス
+parse()
+readSignature()
+readBlockHeader()
+readArrayHeader()
+readElemHeader()
+readDataItem()
+tryAndReadElemFooter()
+finishReadElem()
+tryAndReadArrayFooter()
+finishReadArray()
+skipReadBlock()
+tryAndReadBlockFooter()
+finishReadBlock()
+readTerminator()
CIArchiveStyleText
テキスト形式読み込み操作クラス
+parse()
+readSignature()
+readBlockHeader()
+readArrayHeader()
+readElemHeader()
+readDataItem()
+tryAndReadElemFooter()
+finishReadElem()
+tryAndReadArrayFooter()
+finishReadArray()
+skipReadBlock()
+tryAndReadBlockFooter()
+finishReadBlock()
+readTerminator()
テンプレート引数で渡す
テンプレート引数で渡す

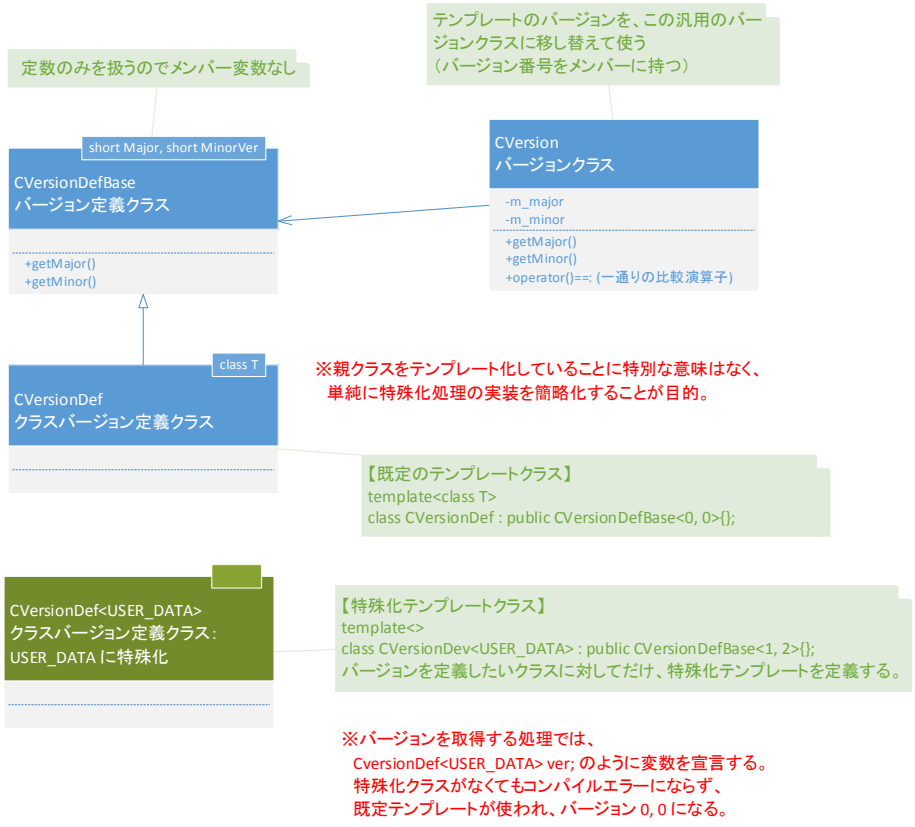
シリアライズクラス：


シリアライズ系
class Arc, class T
save
シリアライズ専用関数オブジェクト
+operator()
class Arc, class T
collector
コレクター関数オブジェクト
+operator()
※すべて既定で空の処理が定義されており、
特殊化クラスが定義されない限りはそれが実
行される。
【既定のテンプレートクラス】
template<class Arc, class T>
struct serialize
シリアライズ／デシリアライズ共通
class Arc, class T
serialize
シリアライズ／デシリアライズ共通
関数オブジェクト
+operator()
class Arc
serialize<USER_DATA>
シリアライズ／デシリアライズ共通
関数オブジェクト:USER_DATAに特殊化
+operator()
【特殊化テンプレートクラス】
template<class Arc>
struct serialize<USER_DATA>
※Arcはアーカイブクラスの指定なので特殊化しない
デシリアライズ系
class Arc, class T
load
デシリアライズ専用関数オブジェクト
+operator()
class Arc, class T
distributor
ディストリビュータ関数オブジェクト
+operator()
class Arc, class T
beforeLoad
ロード前処理関数オブジェクト
+operator()
class Arc, class T
beforeDistribute
ディストリビュート前処理
関数オブジェクト
+operator()
class Arc, class T
afterLoad
ロード後処理関数オブジェクト
+operator()
class Arc, class T
afterDistribute
ディストリビュート後処理
関数オブジェクト
+operator()
class Arc, class T
noticeUnrecognizedItem
セーブデータのみのデータ通知
関数オブジェクト
+operator()
class Arc, class T
noticeUnloadedItem
セーブデータにないデータ通知
関数オブジェクト
+operator()

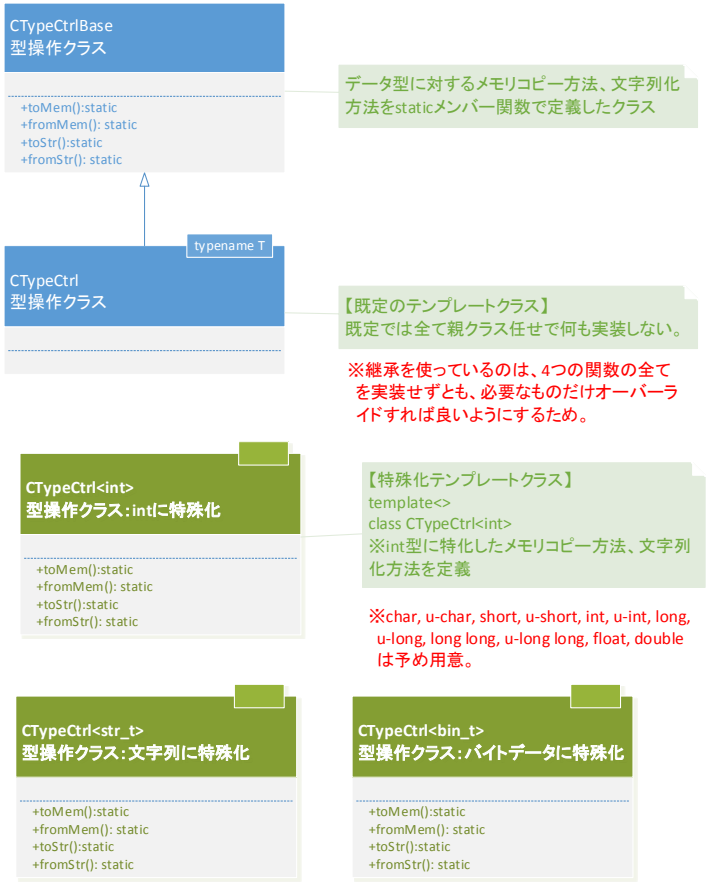
【基本クラス】データ項目情報クラス：



【基本クラス】バージョンクラス：



【基本クラス】型操作クラス：



※特殊化のためだけの構造体 str_t, bin_t を用意し、文字列とバイトデータのための専用処理も定義する。
※同様の手法で、ユーザー定義のデータ型に対する操作方法を追加することも可能。

▼ テンプレートオペレータ

クラス図では明示していないが、アーカイブ書き込み／読み込みクラスのオペレータは、テンプレート関数として定義することで、特殊化を活用した処理を可能とする。

【アーカイブ書き込みクラスのオペレータ】

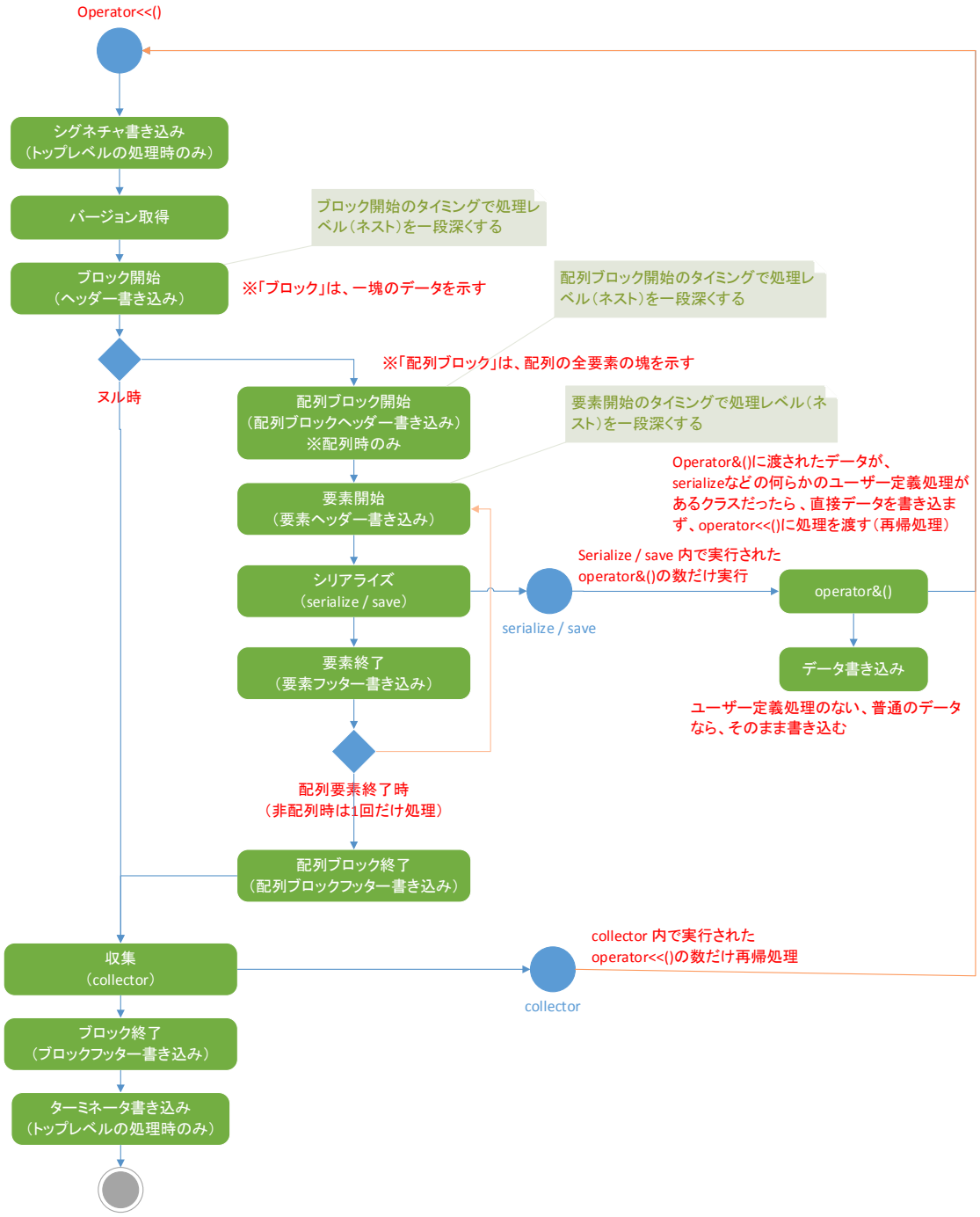
```
//--------------------
//アーカイブ書き込みクラス
template<class STYLE>
class COArchive : public CIOArchiveBase
{
public:
	//「&」オペレータ
	template<class T>
	COArchive& operator&(const CItem<T> item_obj);
	//「<<」オペレータ
	template<class T>
	COArchive& operator<<(const CItem<T> item_obj);
};
```

【アーカイブ読み込みクラスのオペレータ】

```
//--------------------
//アーカイブ読み込みクラス
template<class STYLE>
class CIArchive : public CIOArchiveBase
{
public:
	//「&」オペレータ
	template<class T>
	CIArchive& operator&(const CItem<T> item_obj);
	//「>>」オペレータ
	template<class T>
	CIArchive& operator>>(CItem<T> item_obj_now);//データの書き換えを行うので const が付かない
};
```

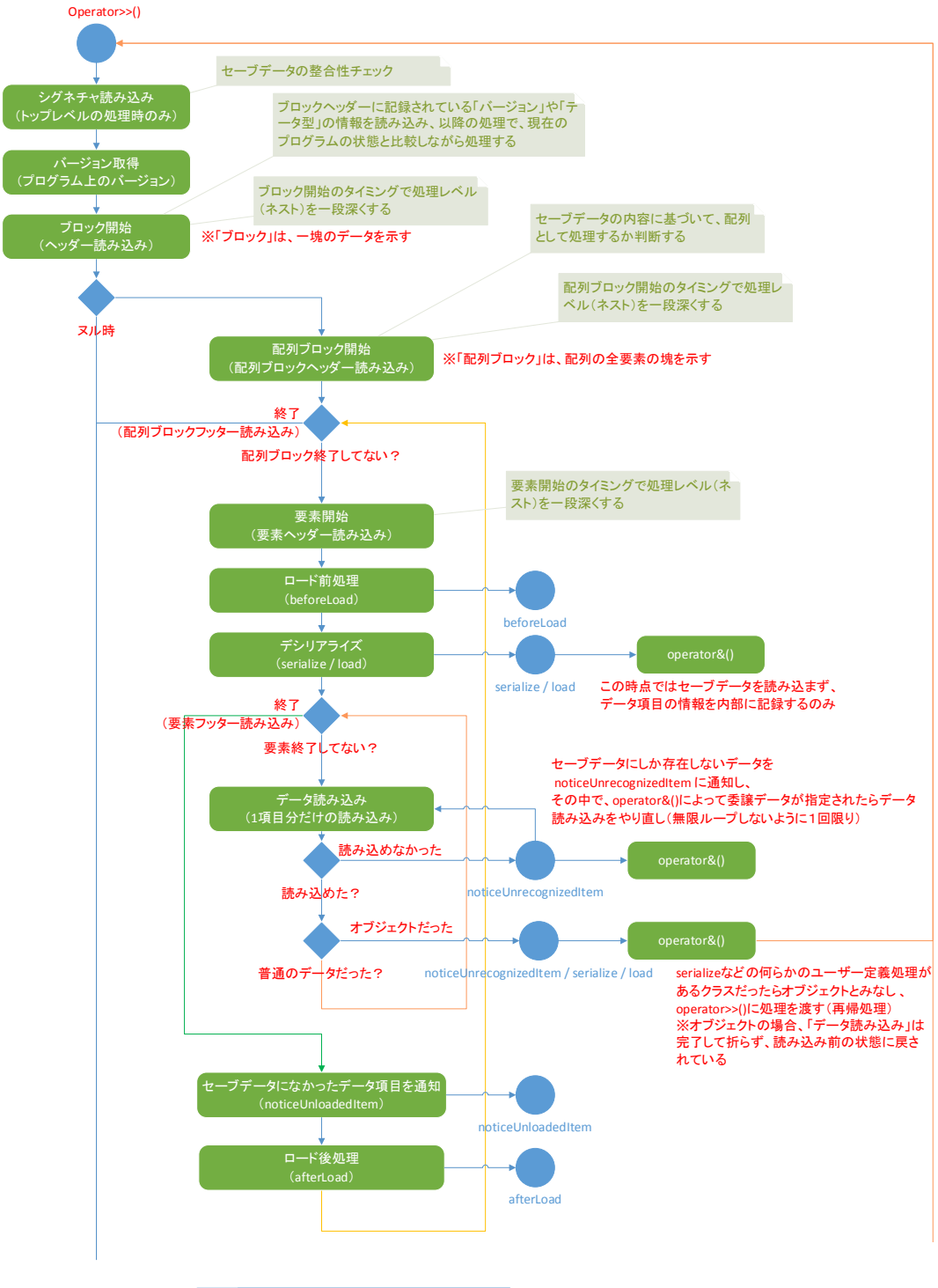
▼ 処理フロー：シリアライズ

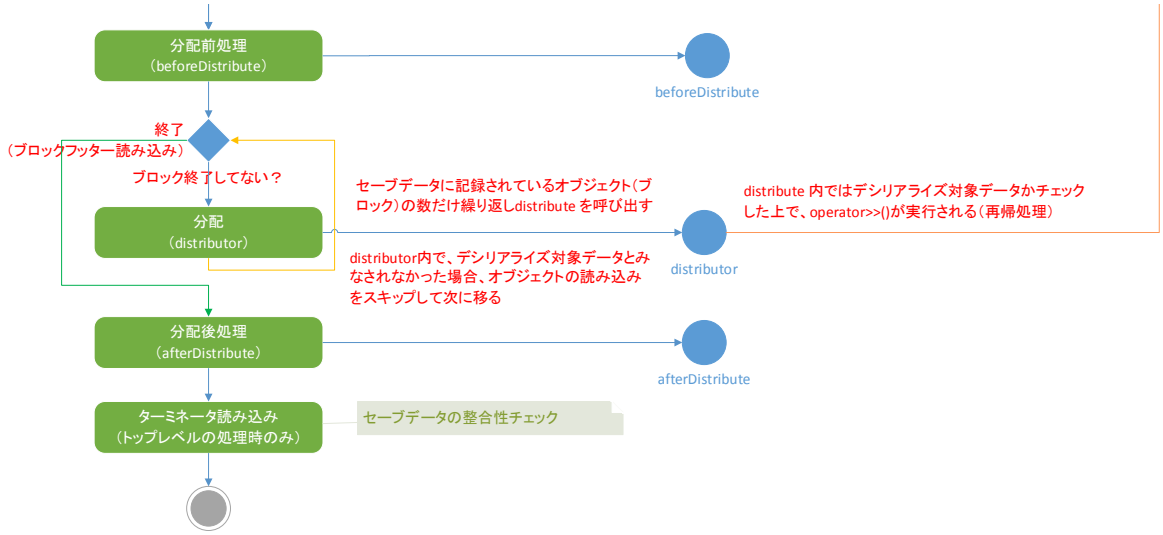
シリアライズ時の処理フローを示す。



▼ 処理フロー：デシリアライズ

デシリアライズ時の処理フローを示す。





▼ シリアライズの特殊な仕組み：ネストした構造体

フローに示したとおり、シリアライズ時には再帰処理を行う。これは、ネストした構造体を処理するためである。

下記のようなシリアライズが定義された場合に処理する。

【ネストした構造体の例】

```
//シリアライズ対象のクラス／構造体
struct USER_DATA
{
	int m_a
	float m_b;
	char m_c[3];
	struct SUB
	{
		int m_x;
		float m_y;
		short m_z[5];
	} m_sub;//サブ構造体のメンバー
};
```

【シリアライズの定義】

```
//シリアライズ＆デシリアライズ共通処理
namespace serial {
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
			arc & pair("sub", obj.m_sub);//←ネストした構造体：普通に & 演算子に構造体のメンバーを渡す
		}
	};
};
```

【ネストした構造体のシリアライズの定義】

```
//シリアライズ＆デシリアライズ共通処理
namespace serial {
```

```
	template<class Archive>
	struct serialize<Archive, USER_DATA::SUB> {
		void operator()(Archive& arc, const USER_DATA::SUB& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("x", obj.m_x) & pair("y", obj.m_y) & pair("z", obj.m_z);
		}
	};
};
```

▼ デシリアライズの特殊な仕組み①：データ項目の不一致に対応した読み込み

フローに示したとおり、デシリアライズ時の処理はシリアライズに比べてやや複雑になる。

逐一データの終わりを検出しながら処理することで、プログラムとセーブデータでデータ項目が一致しない場合でも、それ以外のデータは問題なく読み込みが可能。

▼ デシリアライズの特殊な仕組み②：定義順序に依存しない読み込み

フローに示したとおり、デシリアライズ時の処理は「シリアライズ／デシリアライズ共通処理」および「ロード処理」を呼び出したタイミングでデータを読み込まない。シリアライズ対象項目として内部のリストに記録するだけにとどめ、以後、実際の読み込み処理の際に、該当するデータ項目をリストから照合して、読み込み先を割り当てる。

これにより、プログラムとセーブデータでデータ項目の定義順序が一致しない場合でも、正常に読み込みが可能。

▼ デシリアライズの特殊な仕組み③：ネストした構造体

前述の「シリアライズの特殊な仕組み：ネストした構造体」によって記録されたデータを読み込む場合は、シリアライズと同様の再帰処理を用いる。

ただし、デシリアライズでは、ユーザー定義処理（serialize / load）と読み込み処理が分かれているため、再帰処理を呼び出したいタイミングでは「型」が分からなくなっている。つまり、テンプレートオペレータ「`template<typename T> operator>>()`」に「型 T」を与えることができない。

この対処として、ネストした構造体を読み込むときに、ユーザー定義処理（serialize / load）をもう一度実行する。すると、再度「`template<typename T> operator&()`」の処理が一通り実行されるが、この時は、本来の挙動（シリアライズ対象データ項目の記録）ではなく、再帰処理（「`operator>>()`」の呼び出し）を行うようにする。

このような振る舞いの変更は、アーカイブクラス内でステート管理するだけで十分対処できる。なお、無関係なデータ項目も渡されてくるので、無視しなければならない。

注意点として、ユーザー定義処理の作りに気を付けなければならない。

例えば、内部に処理カウンタを保持して、実行のたびに挙動が変わるような処理が組まれていると、この仕組みが意図通りに動作しなくなってしまう。

▼ デシリアライズの特殊な仕組み④：セーブデータにしかないデータの委譲

セーブデータにしか存在しないデータを再度読み直す仕組みを用意するが、これもステート管理によって、「operator&()」の振る舞いを変えるようにすることで実現できる。

ユーザー定義処理「noticeUnrecognizedItem()」内では、新しい読み込み先が「&」演算子で指定される。

【セーブデータにしか存在しないデータ項目の通知】

```
//セーブデータにはあったが、保存先の指定がなく、ロードできなかったデータ項目の通知処理
namespace serial{
	template<class Archive>
	struct noticeUnrecognizedItem<Archive, USER_DATA> {
		void operator()(Archive& arc, USER_DATA& obj, const CVersion& ver, const CVersion& now_ver
		                                              , const CItemBase& unrecognized_item)
		{                                             //↑読み込めなかったデータ項目が渡される
			//data_x は現在 data3(m_c)に改名
			if (unrecognized_item == "data_x")//引数 unrecognized_item で問題の項目をチェック
			{
				arc & pair("data3", obj.m_c);//読み込み先を指定し直して再実行
			}
		}
	};
}
```

この処理を「読み込み先の委譲」とする。

委譲を実現するには、読み込み処理をリトライした際に、シリアライズ対象項目リストの照会を行わず、移譲先をそのまま使用するように挙動を変えなければならない。

▼ SFINAE を利用したユーザー定義処理実装判定方法

「serialize」などのユーザー定義処理が定義されているかどうかを判別することで、オブジェクト（ブロック）として処理すべきデータ項目かどうかを判定する。

それには特殊化したテンプレートクラスが定義されているかどうかの判定が必要になるが、この処理のために「SFINAE」と呼ばれる動作を利用できる。

「SFINAE」とは、「Substitusion Failure Is Not An Error」（置き換え失敗はエラーではない）のことで、C++のテンプレートクラスの仕様である。別紙の「効果的なテンプレートテクニック」でも説明している。

具体的には、下記のような処理でその判定を行うことができる。

【基本テンプレートクラスとしての serialize クラスの定義】

```
//シリアライズ＆デシリアライズ共通処理
```

```
namespace serial {
	template<class Arc, class T>
	struct serialize {
		typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
		void operator()(Arc& arc, const T& obj, const CVersion& ver, const CVersion& now_ver)
		{}
	};
}
```

【ユーザー定義処理として特殊化した serialize クラスの定義】

```
//シリアライズ＆デシリアライズ共通処理
namespace serial {
	template<class Archive>
	struct serialize<Archive, USER_DATA> {
		//IS_UNDEFINED を定義しない（してはダメ）
		void operator()(Archive& arc, const USER_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("data1", obj.m_a);
			arc & pair("data2", obj.m_b) & pair("data3", obj.m_c);
		}
	};
};
```

【ユーザー定義処理実装判定関数】

```
//--------------------
//ユーザー定義処理クラス定義済みチェック関数
//※SFINAE により、IS_UNDEFINED が定義されている型のオーバーロード関数が選ばれたら未定義とみなす
namespace serial {
	template<class F>
	bool isDefinedFunctor(const typename F::IS_UNDEFINED)
	{
		return false;//未定義
	}
	template<class F>
	bool isDefinedFunctor(...)
	{
		return true;//定義済み
	}
}
```

【ユーザー定義処理実装判定】

```
if(isDefinedFunctor<serialize<ArchiveClass, USER_DATA> >(0) == true)
	//ユーザー定義処理が定義されている

if(isDefinedFunctor<serialize<ArchiveClass, OTHER_DATA> >(0) == true)
	//ユーザー定義処理が定義されていない
```

「OTHER_DATA」クラスは、特殊化テンプレートクラスが定義されていないので、「serialize<ArchiveClass, OTHER_DATA>」のインスタンス化には基本テンプレートが適用される。その結果「typedef IS_UNDEFINED」が存在するため、「isDefinedFunctor()の実行時は、「isDefinedFunctor(const typename F::IS_UNDEFINED)」の方が実行される。

特殊化テンプレートクラスが定義されている「USER_DATA」には「typedef IS_UNDEFINED」が定義されていないので、テンプレート関数「isDefinedFunctor(const typename F::IS_UNDEFINED)」をインスタンス化しようとするとコンパイルエラーになる。しかし、もう一つ、コンパイルエラーにならないオーバーロード関数「isDefinedFunctor(...)」があるため、コンパイラは問題がないほうを選択する。

このように、コンパイルエラーを回避できるなら方法があるならそれで済ませるのが、

「SFINAE」の仕組みである。

　このように、本システムでは特殊化クラスの実装状態を判定するために、SFINAE を活用する。

■ データ仕様

▼ シリアライズデータの基本構造

　アーカイブ形式がバイナリかテキストかによらず、シリアライズデータは下記のの構造と情報を扱う。

基本構造：

≪基本構造≫

シグネチャ	任意の識別子:正しいセーブデータかの判定用
ブロック始端	任意の識別子＋データ名CRC＋属性＋バージョン:オブジェクトの情報を扱う
配列ブロック始端	任意の識別子(＋配列要素数):配列ブロックの開始宣言
要素始端	任意の識別子(＋データ項目数):要素の開始宣言
データ始端	任意の識別子＋データ名CRC＋属性(＋データサイズ＋配列要素数):データの情報を扱う
データ	実際のデータ ※配列なら配列数分連続する
データ終端	任意の識別子:データの終端判定用
要素終端	任意の識別子:要素の終端判定用
配列ブロック終端	任意の識別子:配列ブロックの終端判定用
ブロック終端	任意の識別子:ブロックの終端判定用
ターミネータ	任意の識別子:セーブデータが正しく読み込めたかの判定用

≪属性≫

isNul	オブジェクト型か？
isArr	配列型か？
isPtr	ポインタ型か？
isNul	ヌルか？
isVLen	可変長か？ ※文字列は可変長で扱う
hasVersion	バージョンがあるか？

≪バイナリ形式の場合≫

シグネチャ	0x00, 0xff + "serial:Cbin" + 0xff, 0x00
ブロック始端	0xfb, 0xff + データ名CRC + 属性 + バージョン + ブロックのサイズ（始端と終端分除く）
配列ブロック始端	0xfa, 0xff + 配列要素数 + 配列ブロックのサイズ（始端と終端分除く） ※配列がある時のみ
要素始端	0xfe, 0xff + データ項目数 + 要素のサイズ（始端と終端分除く）
データ始端	0xfd, 0xff + 名前CRC + 属性 + データサイズ（1要素あたりのサイズ） + 配列要素数（配列がある時のみ）
データ	***
データ終端	0xff, 0xfd
要素終端	0xff, 0xfe
配列ブロック終端	0xff, 0xfa ※配列がある時のみ
ブロック終端	0xff, 0xfb
ターミネータ	0xff, 0x00 + "serial:Cbin" + 0x00, 0xff

≪テキスト形式の場合≫

シグネチャ	{"serializer": [
ブロック始端	"データ名": [{"crc": 0x00000000, "isObj": 1, "isArr": 0, "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVersion": 0, "ver": "0.0",
配列ブロック始端	"arrayNum": 2, "array": [※配列がある時のみ
要素始端	"elem": { ※配列がある時は [のみ ("elem":がつかない)
データ始端	"データ名": {"crc": 0x00000000, "itemSize": 1, "isObj": 0, "isArr": 1, "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "arrNum": 0, "data":
データ	*** ※配列やバイトデータなら ***, ***, *** とカンマ区切りで複数のデータが連続する
データ終端	]
要素終端	}
配列ブロック終端	] ※配列がある時のみ
ブロック終端	}
ターミネータ	], "terminator": "ok"}

　なお、最低限ここにあげた情報が読み取れるのであれば、扱うアーカイブ形式に応じて任意の情報を追加してもよい。

実際の入り組んだ構造のイメージ：

≪実際の構造のイメージ≫

シグネチャ	
ブロック始端	"UserData"
配列ブロック始端	[2]
要素始端	(配列要素=0)
データ始端	"data2"
データ	(4バイトデータ)
データ終端	
データ始端	"data2"
データ	(4バイトデータ)
データ終端	
データ始端	"data3"[3]
データ	(3バイトデータ)
データ終端	
ブロック始端	"sub"
要素始端	
データ始端	"x"
データ	(4バイトデータ)
データ終端	
データ始端	"y"
データ	(4バイトデータ)
データ終端	
データ始端	"z"[5]
データ	(10バイトデータ)
データ終端	
要素終端	
ブロック終端	
要素終端	
要素始端	(配列要素=1)
データ始端	"data2"
データ	(4バイトデータ)
データ終端	
データ始端	"data2"
データ	(4バイトデータ)
データ終端	
データ始端	"data3"
データ	(3バイトデータ)
データ終端	
ブロック始端	"sub"
要素始端	
データ始端	"x"
データ	(4バイトデータ)
データ終端	
データ始端	"y"
データ	(4バイトデータ)
データ終端	
データ始端	"z"
データ	(10バイトデータ)
データ終端	
要素終端	
ブロック終端	
要素終端	
配列ブロック終端	
ブロック始端	"UserDataEx1"
配列ブロック始端	[2]
要素始端	(配列要素=0)
データ始端	"data"
データ	(4バイトデータ)
データ終端	
要素終端	
要素始端	(配列要素=1)
データ始端	"data"
データ	(4バイトデータ)
データ終端	
要素終端	
配列ブロック終端	
ブロック終端	
ブロック始端	"UserDataEx2"
要素始端	
データ始端	"data1"
データ	(1バイトデータ)
データ終端	
データ始端	"data2"
データ	(2バイトデータ)
データ終端	
データ始端	"data3"
データ	(8バイトデータ)
データ終端	
要素終端	
ブロック終端	
ブロック終端	
ターミネータ	

シリアライズする構造体イメージ:コメント部はデータ名

```
struct USER_DATA
{
   int m_a;//"data1"
   float m_b;//"data2"
   char m_c[3];//"data3"
   struct SUB
   {
      int m_x;//"x"
      float m_y;//"y"
      short m_z[5];//"z"
   } m_sub;//"sub"
};
USER_DATA user_data[2];//"UserData"
```

collector で追加する構造体①

```
struct USER_DATA_EX1
{
   int m_data;//"data"
};
USER_DATA_EX1 user_data_ex1[2];//"UserDataEx1"
```

collector で追加する構造体②

```
struct USER_DATA_EX2
{
   char m_data1;//"data1"
   short m_data2;//"data2"
   double m_data3;//"data3"

};
USER_DATA_EX2 user_data_ex2;//"UserDataEx2"
```

■ 処理実装サンプル：シリアライズの使用サンプル

実際のシリアライズ処理を使用したプログラムのサンプルを示す。

実際のゲーム開発での利用をイメージし易いように、まずはゲームのデータ管理システムのサンプルを実装する。

▼ 【準備】ゲームのデータ管理システムのサンプル

まずはセーブデータを意識せず、下記の要件でちょっとしたデータ管理システムを構築する。

- 実際にゲームに使用されるものに近い構造とする。
- インベントリデータとして、複数の所持アイテムを扱うものとする。
- 複数のキャラデータを扱うものとする。
- キャラデータには装備武器と盾を含み、インベントリデータを直接参照するポインタを持つものとする。
- アビリティデータを扱うものとする。
- 各キャラは不定数のアビリティを習得し、連結リストでつないで管理するものとする。
- 進行データとして、フェーズとフラグを管理するものとする。
  - フェーズは int 型、フラグは std::bitset<64>で扱うものとする。
- アイテム、キャラ、アビリティの一部のパラメータは共通構造体で扱うものとする。
  - 攻撃力と防御力を扱うものとする。
- アイテム、キャラ、アビリティデータは、それぞれユニーク ID を文字列管理するものとする。
  - CRC 値を保持し、高速な検索に対応するものとする。
- 本来のデータ管理システムでは、静的なデータテーブルと動的なデータを区別して扱うが、このサンプルでは全て動的なデータとして扱う。
  - 内容的には両者が混ざったようなものとし、アイテム名やキャラ名なども扱う。
  - また、本来アイテムのマスターデータとインベントリ（所持アイテム）を区別して扱うが、このサンプルではインベントリがあるのみとする。
- インベントリデータ、キャラデータ、アビリティデータ、キャラ習得アビリティは、それぞれコレクションを構成して管理するものとする。
  - コレクションクラスにより、データの検索やイテレータの取得に対応する。
  - コレクションはプールアロケータを内包し、上限の決まったインスタンスのバッファを管理するものとする。
  - コレクションクラスは汎用クラスとして作成し、それぞれのデータ向けのコレクショ

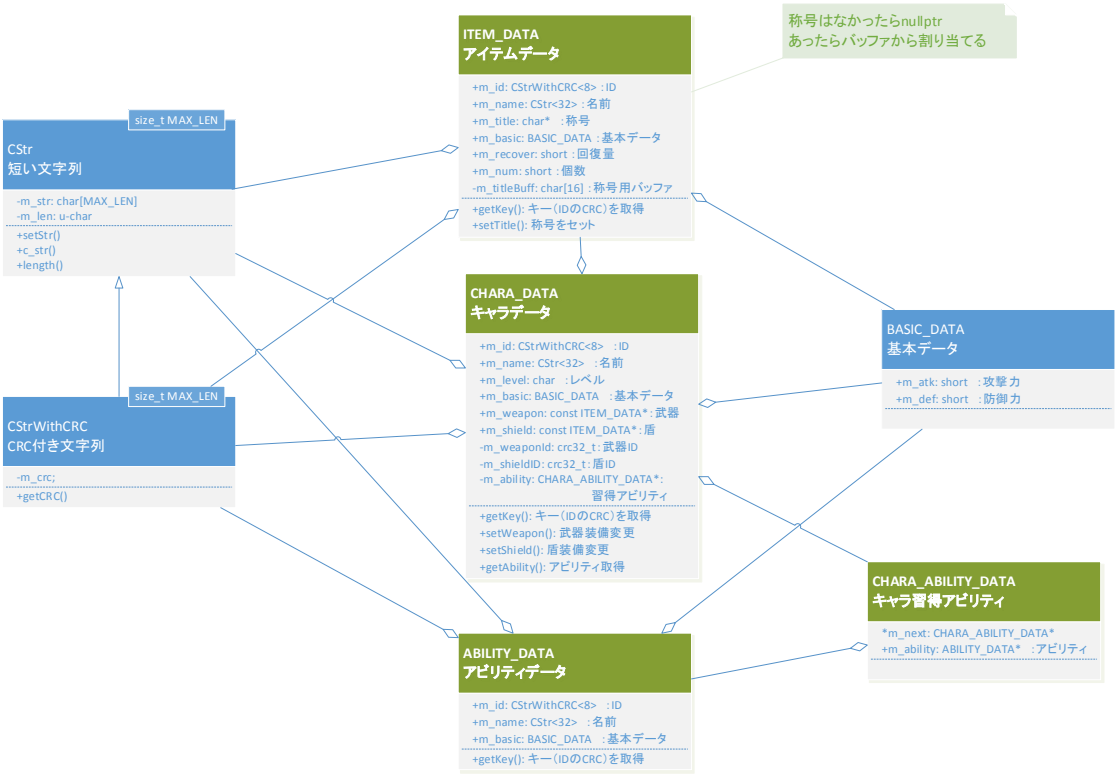
ンを個別に作成する必要がないものとする。

・ コレクションクラス、および、進行データはシングルトンで扱うものとする。

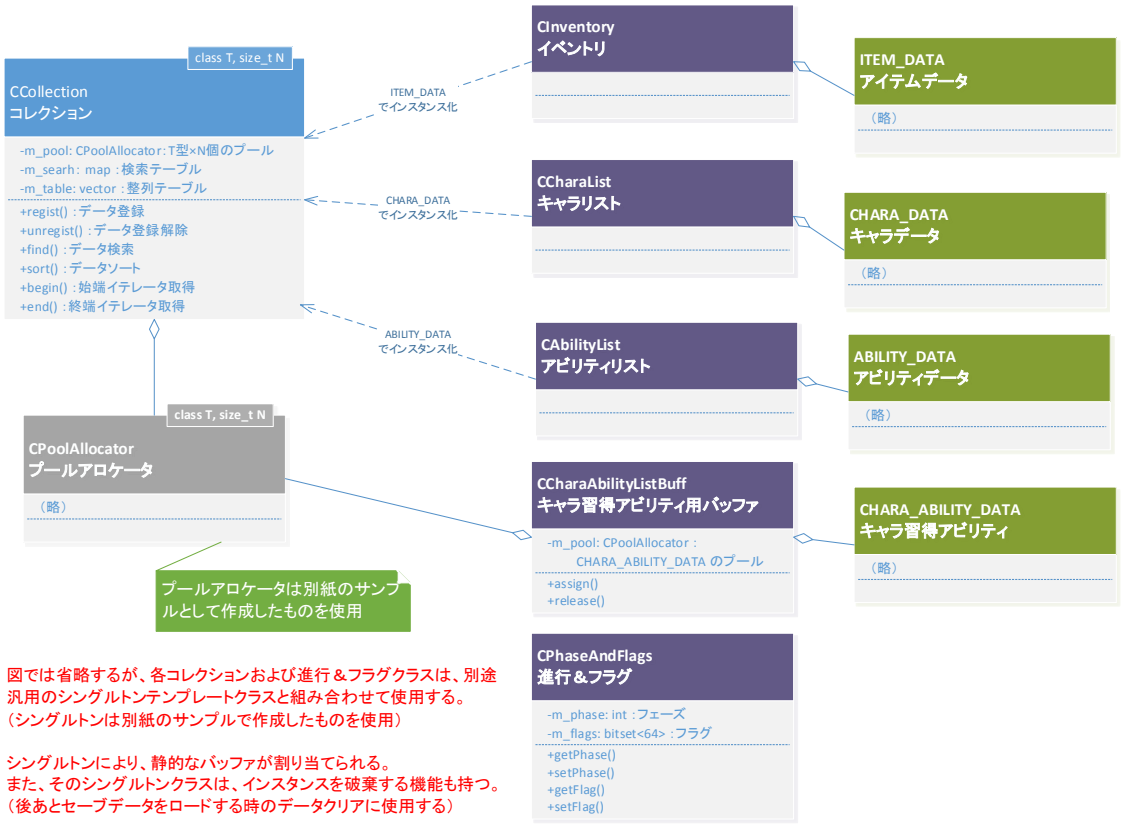
▼ データ管理システムのクラス図

上記の要件に基づくクラス図を示す。

構造体：



コレクションと進行データ：



▼ サンプルプログラムについて

サンプルプログラムは、（非常に便利な）C++11 の新しい構文やライブラリを随所で使用している。

また、別紙に掲載したサンプルコードも流用している。別紙から引用する必要のあるコードについては後述する。

なお、Visual C++ 2013 と Cygwin + GCC 4.8.2 での動作を確認しているコードである。

▼ データ管理システムのサンプルプログラム：定義部

まず、クラス図で示した各クラス／構造体のサンプルを示す。

【インクルード】

```
#include <map>//STL map 用
#include <vector>//STL vector 用

//その他、別紙のサンプルで作成したクラス・関数を使用
// crc32_t calcCRC32(const char* str);//CRC32 計算関数
```

```
//CPoolAllocatorWithType//プールアロケータ
//CStackAllocatorWithBuff//スタックアロケータ
//CTempPolyStackAllocator//多態アロケータ
```

【短い文字列型】

```
//--------------------
//短い文字列型
template <std::size_t S>
class CStr
{
public:
	//定数
	static const std::size_t BUFF_SIZE = S;//バッファサイズ
	static const std::size_t MAX_SIZE = S - 1;//最大文字列長
public:
	//アクセッサ
	const char* c_str() const { return m_str; }//文字列を取得　※std::string互換
	std::size_t length() const { return m_len; }//文字列長を取得　※std::string互換
	std::size_t size() const { return m_len; }//文字列長を取得　※std::string互換
	std::size_t max_size() const { return MAX_SIZE; }//文字列の最大長を取得　※std::string互換
	const char* getStr() const { return m_str; }//文字列を取得
	void setStr(const char* str)//文字列をセット
	{
		if (!str)
		{
			m_str[0] = '¥0';
			m_len = 0;
			return;
		}
	#ifdef USE_STRCPY_S
		strncpy_s(m_str, sizeof(m_str), str, MAX_SIZE);
	#else//USE_STRCPY_S
		strncpy(m_str, str, MAX_SIZE);
	#endif//USE_STRCPY_S
		m_str[MAX_SIZE] = '¥0';
		m_len = strlen(m_str);
	}
public:
	//キャストオペレータ
	operator const char*() const { return m_str; }
public:
	//デフォルトコンストラクタ
	CStr():
		m_len(0)
	{
		m_str[0] = '¥0';
	}
	//コピーコンストラクタ
	template <std::size_t SS>
	CStr(const CStr<SS>& src) :
		m_len(src.m_len < MAX_SIZE ? src.m_len : MAX_SIZE)
	{
		memcpy(m_str, src.m_str, m_len);
		m_str[m_len] = '¥0';
	}
	//コンストラクタ
	CStr(const char* str)
	{
		setStr(str);
	}
protected:
	//フィールド
	unsigned char m_len;//文字列長
	char m_str[BUFF_SIZE];//文字列
};
```

【CRC 付き文字列型】

```
//--------------------
//CRC 付き文字列型
template <std::size_t S>
class CStrWithCRC : public CStr<S>
{
public:
	//型
	typedef CStr<S> CParent;
public:
	//アクセッサ
	crc32_t getCRC() const { return m_crc; }//CRC を取得
	void setStr(const char* str)//文字列をセット
	{
		CParent::setStr(str);
		m_crc = calcCRC32(CParent::m_str);
	}
public:
	//キャストオペレータ
	operator crc32_t() const { return m_crc; }
public:
	//デフォルトコンストラクタ
	CStrWithCRC() :
		CParent(nullptr),
		m_crc(0)
	{}
	//コピーコンストラクタ
	template <std::size_t SS>
	CStrWithCRC(const CStrWithCRC<SS>& src) :
		CStr<SS>(src),
		m_crc(src.m_crc)
	{}
	//コンストラクタ
	CStrWithCRC(const char* str) :
		CParent()
	{
		setStr(str);
	}
private:
	//フィールド
	crc32_t m_crc;//CRC
};
```

【文字列型を使用した既定の型】

```
//--------------------
//ID 型＆名前型
using ID_t = CStrWithCRC<8>;//ID 型
using name_t = CStr<32>;//名前型
```

【基本データ構造体】

```
//--------------------
//基本データ構造体
struct BASIC_DATA
{
	short m_atk;//攻撃力
	short m_def;//守備力
	//デフォルトコンストラクタ
	BASIC_DATA()
	{}
	//コピーコンストラクタ
	BASIC_DATA(const BASIC_DATA& src) :
		m_atk(src.m_atk),
		m_def(src.m_def)
	{}
	//コンストラクタ
```

```
	BASIC_DATA(const int atk, const int def) :
		m_atk(atk),
		m_def(def)
	{}
};
```

【アイテム構造体】

```
//--------------------
//アイテム構造体
struct ITEM_DATA
{
	crc32_t getKey() const { return m_id.getCRC(); }//キーを取得
	ID_t m_id;//ID
	name_t m_name;//名前
	char* m_title;//称号　※可変長文字列（ポインタ）のテスト用
	BASIC_DATA m_basic;//基本データ
	short m_recover;//回復力
	short m_num;//アイテム個数
	//称号をセット
	void setTitle(const char* title)
	{
		if (!title)
			m_title = nullptr;
		else
		{
			m_title = m_titleBuff;
		#ifdef USE_STRCPY_S
			strcpy_s(m_title, sizeof(m_titleBuff), title);
		#else//USE_STRCPY_S
			strcpy(m_title, title);
		#endif//USE_STRCPY_S
		}
	}
	//デフォルトコンストラクタ
	ITEM_DATA()
	{
		m_title = m_titleBuff;
		m_titleBuff[0] = '¥0';
	}
	//コンストラクタ
	ITEM_DATA(const char* id, const char* name, const int atk, const int def, const int recover, const int num) :
		m_id(id),
		m_name(name),
		m_basic(atk, def),
		m_recover(recover),
		m_num(num)
	{
		m_title = m_titleBuff;
		m_title[0] = '¥0';
	}
	//コピーコンストラクタ
	ITEM_DATA(const ITEM_DATA& src):
		m_id(src.m_id),
		m_name(src.m_name),
		m_basic(src.m_basic),
		m_recover(src.m_recover),
		m_num(src.m_num)
	{
		if (src.m_title)
		{
			m_title = m_titleBuff;
		#ifdef USE_STRCPY_S
			strcpy_s(m_title, sizeof(m_titleBuff), src.m_title);
		#else//USE_STRCPY_S
			strcpy(m_title, src.m_title);
```

```
            #endif//USE_STRCPY_S
            }
            else
                  m_title = nullptr;
      }
private:
      char m_titleBuff[16];//称号用バッファ
};
```

【アビリティデータ構造体】

```
//--------------------
//アビリティデータ構造体
struct ABILITY_DATA
{
      crc32_t getKey() const { return m_id.getCRC(); }//キーを取得
      ID_t m_id;//ID
      name_t m_name;//名前
      BASIC_DATA m_basic;//基本データ
      //デフォルトコンストラクタ
      ABILITY_DATA()
      {}
      //コンストラクタ
      ABILITY_DATA(const char* id, const char* name, const int atk, const int def) :
            m_id(id),
            m_name(name),
            m_basic(atk, def)
      {}
};
```

【キャラデータ構造体】

```
//--------------------
//キャラデータ構造体
struct CHARA_ABILITY_DATA;
struct CHARA_DATA
{
      crc32_t getKey() const { return m_id.getCRC(); }//キーを取得
      ID_t m_id;//ID
      name_t m_name;//名前
      char m_level;//レベル
      BASIC_DATA m_basic;//基本データ
      const ITEM_DATA* m_weapon;//武器
      const ITEM_DATA* m_shield;//盾
      int m_param1[2];//ダミーパラメータ 1
      int m_param2[2];//ダミーパラメータ 2
      //メソッド
      void setWeapon(const crc32_t weapon_id);//武器をセット
      void setWeapon(const char* weapon_id){ setWeapon(calcCRC32(weapon_id)); }//武器をセット
      void setShield(const crc32_t shield_id);//盾をセット
      void setShield(const char* shield_id){ setShield(calcCRC32(shield_id)); }//盾をセット
      void attachItems();//アイテムを参照し直す
      void addAbility(const crc32_t ability_id);//アビリティを追加
      void addAbility(const char* ability_id){ addAbility(calcCRC32(ability_id)); }//アビリティを追加
      ABILITY_DATA* getAbility(const int index);//アビリティを取得
      //デフォルトコンストラクタ
      CHARA_DATA()
      {}
      //コンストラクタ
      CHARA_DATA(const char* id, const char* name, const int level, const int atk, const int def, const char* weapon_id,
                  const char* shield_id, const int param1a, const int param1b, const int param2a, const int param2b) :
            m_id(id),
            m_name(name),
            m_level(level),
            m_basic(atk, def),
            m_weapon(nullptr),
            m_shield(nullptr),
```

```
            m_abilities(nullptr)
      {
            setWeapon(weapon_id);
            setShield(shield_id);
            m_param1[0] = param1a;
            m_param1[1] = param1b;
            m_param2[0] = param2a;
            m_param2[1] = param2b;
      }
private:
      crc32_t m_weaponId;//武器 ID
      crc32_t m_shieldId;//盾 ID
      CHARA_ABILITY_DATA* m_abilities;//キャラ習得アビリティ
};
```

【キャラ習得アビリティデータ構造体】

```
//--------------------
//キャラ習得アビリティデータ構造体
struct CHARA_ABILITY_DATA
{
      CHARA_ABILITY_DATA* m_next;//連結リスト
      ABILITY_DATA* m_ability;//アビリティ
};
```

【汎用コレクションクラス】

```
//--------------------
//汎用コレクションクラス
template<class T, std::size_t N, std::size_t BUFF1, std::size_t BUFF2>
class CCollection
{
private:
      //型
      typedef std::map<crc32_t, T*> search_t;//検索テーブル型
      typedef std::vector<T*> table_t;//整列テーブル型
      typedef typename table_t::iterator iterator;//イテレータ型
      typedef typename table_t::const_iterator const_iterator;//イテレータ型
public:
      //定数
      static const std::size_t DATA_NUM_MAX = N;//最大データ数
      static const std::size_t SEARCH_BUFF_SIZE = BUFF1;//検索テーブルのバッファサイズ
      static const std::size_t TABLE_BUFF_SIZE = BUFF2;//整列テーブルバッファサイズ
public:
      //アクセッサ
      iterator begin(){ return m_table->begin(); }//開始イテレータ
      iterator end(){ return m_table->end(); }//終了イテレータ
      const_iterator begin() const { return m_table->begin(); }//開始イテレータ
      const_iterator end() const { return m_table->end(); }//終了イテレータ
      const_iterator cbegin() const { return m_table->cbegin(); }//開始イテレータ
      const_iterator cend() const { return m_table->cend(); }//終了イテレータ
public:
      //検索
      T* find(const crc32_t key)
      {
            auto ite = m_search->find(key);
            if (ite == m_search->end())
                  return nullptr;//見つからなかった
            return ite->second;
      }
      T* find(const char* id)
      {
            return find(calcCRC32(id));
      }
      //登録
      T* regist(const T& data)
      {
```

```
		//プールアロケータから割り当て
		T* reg_data = m_pool.createData(data);
		if (!reg_data)
			return nullptr;//規定の個数が割り当て済みなら失敗
		{
			//検索テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_searchBuff);

			//検索テーブルに登録
			m_search->emplace(reg_data->getKey(), reg_data);
		}
		{
			//整列テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_tableBuff);

			//整列テーブルにも追加
			m_table->push_back(reg_data);
		}
		return reg_data;
	}
	//破棄
	void unregist(const crc32_t key)
	{
		T* data = find(key);
		if (!data)
			return;
		{
			//検索テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_searchBuff);

			//検索テーブルからアイテムを破棄
			m_search->erase(key);
		}
		//プールアロケータから解放
		m_pool.destroy(data);
	}
private:
	//整列テーブルを生成
	void createTable()
	{
		if (!m_table)
		{
			//整列テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_tableBuff);

			m_table = new table_t();
		}
	}
	//整列テーブル破棄
	void destroyTable()
	{
		if (m_table)
		{
			//整列テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_tableBuff);

			delete m_table;
			m_table = nullptr;
		}
		//バッファをクリア
		m_tableBuff.clearN();
	}
public:
	//ソート
	//※整列テーブル再作成
```

```
	void sort(const bool is_descendant = false)
	{
		//整列テーブルの破棄と生成
		destroyTable();//破棄
		createTable();//生成
		{
			//整列テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_tableBuff);

			//検索テーブルから整列テーブルに全要素コピー
			for (auto& pair : *m_search)
			{
				m_table->push_back(pair.second);
			}

			//名前順ソート
			std::sort(m_table->begin(), m_table->end(),
				[&is_descendant](T* lhs, T* rhs)->bool
				{
					const int cmp = strcmp(lhs->m_name, rhs->m_name);
					return is_descendant ? cmp > 0 : cmp < 0;
				}
			);
		}
	}
public:
	//コンストラクタ
	CCollection()
	{
		{
			//検索テーブルバッファを多態アロケータにセット
			//※以後、処理ブロックを抜けるまで、new 演算子による
			//  メモリ確保が m_searchBuff から行われる
			CTempPolyStackAllocator allocator(m_searchBuff);

			//検索テーブル生成
			m_search = new search_t;
		}
		//整列テーブル生成
		createTable();
	}
	//デストラクタ
	~CCollection()
	{
		{
			//検索テーブルバッファを多態アロケータにセット
			CTempPolyStackAllocator allocator(m_searchBuff);

			//検索テーブル破棄
			//※とくに必要ではないが念のため
			delete m_search;
		}
		//整列テーブル破棄
		destroyTable();
	}
private:
	//フィールド
	CPoolAllocatorWithType<T, DATA_NUM_MAX> m_pool;//データ用のバッファ
	CStackAllocatorWithBuff<SEARCH_BUFF_SIZE> m_searchBuff;//検索テーブルバッファ
	search_t* m_search;//検索テーブル
	CStackAllocatorWithBuff<TABLE_BUFF_SIZE> m_tableBuff;//整列テーブルバッファ
	table_t* m_table;//整列テーブル
};
```

【インベントリクラス】

```
//--------------------
//インベントリクラス
using CInventory = CCollection<ITEM_DATA, 100, 8 * 1024, 8 * 1024>;
```

【アビリティリストクラス】

```
//--------------------
//アビリティリストクラス
using CAbilityList = CCollection<ABILITY_DATA, 40, 8 * 1024, 8 * 1024>;
```

【キャラリストクラス】

```
//--------------------
//キャラリストクラス
using CCharaList = CCollection<CHARA_DATA, 10, 8 * 1024, 8 * 1024>;
```

【キャラ習得アビリティデータバッファクラス】

```
//--------------------
//キャラ習得アビリティデータバッファクラス
class CCharaAbilityBuff
{
public:
	//メソッド
	CHARA_ABILITY_DATA* assign(){ return m_pool.createData(); }
	void release(CHARA_ABILITY_DATA* data){ m_pool.destroyData(data); }
public:
	//コンストラクタ
	CCharaAbilityBuff()
	{}
private:
	//フィールド
	CPoolAllocatorWithType<CHARA_ABILITY_DATA, 100> m_pool;//データ用のバッファ
};
```

【キャラデータ構造体：メソッド実装】

```
//--------------------
//キャラデータ構造体：メソッド実装
//武器をセット
void CHARA_DATA::setWeapon(const crc32_t weapon_id)
{
	CSingleton<CInventory> inventory;
	ITEM_DATA* item = inventory->find(weapon_id);//イベントリから検索
	if (!item)
	{
		//対象アイテムがないので持っていないことにする
		m_weaponId = 0;
		m_weapon = nullptr;
		return;
	}
	m_weaponId = weapon_id;
	m_weapon = item;
}
//盾をセット
void CHARA_DATA::setShield(const crc32_t shield_id)
{
	CSingleton<CInventory> inventory;
	ITEM_DATA* item = inventory->find(shield_id);//イベントリから検索
	if (!item)
	{
		//対象アイテムがないので持っていないことにする
		m_shieldId = 0;
		m_shield = nullptr;
		return;
	}
	m_shieldId = shield_id;
	m_shield = item;
```

```
}
//アイテムを参照し直す
void CHARA_DATA::attachItems()
{
	setWeapon(m_weaponId);//武器
	setShield(m_shieldId);//盾
}
//キャラ習得アビリティを追加
void CHARA_DATA::addAbility(const crc32_t ability_id)
{
	CSingleton<CAbilityList> chara_ability_list;
	ABILITY_DATA* ability = chara_ability_list->find(ability_id);//アビリティリストから検索
	if (!ability)//アビリティがなければ終了
		return;
	//キャラ習得アビリティのバッファを割り当て
	CSingleton<CCharaAbilityBuff> chara_ability_buff;
	CHARA_ABILITY_DATA* new_chara_ability = chara_ability_buff->assign();
	if (!new_chara_ability)//バッファが足りなければ失敗
		return;
	//バッファにアビリティをセットして連結リストの最後に連結
	new_chara_ability->m_next = nullptr;
	new_chara_ability->m_ability = ability;
	CHARA_ABILITY_DATA* chara_ability = m_abilities;
	while (chara_ability && chara_ability->m_next)
		chara_ability = chara_ability->m_next;
	if (chara_ability)
		chara_ability->m_next = new_chara_ability;
	else
		m_abilities = new_chara_ability;
}
//キャラ習得アビリティを取得
ABILITY_DATA* CHARA_DATA::getAbility(const int index)
{
	CHARA_ABILITY_DATA* chara_ability = m_abilities;
	for (int i = 0; i < index && chara_ability; ++i)
		chara_ability = chara_ability->m_next;
	return chara_ability == nullptr ? nullptr : chara_ability->m_ability;
}
```

【進行＆フラグデータ】

```
//--------------------
//進行＆フラグデータ
class CPhaseAndFlags
{
public:
	//型
	typedef std::bitset<64> flag_t;
public:
	//アクセッサ
	int getPhase() const { return m_phase; }//フェーズ取得
	void setPhase(const int phase){ m_phase = phase; }//フェーズ更新
	bool getFlag(const int index) const { return m_flags[index]; }//フラグ取得
	void setFlag(const int index, const bool flag){ m_flags[index] = flag; }//フラグ更新
	const flag_t& getFlags() const { return m_flags; }
public:
	//コンストラクタ
	CPhaseAndFlags() :
		m_phase(0)
	{
		m_flags.reset();
	}
private:
	//フィールド
	short m_phase;//進行フェーズ
	flag_t m_flags;//フラグ
```

```
};
```

▼ データ管理システムのサンプルプログラム：テスト処理部

以上のクラス・構造体を使用し、実際にデータ操作するサンプルを示す。

なお、サンプルプログラム中に汎用シングルトンテンプレートクラス「CSingleton<T>」が突然出てくるが、これはコンパイル時点で静的なバッファを割り当てておき、初めてアクセスがあった時にインスタンスを生成する仕組みである。

【全データをリセット】

```
//--------------------
//全データリセット
void resetAll()
{
	printf("------------------------------------------------------------¥n");
	printf("【全データリセット】¥n");

	//インベントリデータ破棄
	{
		CSingleton<CInventory> inventory;
		inventory.destroy();
	}
	//アビリティデータ破棄
	{
		CSingleton<CAbilityList> ability_list;
		ability_list.destroy();
	}
	//キャラデータ破棄
	{
		CSingleton<CCharaList> chara_list;
		chara_list.destroy();
	}
	//キャラアビリティデータ破棄
	{
		CSingleton<CAbilityList> chara_ability_list;
		chara_ability_list.destroy();
	}
	//フェーズ＆フラグデータ破棄
	{
		CSingleton<CPhaseAndFlags> phase_and_flags;
		phase_and_flags.destroy();
	}
}
```

【テストデータ作成】※パラメータに応じて作成されるデータが変わる

```
//--------------------
//テストデータ作成
void makeTestData(const int pattern)
{
	printf("------------------------------------------------------------¥n");
	printf("【テストデータ作成】(pattern=%d)¥n", pattern);

	//インベントリデータ登録
	{
		CSingleton<CInventory> inventory;
		//武器登録
		for (int i = 0; i < 5 + pattern * 10; i += (1 + pattern))
		{
			char id[8];
```

```
		char name[32];
	#ifdef USE_STRCPY_S
		sprintf_s(id, sizeof(id), "w%05d", (i + 1) * 10);
		sprintf_s(name, sizeof(name), "武器%03d", i + 1);
	#else//USE_STRCPY_S
		sprintf(id, "w%05d", i);
		sprintf(name, "武器%03d", i);
	#endif//USE_STRCPY_S
		ITEM_DATA item(id, name, 10 + i, i / 2, 0, 1);
		if (item.m_title && i > 0)
		{
			char buf[16];
		#ifdef USE_STRCPY_S
			sprintf_s(buf, sizeof(buf), "性能:+%d", i);
		#else//USE_STRCPY_S
			sprintf(buf, "性能:+%d", i);
		#endif//USE_STRCPY_S
			item.setTitle(buf);
		}
		else
			item.setTitle(nullptr);
		inventory->regist(item);
	}
	//盾登録
	for (int i = 0; i < 5 + pattern * 10; i += (1 + pattern))
	{
		char id[8];
		char name[32];
	#ifdef USE_STRCPY_S
		sprintf_s(id, sizeof(id), "s%05d", (i + 1) * 10);
		sprintf_s(name, sizeof(name), "盾%03d", i + 1);
	#else//USE_STRCPY_S
		sprintf(id, "s%05d", i);
		sprintf(name, "盾%03d", i);
	#endif//USE_STRCPY_S
		ITEM_DATA item(id, name, 0, 5 + i, 0, 1);
		if (item.m_title && i > 0)
		{
			char buf[16];
		#ifdef USE_STRCPY_S
			sprintf_s(buf, sizeof(buf), "グレード:%c", 'A' + i);
		#else//USE_STRCPY_S
			sprintf(buf, "グレード:%c", 'A' + i);
		#endif//USE_STRCPY_S
			item.setTitle(buf);
		}
		else
			item.setTitle(nullptr);
		inventory->regist(item);
	}
	//回復薬登録
	for (int i = pattern; i < 3 + pattern * 3; ++i)
	{
		char id[8];
		char name[32];
	#ifdef USE_STRCPY_S
		sprintf_s(id, sizeof(id), "r%05d", (i + 1) * 10);
		sprintf_s(name, sizeof(name), "回復薬%03d", i + 1);
	#else//USE_STRCPY_S
		sprintf(id, "r%05d", i);
		sprintf(name, "回復薬%03d", i);
	#endif//USE_STRCPY_S
		ITEM_DATA item(id, name, 0, 0, 5 + i * 2, 10);
		item.setTitle(nullptr);
		inventory->regist(item);
```

```
		}
		inventory->sort();
	}
	//アビリティデータ登録
	{
		CSingleton<CAbilityList> ability_list;
		for (int i = pattern; i < 10 + pattern * 10; ++i)
		{
			char id[8];
			char name[32];
		#ifdef USE_STRCPY_S
			sprintf_s(id, sizeof(id), "a%05d", (i + 1) * 10);
			sprintf_s(name, sizeof(name), "特技%03d", i + 1);
		#else//USE_STRCPY_S
			sprintf(id, "a%05d", i);
			sprintf(name, "特技%03d", i);
		#endif//USE_STRCPY_S
			ABILITY_DATA ability(id, name, 5 + i * 3, 4 + i * 2);
			ability_list->regist(ability);
		}
		ability_list->sort();
	}
	//キャラデータ登録
	{
		CSingleton<CCharaList> chara_list;
		if (pattern & 1)
		{
			CHARA_DATA chara1("c00010", "たろう", 11, 15, 20, "w00030", "s00030", 111, 222, 333, 444);
			CHARA_DATA chara2("c00020", "じろう", 21, 25, 40, "w00050", "s00010", 999, 888, 777, 666);
			CHARA_DATA chara3("c00030", "さぶろう", 31, 55, 3, "w00070", "s00050", 123, 456, 987, 654);
			CHARA_DATA chara4("c00040", "しろう", 41, 55, 3, "w00090", "s00090", 123, 456, 987, 654);
			chara1.addAbility("a00030");
			chara1.addAbility("a00010");
			chara3.addAbility("a00050");
			chara3.addAbility("a00010");
			chara3.addAbility("a00040");
			chara4.addAbility("a00020");
			chara_list->regist(chara1);
			chara_list->regist(chara2);
			chara_list->regist(chara3);
			chara_list->regist(chara4);
			chara_list->sort(true);
		}
		else
		{
			CHARA_DATA chara1("c00010", "たろう", 10, 15, 20, "w00020", "s00020", 111, 222, 333, 444);
			CHARA_DATA chara2("c00020", "じろう", 20, 25, 40, "w00050", "s00010", 999, 888, 777, 666);
			CHARA_DATA chara3("c00030", "さぶろう", 30, 55, 3, "w00010", "s00030", 123, 456, 987, 654);
			chara1.addAbility("a00030");
			chara1.addAbility("a00020");
			chara1.addAbility("a00050");
			chara1.addAbility("a00010");
			chara3.addAbility("a00050");
			chara_list->regist(chara1);
			chara_list->regist(chara2);
			chara_list->regist(chara3);
			chara_list->sort();
		}
	}
	//進行データ更新
	{
		CSingleton<CPhaseAndFlags> phase_and_flags;
		if (pattern & 1)
		{
			phase_and_flags->setPhase(3 + pattern);
```

```
                phase_and_flags->setFlag(1, true);
                phase_and_flags->setFlag(2 + pattern, true);
                phase_and_flags->setFlag(62, true);
            }
            else
            {
                phase_and_flags->setPhase(7 + pattern);
                phase_and_flags->setFlag(0, true);
                phase_and_flags->setFlag(2, true);
                phase_and_flags->setFlag(4, true);
                phase_and_flags->setFlag(5, true);
                phase_and_flags->setFlag(63 - pattern, true);
            }
        }
}
```

【現在のデータを表示】

```
//--------------------
//現在のデータを表示
void printDataAll()
{
    printf("------------------------------------------------------------¥n");
    printf("【現在のデータを表示】¥n");

    //インベントリデータ表示
    printf("--- インベントリ ---¥n");
    {
        CSingleton<CInventory> inventory;
        for (auto& ite : *inventory)
        {
            printf("ID=¥"%s¥"(0x%08x), name=¥"%s¥", atk=%d, def=%d, recover=%d, num=%d¥n",
                ite->m_id.c_str(), ite->m_id.getCRC(), ite->m_name.c_str(), ite->m_basic.m_atk,
                                                    ite->m_basic.m_def, ite->m_recover, ite->m_num);
            if (ite->m_title)
                printf("                              title=¥"%s¥"¥n", ite->m_title);
        }
    }
    //アビリティデータ表示
    printf("--- アビリティ ---¥n");
    {
        CSingleton<CAbilityList> ability_list;
        for (auto& ite : *ability_list)
        {
            printf("ID=¥"%s¥"(0x%08x), name=¥"%s¥", atk=%d, def=%d¥n",
                ite->m_id.c_str(), ite->m_id.getCRC(), ite->m_name.c_str(), ite->m_basic.m_atk,
                                                                                ite->m_basic.m_def);
        }
    }
    //キャラデータ表示
    printf("--- キャラ ---¥n");
    {
        CSingleton<CCharaList> chara_list;
        for (auto& ite : *chara_list)
        {
            printf("ID=¥"%s¥"(0x%08x), name=¥"%s¥", level=%d, atk=%d, def=%d¥n",
                ite->m_id.c_str(), ite->m_id.getCRC(), ite->m_name.c_str(), ite->m_level, ite->m_basic.m_atk,
                                                                                ite->m_basic.m_def);
            printf("                              param1={%d, %d}, param2={%d, %d}¥n",
                ite->m_param1[0], ite->m_param1[1], ite->m_param2[0], ite->m_param2[1]);
            if (ite->m_weapon)
                printf("                              weapon=%s¥n", ite->m_weapon->m_name.c_str());
            if (ite->m_shield)
                printf("                              shield=%s¥n", ite->m_shield->m_name.c_str());
            for (int i = 0;; ++i)
            {
```

```
                        const ABILITY_DATA* ability = ite->getAbility(i);
                        if (!ability)
                              break;
                        printf("                                [%s]¥n", ability->m_name.c_str());
                  }
            }
      }
      //進行データ表示
      printf("--- 進行データ ---¥n");
      {
            CSingleton<CPhaseAndFlags> phase_flags;
            printf("pahse=%d¥n", phase_flags->getPhase());
            printf("flags=¥n");
            for (unsigned int i = 0; i < phase_flags->getFlags().size(); ++i)
            {
                  if (i > 0 && i % 16 == 0)
                        printf("¥n");
                  printf(" %d", phase_flags->getFlag(i));
            }
            printf("¥n");
      }
}
```

【データ操作テスト】

```
//--------------------
//データ操作テスト
void testDataControl()
{
      //全データリセット
      resetAll();
      //現在のデータを表示
      printDataAll();
      //テストデータ作成(pattern=0)
      makeTestData(0);
      //現在のデータを表示
      printDataAll();
      //全データリセット
      resetAll();
      //テストデータ作成(pattern=1)
      makeTestData(1);
      //現在のデータを表示
      printDataAll();
}
```

【テストメイン】

```
//--------------------
//テスト
int main(const int argc, const char* argv[])
{
      testDataControl();
      return EXIT_SUCCESS;
}
```

↓（実行結果）

```
------------------------------------------------------------
【全データリセット】                                    ↓データが存在しない状態
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
--- アビリティ ---
--- キャラ ---
--- 進行データ ---
pahse=0
flags=
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
------------------------------------------------------------
【テストデータ作成】(pattern=0)                    ↓パターン 0 で作成したデータ
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
ID="r00010"(0x62cd9fc0), name="回復薬 001", atk=0, def=0, recover=5, num=10
ID="r00020"(0x49e0cc03), name="回復薬 002", atk=0, def=0, recover=7, num=10
ID="r00030"(0x50fbfd42), name="回復薬 003", atk=0, def=0, recover=9, num=10
ID="s00010"(0xa9914c65), name="盾 001", atk=0, def=5, recover=0, num=1
ID="s00020"(0x82bc1fa6), name="盾 002", atk=0, def=6, recover=0, num=1
                         title="グレード:B"
ID="s00030"(0x9ba72ee7), name="盾 003", atk=0, def=7, recover=0, num=1
                         title="グレード:C"
ID="s00040"(0xd4e6b820), name="盾 004", atk=0, def=8, recover=0, num=1
                         title="グレード:D"
ID="s00050"(0xcdfd8961), name="盾 005", atk=0, def=9, recover=0, num=1
                         title="グレード:E"
ID="w00010"(0x32000e73), name="武器 001", atk=10, def=0, recover=0, num=1
ID="w00020"(0x192d5db0), name="武器 002", atk=11, def=0, recover=0, num=1
                         title="性能:+1"
ID="w00030"(0x00366cf1), name="武器 003", atk=12, def=1, recover=0, num=1
                         title="性能:+2"
ID="w00040"(0x4f77fa36), name="武器 004", atk=13, def=1, recover=0, num=1
                         title="性能:+3"
ID="w00050"(0x566ccb77), name="武器 005", atk=14, def=2, recover=0, num=1
                         title="性能:+4"
--- アビリティ ---
ID="a00010"(0xe78feff5), name="特技 001", atk=5, def=4
ID="a00020"(0xcca2bc36), name="特技 002", atk=8, def=6
ID="a00030"(0xd5b98d77), name="特技 003", atk=11, def=8
ID="a00040"(0x9af81bb0), name="特技 004", atk=14, def=10
ID="a00050"(0x83e32af1), name="特技 005", atk=17, def=12
ID="a00060"(0xa8ce7932), name="特技 006", atk=20, def=14
ID="a00070"(0xb1d54873), name="特技 007", atk=23, def=16
ID="a00080"(0x364d54bc), name="特技 008", atk=26, def=18
ID="a00090"(0x2f5665fd), name="特技 009", atk=29, def=20
ID="a00100"(0xff56b483), name="特技 010", atk=32, def=22
--- キャラ ---
ID="c00030"(0x98712c7c), name="さぶろう", level=30, atk=55, def=3
                         param1={123, 456}, param2={987, 654}
                         weapon=武器 001
                         shield=盾 003
                         [特技 005]
ID="c00020"(0x816a1d3d), name="じろう", level=20, atk=25, def=40
                         param1={999, 888}, param2={777, 666}
                         weapon=武器 005
                         shield=盾 001
ID="c00010"(0xaa474efe), name="たろう", level=10, atk=15, def=20
                         param1={111, 222}, param2={333, 444}
                         weapon=武器 002
                         shield=盾 002
                         [特技 003]
                         [特技 002]
                         [特技 005]
                         [特技 001]
--- 進行データ ---
pahse=7
flags=
 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
```

```
------------------------------------------------------------
【全データリセット】
------------------------------------------------------------
【テストデータ作成】(pattern=1)                    ↓パターン 1 で作成したデータ
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
ID="r00020"(0x49e0cc03), name="回復薬 002", atk=0, def=0, recover=7, num=10
ID="r00030"(0x50fbfd42), name="回復薬 003", atk=0, def=0, recover=9, num=10
ID="r00040"(0x1fba6b85), name="回復薬 004", atk=0, def=0, recover=11, num=10
ID="r00050"(0x06a15ac4), name="回復薬 005", atk=0, def=0, recover=13, num=10
ID="r00060"(0x2d8c0907), name="回復薬 006", atk=0, def=0, recover=15, num=10
ID="s00010"(0xa9914c65), name="盾 001", atk=0, def=5, recover=0, num=1
ID="s00030"(0x9ba72ee7), name="盾 003", atk=0, def=7, recover=0, num=1
                         title="グレード:C"
ID="s00050"(0xcdfd8961), name="盾 005", atk=0, def=9, recover=0, num=1
                         title="グレード:E"
ID="s00070"(0xffcbebe3), name="盾 007", atk=0, def=11, recover=0, num=1
                         title="グレード:G"
ID="s00090"(0x6148c66d), name="盾 009", atk=0, def=13, recover=0, num=1
                         title="グレード:I"
ID="s00110"(0xa8532652), name="盾 011", atk=0, def=15, recover=0, num=1
                         title="グレード:K"
ID="s00130"(0x9a6544d0), name="盾 013", atk=0, def=17, recover=0, num=1
                         title="グレード:M"
ID="s00150"(0xcc3fe356), name="盾 015", atk=0, def=19, recover=0, num=1
                         title="グレード:O"
ID="w00010"(0x32000e73), name="武器 001", atk=10, def=0, recover=0, num=1
ID="w00030"(0x00366cf1), name="武器 003", atk=12, def=1, recover=0, num=1
                         title="性能:+2"
ID="w00050"(0x566ccb77), name="武器 005", atk=14, def=2, recover=0, num=1
                         title="性能:+4"
ID="w00070"(0x645aa9f5), name="武器 007", atk=16, def=3, recover=0, num=1
                         title="性能:+6"
ID="w00090"(0xfad9847b), name="武器 009", atk=18, def=4, recover=0, num=1
                         title="性能:+8"
ID="w00110"(0x33c26444), name="武器 011", atk=20, def=5, recover=0, num=1
                         title="性能:+10"
ID="w00130"(0x01f406c6), name="武器 013", atk=22, def=6, recover=0, num=1
                         title="性能:+12"
ID="w00150"(0x57aea140), name="武器 015", atk=24, def=7, recover=0, num=1
                         title="性能:+14"
--- アビリティ ---
ID="a00020"(0xcca2bc36), name="特技 002", atk=8, def=6
ID="a00030"(0xd5b98d77), name="特技 003", atk=11, def=8
ID="a00040"(0x9af81bb0), name="特技 004", atk=14, def=10
ID="a00050"(0x83e32af1), name="特技 005", atk=17, def=12
ID="a00060"(0xa8ce7932), name="特技 006", atk=20, def=14
ID="a00070"(0xb1d54873), name="特技 007", atk=23, def=16
ID="a00080"(0x364d54bc), name="特技 008", atk=26, def=18
ID="a00090"(0x2f5665fd), name="特技 009", atk=29, def=20
ID="a00100"(0xff56b483), name="特技 010", atk=32, def=22
ID="a00110"(0xe64d85c2), name="特技 011", atk=35, def=24
ID="a00120"(0xcd60d601), name="特技 012", atk=38, def=26
ID="a00130"(0xd47be740), name="特技 013", atk=41, def=28
ID="a00140"(0x9b3a7187), name="特技 014", atk=44, def=30
ID="a00150"(0x822140c6), name="特技 015", atk=47, def=32
ID="a00160"(0xa90c1305), name="特技 016", atk=50, def=34
ID="a00170"(0xb0172244), name="特技 017", atk=53, def=36
ID="a00180"(0x378f3e8b), name="特技 018", atk=56, def=38
ID="a00190"(0x2e940fca), name="特技 019", atk=59, def=40
ID="a00200"(0xfd100ada), name="特技 020", atk=62, def=42
--- キャラ ---
ID="c00010"(0xaa474efe), name="たろう", level=11, atk=15, def=20
                         param1={111, 222}, param2={333, 444}
```

```
                                weapon=武器 003
                                shield=盾 003
                                [特技 003]
ID="c00020"(0x816a1d3d), name="じろう", level=21, atk=25, def=40
                                param1={999, 888}, param2={777, 666}
                                weapon=武器 005
                                shield=盾 001
ID="c00040"(0xd730babb), name="しろう", level=41, atk=55, def=3
                                param1={123, 456}, param2={987, 654}
                                weapon=武器 009
                                shield=盾 009
                                [特技 002]
ID="c00030"(0x98712c7c), name="さぶろう", level=31, atk=55, def=3
                                param1={123, 456}, param2={987, 654}
                                weapon=武器 007
                                shield=盾 005
                                [特技 005]
                                [特技 004]
--- 進行データ ---
pahse=4
flags=
 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
```

▼ データ管理システムに対するシリアライズ処理

ここまではシリアライズは関係していない。

このデータをセーブ／ロードできるように、シリアライズの処理を実装する。

● 準備：フレンド化

まずは準備として、プライベートメンバーをシリアライズする必要のあるクラス・構造体に対して、フレンド化を設定する。

【フレンド化の設定】

```
//--------------------
//短い文字列型
template <std::size_t S>
class CStr
{
//... (略) ...
      //シリアライズ用のフレンド設定
      FRIEND_SERIALIZE();
};
//--------------------
//CRC 付き文字列型
template <std::size_t S>
class CStrWithCRC : public CStr<S>
{
//... (略) ...
      //シリアライズ用のフレンド設定
      FRIEND_SERIALIZE();
};
//--------------------
//キャラデータ構造体
struct CHARA_ABILITY_DATA;
```

```
struct CHARA_DATA
{
//...（略）...
      //シリアライズ用のフレンド設定
      FRIEND_SERIALIZE();
};
//--------------------
//進行＆フラグデータ
class CPhaseAndFlags
{
//...（略）...
      //シリアライズ用のフレンド設定
      FRIEND_SERIALIZE();
};
```

● 【任意】準備：バージョン定義

続いて、シリアライズ対象のクラス・構造体にバージョンを定義する。必須ではないが、なるべく付けておいたほうがよい。

なお、バージョンの定義はどこに記述してもよいが、構造変更時にこまめに更新できるように、対象クラス・構造体の近くに定義したほうがよい。

【バージョン定義】

```
//--------------------
//基本データ構造体
struct BASIC_DATA
{
//...（略）...
};
//構造体バージョン
SERIALIZE_VERSION_DEF(BASIC_DATA, 1, 0);
//--------------------
//アイテム構造体
struct ITEM_DATA
{
//...（略）...
};
//構造体バージョン
SERIALIZE_VERSION_DEF(ITEM_DATA, 1, 0);
//--------------------
//アビリティデータ構造体
struct ABILITY_DATA
{
//...（略）...
};
//構造体バージョン
SERIALIZE_VERSION_DEF(ABILITY_DATA, 1, 0);
//--------------------
//キャラデータ構造体
struct CHARA_DATA
{
//...（略）...
};
//構造体バージョン
SERIALIZE_VERSION_DEF(CHARA_DATA, 1, 0);
//--------------------
//進行＆フラグデータ
class CPhaseAndFlags
{
//...（略）...
```

```
};
//構造体バージョン
SERIALIZE_VERSION_DEF(CPhaseAndFlags, 1, 0);
```

● 準備：セーブデータ専用構造体

「キャラ習得アビリティデータ：CHARA_ABILITY_DATA」は、セーブデータに保存する必要のある情報だが、ポインタしか持っていない。

このようなデータを扱うために、シリアライズ／デシリアライズ時にだけ一時的に使用する構造体を用意する。

【キャラ習得アビリティデータのシリアライズ用構造体】

```
//--------------------
//キャラ習得アビリティデータ構造体；シリアライズ用
struct CHARA_ABILITY_DATA_FOR_SAVE
{
	crc32_t m_abilityId;//アビリティ ID
};
```

● シリアライズ／デシリアライズの定義：汎用クラス

まずは汎用クラスのシリアライズを定義する。

対象がテンプレートクラスの場合、そのテンプレートクラス固有のテンプレート引数をいっしょに定義する。

シリアライズ対象項目とその指定方法に関する部分を赤字で示す。

【std::bitset のシリアライズ／デシリアライズ共通処理】
※標準ライブラリのクラスなので private メンバーにアクセスできない⇒バイトデータとして保存

```
namespace serial{
	//--------------------
	//シリアライズ処理：std::bitset<>
	template<class Arc, std::size_t N>
	struct serialize<Arc, std::bitset<N> > {
		void operator()(Arc& arc, const std::bitset<N>& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pairBin("bitset", obj);//オブジェクト全体をバイトデータとして扱う
		}
	};
}
```

【CStr のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
	//--------------------
	//シリアライズ処理：CStr<>
	template<class Arc, std::size_t S>
	struct serialize<Arc, CStr<S> > {
		void operator()(Arc& arc, const CStr<S>& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("len", obj.m_len);
			arc & pairStr("str", obj.m_str);//文字列データとして扱う
		}
	};
}
```

【CStrWithCRC のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
	//--------------------
	//シリアライズ処理：CStrWithCRC<>
	template<class Arc, std::size_t S>
	struct serialize<Arc, CStrWithCRC<S> > {
		void operator()(Arc& arc, const CStrWithCRC<S>& obj, const CVersion& ver, const CVersion& now_ver)
		{
			serialize<Arc, typename CStrWithCRC<S>::CParent> parent_functor;
			parent_functor(arc, obj, ver, now_ver);//親クラスのシリアライズを直接呼び出し
			arc & pair("crc", obj.m_crc);
		}
	};
}
```

● シリアライズ／デシリアライズの定義：共通データ

【BASIC_DATA のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
	//--------------------
	//シリアライズ処理：BASIC_DATA
	template<class Arc>
	struct serialize<Arc, BASIC_DATA> {
		void operator()(Arc& arc, const BASIC_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("atk", obj.m_atk);
			arc & pair("def", obj.m_def);
		}
	};
}
```

● シリアライズ／デシリアライズの定義：アイテムデータ

アイテムデータには特殊なメンバー「称号：m_title」がある。

これは、ポインタ型の変数で、称号が付かないアイテムは nullptr として扱うものである。称号が付く場合は内部の「称号用バッファ：char m_titleBuff[16]」を割り当てて文字列をセットする。

デシリアライズの際にこの処理を自動的に行わせることができないので、デフォルトコンストラクタで無条件にバッファを割り当てるものとする。その上で、「ロード後処理」を定義し、デシリアライズの結果が「空文字列」（1 バイト目が　‘¥0’　）なら、nullptr に置き換える処理を行う。

【ITEM_DATA のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
	//--------------------
	//シリアライズ処理：ITEM_DATA
	template<class Arc>
	struct serialize<Arc, ITEM_DATA> {
		void operator()(Arc& arc, const ITEM_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pair("id", obj.m_id);
			arc & pair("name", obj.m_name);
			arc & pairStr("title", obj.m_title);//可変長文字列（ポインタ）
			arc & pair("basic", obj.m_basic);
			arc & pair("recover", obj.m_recover);
```

```
                arc & pair("num", obj.m_num);
            }
        };
}
```

【ITEM_DATA のロード後処理】

```
namespace serial{
      //--------------------
      //ロード後処理：ITEM_DATA
      template<class Arc>
      struct afterLoad<Arc, ITEM_DATA> {
            void operator()(Arc& arc, ITEM_DATA& obj, const CVersion& ver, const CVersion& now_ver)
            {
                  if (obj.m_title && obj.m_title[0] == '¥0')//空文字列の称号は nullptr にする
                        obj.m_title = nullptr;
            }
      };
}
```

● シリアライズ／デシリアライズの定義：アビリティデータ

【ABILITY_DATA のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
      //--------------------
      //シリアライズ処理：ABILITY_DATA
      template<class Arc>
      struct serialize<Arc, ABILITY_DATA> {
            void operator()(Arc& arc, const ABILITY_DATA& obj, const CVersion& ver, const CVersion& now_ver)
            {
                  arc & pair("id", obj.m_id);
                  arc & pair("name", obj.m_name);
                  arc & pair("basic", obj.m_basic);
            }
      };
}
```

● シリアライズ／デシリアライズの定義：キャラデータ

キャラデータには幾つか特殊な要件がある。

まず、このデータでは「武器」と「盾」の参照（ITEM_DATA のポインタ）を扱うが、セーブデータに記録するのはその ID（CRC）のみである。これは、「ロード後処理」を利用し、参照の割り当てを行う事で対処する。

それと、連結リストによる「習得アビリティ」を扱う。メンバー変数の中にはキャラア習得アビリティデータのポインタがあるが、これをそのままシリアライズするのは難しいため、「コレクター」と「ディストリビュータ」を利用して、「アビリティ ID」を保存するように処理する。

【CHARA_DATA のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
      //--------------------
      //シリアライズ処理：CHARA_DATA
      template<class Arc>
      struct serialize<Arc, CHARA_DATA> {
            void operator()(Arc& arc, const CHARA_DATA& obj, const CVersion& ver, const CVersion& now_ver)
            {
```

```
                arc & pair("id", obj.m_id);
                arc & pair("name", obj.m_name);
                arc & pair("level", obj.m_level);
                arc & pair("basic", obj.m_basic);
                arc & pair("weapon", obj.m_weaponId);
                arc & pair("shield", obj.m_shieldId);
                arc & pair("param1", obj.m_param1);
                arc & pair("param2", obj.m_param2);
                //下記の項目はセーブデータに記録しない
                //obj.m_weapon//ITEM_DATA*
                //obj.m_shield//ITEM_DATA*
                //obj.m_abilities//CHARA_ABILITY_DATA*
        }
    };
}
```

【CHARA_DATA のロード後処理】

```
namespace serial{
    //--------------------
    //ロード後処理：CHARA_DATA
    template<class Arc>
    struct afterLoad<Arc, CHARA_DATA> {
        void operator()(Arc& arc, CHARA_DATA& obj, const CVersion& ver, const CVersion& now_ver)
        {
            CSingleton<CInventory> inventory;//インベントリ取得
            obj.m_weapon = inventory->find(obj.m_weaponId);//武器の参照（ポインタ）を割り当て
            obj.m_shield = inventory->find(obj.m_shieldId);//盾の参照（ポインタ）を割り当て
            obj.m_abilities = nullptr;//アビリティは一旦 nullptr にしておく
        }
    };
}
```

【CHARA_DATA のコレクター（収取処理）】

```
namespace serial{
    //--------------------
    //収集処理：CHARA_DATA
    template<class Arc>
    struct collector<Arc, CHARA_DATA> {
        void operator()(Arc& arc, const CHARA_DATA& obj, const CVersion& ver)
        {
            CHARA_ABILITY_DATA* chara_ability = obj.m_abilities;
            while (chara_ability)//全ての連結リストをたどって処理
            {
                //アビリティ情報取得
                ABILITY_DATA* ability_data = chara_ability->m_ability;
                //セーブ用アビリティ情報作成
                //※一時的にローカル変数に記録した情報をセーブする
                CHARA_ABILITY_DATA_FOR_SAVE chara_ability_data_for_save;
                chara_ability_data_for_save.m_abilityId = ability_data->getKey();//アビリティ ID
                //シリアライズ
                arc << pair("charaAbility", chara_ability_data_for_save);//名前が重要
                //次のアビリティ
                chara_ability = chara_ability->m_next;
            }
        }
    };
}
```

【CHARA_DATA のディストリビュータ（分配処理）】

```
namespace serial{
    //--------------------
    //分配処理：CHARA_DATA
    template<class Arc>
    struct distributor<Arc, CHARA_DATA> {
        void operator()(Arc& arc, CHARA_DATA& obj, const std::size_t array_num_on_save_data,
```

```
                                const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
                                                                        const CItemBase& target_item)
        {
            if (target_item == "charaAbility")//キャラ習得アビリティのデータか名前で判定
            {
                //セーブ用アビリティ情報復元
                //※一時的にローカル変数に展開して実際のデータに反映させる
                CHARA_ABILITY_DATA_FOR_SAVE chara_ability_data_for_save;
                //デシリアライズ
                arc >> pair("charaAbility", chara_ability_data_for_save);
                //アビリティ追加
                obj.addAbility(chara_ability_data_for_save.m_abilityId);
            }
        }
    };
}
```

【シリアライズ専用構造体 CHARA_ABILITY_DATA_FOR_SAVE のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
    //--------------------
    //シリアライズ処理：CHARA_ABILITY_DATA_FOR_SAVE
    template<class Arc>
    struct serialize<Arc, CHARA_ABILITY_DATA_FOR_SAVE> {
        void operator()(Arc& arc, const CHARA_ABILITY_DATA_FOR_SAVE& obj, const CVersion& ver,
                                                                        const CVersion& now_ver)
        {
            arc & pair("id", obj.m_abilityId);
        }
    };
}
```

● シリアライズ／デシリアライズの定義：進行＆フラグデータ

【CPhaseAndFlags のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
    //--------------------
    //シリアライズ処理：CPhaseAndFlags
    template<class Arc>
    struct serialize<Arc, CPhaseAndFlags> {
        void operator()(Arc& arc, const CPhaseAndFlags& obj, const CVersion& ver, const CVersion& now_ver)
        {
            arc & pair("phase", obj.m_phase);
            arc & pair("flags", obj.m_flags);
        }
    };
}
```

● 全体を統括するセーブデータの追加

ここまでで、各クラス・構造体に対してシリアライズの定義を行ったので、これらをセーブデータとしてひとまとめにするための処理を追加する。

セーブデータ統括用の専用クラスを定義し、「コレクター」と「ディストリビュータ」を定義することで対応する。

サンプルとしては若干複雑になるが、「部分ロード」を実現するための処理も合わせて実装する。

【セーブデータ統括用クラスの定義】

```
//--------------------
//セーブデータ統括用クラス
//※中身が空のクラスでよい
//※これに対するコレクターとディストリビュータを定義して使用する
class CSaveData{};
```

【CSaveData のコレクター（収集処理）】

```
namespace serial{
	//--------------------
	//収集処理：CSaveData
	template<class Arc>
	struct collector<Arc, CSaveData> {
		void operator()(Arc& arc, const CSaveData& obj, const CVersion& ver)
		{
			//インベントリのデータを収集
			CSingleton<CInventory> inventory;
			for (auto item_data : *inventory)
			{
				//アイテムデータを一つずつシリアライズ
				arc << pair("item", *item_data);//名前が重要
			}
			//アビリティのデータを収集
			CSingleton<CAbilityList> ability_list;
			for (auto ability_data : *ability_list)
			{
				//アビリティデータを一つずつシリアライズ
				arc << pair("ability", *ability_data);
			}
			//キャラのデータを収集
			CSingleton<CCharaList> chara_list;
			for (auto chara_data : *chara_list)
			{
				//キャラデータを一つずつシリアライズ
				arc << pair("chara", *chara_data);
			}
			//進行＆フラグデータをシリアライズ
			CSingleton<CPhaseAndFlags> phase_and_flags;
			arc << pair("phase_and_flags", *phase_and_flags);
		}
	};
}
```

【CSaveData の部分ロード用処理】

```
namespace serial{
	//--------------------
	//部分ロード対応処理：CSaveData
	crc32_t s_loadTarget;//ロード対象
	void setLoadTarget(const crc32_t loada_target)//ロード対象をセット
	{
		s_loadTarget = loada_target;
	}
	void setLoadTarget(const char* name)//ロード対象をセット
	{
		setLoadTarget(calcCRC32(name));
	}
	void resetLoadTarget()//ロード対象をリセット
	{
		s_loadTarget = 0;
	}
	bool isLoadTarget(const crc32_t name_crc)//ロード対象か？
	{
		return s_loadTarget == 0 || s_loadTarget == name_crc;
	}
	bool isLoadTarget(const char* name)//ロード対象か？
```

```
	{
		return isLoadTarget(calcCRC32(name));
	}
	bool isLoadTarget(const CItemBase& item)//ロード対象か？
	{
		return isLoadTarget(item.m_nameCrc);
	}
	bool isPartLoad()//部分ロードか？
	{
		return s_loadTarget != 0;
	}
	bool isPartLoad(const crc32_t name_crc)//部分ロードか？かつ、その対象項目か？
	{
		return isPartLoad() && isLoadTarget(name_crc);
	}
	bool isPartLoad(const char* name)//部分ロードか？かつ、その対象項目か？
	{
		return isPartLoad() && isLoadTarget(name);
	}
}
```

【CSaveData のディストリビュート前処理（分配前処理）】

```
namespace serial{
	//--------------------
	//分配前処理：CSaveData
	template<class Arc>
	struct beforeDistribute<Arc, CSaveData> {
		void operator()(Arc& arc, CSaveData& obj, const std::size_t array_num_on_save_data,
					const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
		{
			if (isLoadTarget("item"))//全体ロードもしくは部分ロードの対象なら実行
			{
				//インベントリデータクリア
				CSingleton<CInventory> inventory;
				inventory.destroy();
			}
			if (isLoadTarget("ability"))//全体ロードもしくは部分ロードの対象なら実行
			{
				//アビリティデータクリア
				CSingleton<CAbilityList> ability_list;
				ability_list.destroy();
			}
			if (isLoadTarget("chara"))//全体ロードもしくは部分ロードの対象なら実行
			{
				//キャラデータクリア
				CSingleton<CCharaList> chara_list;
				chara_list.destroy();
			}
			if (isLoadTarget("ability") || isLoadTarget("chara"))//全体ロードもしくは部分ロードの対象なら実行
			{
				//キャラ習得アビリティデータクリア
				CSingleton<CCharaAbilityBuff> chara_ability_list;
				chara_ability_list.destroy();
			}
			if (isLoadTarget("phase_and_flags"))//全体ロードもしくは部分ロードの対象なら実行
			{
				//フェーズ＆進行データクリア
				CSingleton<CPhaseAndFlags> phase_and_flags;
				phase_and_flags.destroy();
			}
		}
	};
}
```

【CSaveData のディストリビュータ（分配処理）】

```
namespace serial{
	//--------------------
	//分配処理：CSaveData
	template<class Arc>
	struct distributor<Arc, CSaveData> {
		void operator()(Arc& arc, CSaveData& obj, const std::size_t array_num_on_save_data,
				const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver,
				                                                const CItemBase& target_item)
		{
			if (target_item == "item" && isLoadTarget(target_item))//対象項目かつロード対象なら実行
			{
				//インベントリデータ復元
				CSingleton<CInventory> inventory;
				ITEM_DATA item_data;
				arc >> pair("item", item_data);//デシリアライズ
				inventory->regist(item_data);//インベントリに登録
			}
			else if (target_item == "ability" && isLoadTarget(target_item))//対象項目かつロード対象なら実行
			{
				//アビリティデータ復元
				CSingleton<CAbilityList> ability_list;
				ABILITY_DATA ability_data;
				arc >> pair("ability", ability_data); //デシリアライズ
				ability_list->regist(ability_data);//アビリティリストに登録
			}
			else if (target_item == "chara" && isLoadTarget(target_item))//対象項目かつロード対象なら実行
			{
				//キャラデータ復元
				CSingleton<CCharaList> chara_list;
				CHARA_DATA chara_data;
				arc >> pair("chara", chara_data);//デシリアライズ
				chara_list->regist(chara_data);//キャラリストに登録
			}
			else if (target_item == "phase_and_flags" && isLoadTarget(target_item))//対象項目かつロード対象なら実
			{
				//フェーズ＆進行データ復元
				CSingleton<CPhaseAndFlags> phase_and_flags;
				arc >> pair("phase_and_flags", *phase_and_flags);//デシリアライズ
			}
		}
	};
}
```

【CSaveData のディストリビュート後処理（分配後処理）】

```
namespace serial{
	//--------------------
	//分配後処理：CSaveData
	template<class Arc>
	struct afterDistribute<Arc, CSaveData> {
		void operator()(Arc& arc, CSaveData& obj, const std::size_t array_num_on_save_data,
				const std::size_t array_num_loaded, const CVersion& ver, const CVersion& now_ver)
		{
			//部分ロードにより、インベントリデータだけ復元してキャラデータがそのままなら武器と盾を参照し直す
			if (isPartLoad("item") && !isPartLoad("chara"))
			{
				CSingleton<CCharaList> chara_list;
				for (auto& chara_data : *chara_list)
				{
					chara_data->attachItems();//アイテムを参照し直す
				}
			}
		}
	};
}
```

● シリアライズ実行処理

実際にシリアライズを実行してファイルに書き込む処理を作成する。

アーカイブ形式をバイナリとテキストで切り替えできるように、アーカイブクラスをテンプレート引数で受け取るテンプレート関数にする。

【シリアライズ実行】※テンプレート関数

```
//--------------------
//シリアライズ
template<class Arc>
std::size_t serialize(void* save_data_buff, const std::size_t save_data_buff_size)
{
	printf("------------------------------------------------------------¥n");
	printf("【シリアライズ】¥n");

	//ワークバッファ準備
	const std::size_t WORK_BUFF_SIZE = 1 * 1024 * 1024;
	void* work_buff = new char[WORK_BUFF_SIZE];

	//シリアライズ用アーカイブオブジェクト生成
	Arc arc(save_data_buff, save_data_buff_size, work_buff, WORK_BUFF_SIZE);

	//シリアライズ
	arc << serial::pair<CSaveData>("SaveData");//CSaveDataクラスのシリアライズを実行

	//シリアライズの結果を表示
	serial::CResult result = arc.getResult();
	printf("シリアライズ結果：¥n");
	printf("  致命的なエラー ... %s¥n", result.hasFatalError() ? "発生！" : "なし");

	//セーブデータのサイズを返す
	return result.getSaveDataSize();
}
```

【ファイル書き込み処理】※セーブデータのイメージとファイルパスを受け取ってファイルに保存する

```
//--------------------
//ファイル書き込み
void writeFile(const char* file_path, const void* file_image, const std::size_t file_image_size)
{
	printf("------------------------------------------------------------¥n");
	printf("【ファイル書き込み】¥n");
	printf("file=¥"%s¥", size=%d¥n", file_path, file_image_size);
	if (file_image_size == 0)
	{
		printf("Save data not found!¥n");
		return;
	}

	//ファイルに保存
	FILE* fp;
#ifdef USE_STRCPY_S
	fopen_s(&fp, file_path, "wb");
#else//USE_STRCPY_S
	fp = fopen(file_path, "wb");
#endif//USE_STRCPY_S
	fwrite(file_image, 1, file_image_size, fp);
	fclose(fp);
}
```

【セーブ処理（シリアライズ＆ファイル書き込み）】※テンプレート関数：保存先のファイルパスは外部指定

```
//--------------------
//セーブ
```

```
template<class Arc>
void save(const char* file_path)
{
	//セーブデータ用バッファ準備
	const std::size_t SAVE_DATA_BUFF_SIZE = 1 * 1024 * 1024;
	void* save_data_buff = new char[SAVE_DATA_BUFF_SIZE];

	//シリアライズ
	const std::size_t save_data_size = serialize<Arc>(save_data_buff, SAVE_DATA_BUFF_SIZE);

	//ファイル書き込み
	writeFile(file_path, save_data_buff, save_data_size);
}
```

● デシリアライズ実行処理

ファイルからセーブデータを読み込んでデシリアライズを実行する処理を作成する。

【デシリアライズ実行】※テンプレート関数

```
//-------------------
//デシリアライズ
template<class Arc>
void deserialize(void* save_data, const std::size_t save_data_size)
{
	printf("-----------------------------------------------------------------¥n");
	printf("【デシリアライズ】¥n");

	//ワークバッファ準備
	const std::size_t WORK_BUFF_SIZE = 1 * 1024 * 1024;
	void* work_buff = new char[WORK_BUFF_SIZE];

	//デシリアライズ用アーカイブオブジェクト生成
	Arc arc(save_data, save_data_size, work_buff, WORK_BUFF_SIZE);

	//デシリアライズ
	arc >> serial::pair<CSaveData>("SaveData");//CSaveData クラスのデシリアライズを実行

	//デシリアライズの結果を表示
	serial::CResult result = arc.getResult();
	printf("デシリアライズ結果：¥n");
	printf("  致命的なエラー ... %s¥n", result.hasFatalError() ? "発生！" : "なし");
}
```

【ファイル読み込み処理】※セーブデータイメージ用のバッファとファイルパスを受け取ってファイルから読み込む

```
//--------------------
//ファイルサイズ取得
std::size_t getFileSize(FILE* fp)
{
	fseek(fp, 0, SEEK_END);
	const std::size_t file_size = ftell(fp);
	fseek(fp, 0, SEEK_SET);
	return file_size;
}
//--------------------
//ファイル読み込み
std::size_t readFile(const char* file_path, void* file_image_buff, const std::size_t file_image_buff_size_max)
{
	printf("-----------------------------------------------------------------¥n");
	printf("【ファイル読み込み】¥n");

	//ファイルの内容をまとめて読み込み
	FILE* fp;
#ifdef USE_STRCPY_S
```

```
	fopen_s(&fp, file_path, "rb");
#else//USE_STRCPY_S
	fp = fopen(file_path, "rb");
#endif//USE_STRCPY_S
	std::size_t file_image_size = getFileSize(fp);
	printf("file=¥"%s¥", size=%d¥n", file_path, file_image_size);
	if (file_image_size <= file_image_buff_size_max)
		fread(file_image_buff, 1, file_image_size, fp);
	else
		file_image_size = 0;
	fclose(fp);
	return file_image_size;
}
```

【ロード処理（ファイル読み込み＆デシリアライズ）】※テンプレート関数：読み込み先のファイルパスは外部指定

```
//--------------------
//ロード
template<class Arc>
void load(const char* file_path)
{
	//セーブデータ用バッファ準備
	const std::size_t SAVE_DATA_BUFF_SIZE = 1 * 1024 * 1024;
	void* save_data_buff = new char[SAVE_DATA_BUFF_SIZE];

	//ファイル読み込み
	const std::size_t save_data_size = readFile(file_path, save_data_buff, SAVE_DATA_BUFF_SIZE);

	//デシリアライズ
	deserialize<Arc>(save_data_buff, save_data_size);
}
```

● テスト処理

以上の処理を使用し、シリアライズとデシリアライズのテストを実行する。

【テスト処理】

```
//--------------------
//シリアライズ＆デシリアライズテスト
void testDataSerialize()
{
	//-----
	//セーブデータファイルパス
	const char* save_file_name_bin = "save_data.bin";//バイナリ形式用
	const char* save_file_name_txt = "save_data.txt";//テキスト形式用

	//-----
	//データを作成してセーブ

	//全データリセット
	resetAll();
	//テストデータ作成(pattern=0)
	makeTestData(0);
	//現在のデータを表示
	printDataAll();
	//セーブ（シリアライズ＆ファイル書き込み）
	save<serial::COBinaryArchive>(save_file_name_bin);//バイナリ形式を使用
	save<serial::COTextArchive>(save_file_name_txt);//テキスト形式を使用

	//-----
	//データをリセットしロード

	//全データリセット
	resetAll();
```

```
	//現在のデータを表示
	printDataAll();
	//ロード（ファイル読み込み＆デシリアライズ）
	load<serial::CIBinaryArchive>(save_file_name_bin);
	//現在のデータを表示
	printDataAll();

	//-----
	//部分ロードのテストのために、セーブデータと異なるデータを作成

	//全データリセット
	resetAll();
	//テストデータ作成(pattern=1)
	makeTestData(1);
	//現在のデータを表示
	printDataAll();

	//-----
	//部分ロード処理

	//部分ロード指定
	//serial::setLoadTarget("phase_and_flags");
	serial::setLoadTarget("chara");
	//ロード（ファイル読み込み＆デシリアライズ）
	load<serial::CIBinaryArchive>(save_file_name_bin);
	//現在のデータを表示
	printDataAll();
	//部分ロード指定解除
	serial::resetLoadTarget();
}
```

【テストメイン】

```
//--------------------
//テスト
int main(const int argc, const char* argv[])
{
	testDataSerialize();
	return EXIT_SUCCESS;
}
```

↓（実行結果）

```
------------------------------------------------------------
【全データリセット】
------------------------------------------------------------
【テストデータ作成】(pattern=0)                    ↓このデータをシリアライズする
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
ID="r00010"(0x62cd9fc0), name="回復薬 001", atk=0, def=0, recover=5, num=10
ID="r00020"(0x49e0cc03), name="回復薬 002", atk=0, def=0, recover=7, num=10
ID="r00030"(0x50fbfd42), name="回復薬 003", atk=0, def=0, recover=9, num=10
ID="s00010"(0xa9914c65), name="盾 001", atk=0, def=5, recover=0, num=1
ID="s00020"(0x82bc1fa6), name="盾 002", atk=0, def=6, recover=0, num=1
                         title="グレード:B"
ID="s00030"(0x9ba72ee7), name="盾 003", atk=0, def=7, recover=0, num=1
                         title="グレード:C"
ID="s00040"(0xd4e6b820), name="盾 004", atk=0, def=8, recover=0, num=1
                         title="グレード:D"
ID="s00050"(0xcdfd8961), name="盾 005", atk=0, def=9, recover=0, num=1
                         title="グレード:E"
ID="w00010"(0x32000e73), name="武器 001", atk=10, def=0, recover=0, num=1
ID="w00020"(0x192d5db0), name="武器 002", atk=11, def=0, recover=0, num=1
                         title="性能:+1"
ID="w00030"(0x00366cf1), name="武器 003", atk=12, def=1, recover=0, num=1
                         title="性能:+2"
```

```
ID="w00040"(0x4f77fa36), name="武器 004", atk=13, def=1, recover=0, num=1
                        title="性能:+3"
ID="w00050"(0x566ccb77), name="武器 005", atk=14, def=2, recover=0, num=1
                        title="性能:+4"
--- アビリティ ---
ID="a00010"(0xe78feff5), name="特技 001", atk=5, def=4
ID="a00020"(0xcca2bc36), name="特技 002", atk=8, def=6
ID="a00030"(0xd5b98d77), name="特技 003", atk=11, def=8
ID="a00040"(0x9af81bb0), name="特技 004", atk=14, def=10
ID="a00050"(0x83e32af1), name="特技 005", atk=17, def=12
ID="a00060"(0xa8ce7932), name="特技 006", atk=20, def=14
ID="a00070"(0xb1d54873), name="特技 007", atk=23, def=16
ID="a00080"(0x364d54bc), name="特技 008", atk=26, def=18
ID="a00090"(0x2f5665fd), name="特技 009", atk=29, def=20
ID="a00100"(0xff56b483), name="特技 010", atk=32, def=22
--- キャラ ---
ID="c00030"(0x98712c7c), name="さぶろう", level=30, atk=55, def=3
                        param1={123, 456}, param2={987, 654}
                        weapon=武器 001
                        shield=盾 003
                        [特技 005]
ID="c00020"(0x816a1d3d), name="じろう", level=20, atk=25, def=40
                        param1={999, 888}, param2={777, 666}
                        weapon=武器 005
                        shield=盾 001
ID="c00010"(0xaa474efe), name="たろう", level=10, atk=15, def=20
                        param1={111, 222}, param2={333, 444}
                        weapon=武器 002
                        shield=盾 002
                        [特技 003]
                        [特技 002]
                        [特技 005]
                        [特技 001]
--- 進行データ ---
pahse=7
flags=
 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
------------------------------------------------------------
【シリアライズ】                                ←バイナリ形式シリアライズ
シリアライズ結果：
  致命的なエラー ... なし
------------------------------------------------------------
【ファイル書き込み】                            ←バイナリ形式セーブデータの保存
file="save_data.bin", size=7117
------------------------------------------------------------
【シリアライズ】                                ←テキスト形式シリアライズ
シリアライズ結果：
  致命的なエラー ... なし
------------------------------------------------------------
【ファイル書き込み】                            ←テキスト形式セーブデータの保存
file="save_data.txt", size=65936
------------------------------------------------------------
【全データリセット】                            ↓一旦データを消去（デシリアライズのテストのため）
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
--- アビリティ ---
--- キャラ ---
--- 進行データ ---
pahse=0
flags=
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
------------------------------------------------------------
【ファイル読み込み】                                    ←バイナリ形式セーブデータのファイル読み込み
file="save_data.bin", size=7117
------------------------------------------------------------
【デシリアライズ】                                      ←バイナリ形式デシリアライズ
デシリアライズ結果：
  致命的なエラー ... なし                               ↓すべてのデータが復元に成功している
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
ID="r00010"(0x62cd9fc0), name="回復薬 001", atk=0, def=0, recover=5, num=10
ID="r00020"(0x49e0cc03), name="回復薬 002", atk=0, def=0, recover=7, num=10
ID="r00030"(0x50fbfd42), name="回復薬 003", atk=0, def=0, recover=9, num=10
ID="s00010"(0xa9914c65), name="盾 001", atk=0, def=5, recover=0, num=1
ID="s00020"(0x82bc1fa6), name="盾 002", atk=0, def=6, recover=0, num=1
ID="s00030"(0x9ba72ee7), name="盾 003", atk=0, def=7, recover=0, num=1
ID="s00040"(0xd4e6b820), name="盾 004", atk=0, def=8, recover=0, num=1
ID="s00050"(0xcdfd8961), name="盾 005", atk=0, def=9, recover=0, num=1
ID="w00010"(0x32000e73), name="武器 001", atk=10, def=0, recover=0, num=1
ID="w00020"(0x192d5db0), name="武器 002", atk=11, def=0, recover=0, num=1
ID="w00030"(0x00366cf1), name="武器 003", atk=12, def=1, recover=0, num=1
ID="w00040"(0x4f77fa36), name="武器 004", atk=13, def=1, recover=0, num=1
ID="w00050"(0x566ccb77), name="武器 005", atk=14, def=2, recover=0, num=1
--- アビリティ ---
ID="a00010"(0xe78feff5), name="特技 001", atk=5, def=4
ID="a00020"(0xcca2bc36), name="特技 002", atk=8, def=6
ID="a00030"(0xd5b98d77), name="特技 003", atk=11, def=8
ID="a00040"(0x9af81bb0), name="特技 004", atk=14, def=10
ID="a00050"(0x83e32af1), name="特技 005", atk=17, def=12
ID="a00060"(0xa8ce7932), name="特技 006", atk=20, def=14
ID="a00070"(0xb1d54873), name="特技 007", atk=23, def=16
ID="a00080"(0x364d54bc), name="特技 008", atk=26, def=18
ID="a00090"(0x2f5665fd), name="特技 009", atk=29, def=20
ID="a00100"(0xff56b483), name="特技 010", atk=32, def=22
--- キャラ ---
ID="c00030"(0x98712c7c), name="さぶろう", level=30, atk=55, def=3
                            param1={123, 456}, param2={987, 654}
                            weapon=武器 001
                            shield=盾 003
                            [特技 005]
ID="c00020"(0x816a1d3d), name="じろう", level=20, atk=25, def=40
                            param1={999, 888}, param2={777, 666}
                            weapon=武器 005
                            shield=盾 001
ID="c00010"(0xaa474efe), name="たろう", level=10, atk=15, def=20
                            param1={111, 222}, param2={333, 444}
                            weapon=武器 002
                            shield=盾 002
                            [特技 003]
                            [特技 002]
                            [特技 005]
                            [特技 001]
--- 進行データ ---
pahse=7
flags=
 1 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
------------------------------------------------------------
【全データリセット】
------------------------------------------------------------
```

```
【テストデータ作成】(pattern=1)                    ↓セーブデータの状態と異なるデータを作成
------------------------------------------------------------
【現在のデータを表示】
--- インベントリ ---
ID="r00020"(0x49e0cc03), name="回復薬 002", atk=0, def=0, recover=7, num=10
ID="r00030"(0x50fbfd42), name="回復薬 003", atk=0, def=0, recover=9, num=10
ID="r00040"(0x1fba6b85), name="回復薬 004", atk=0, def=0, recover=11, num=10
ID="r00050"(0x06a15ac4), name="回復薬 005", atk=0, def=0, recover=13, num=10
ID="r00060"(0x2d8c0907), name="回復薬 006", atk=0, def=0, recover=15, num=10
ID="s00010"(0xa9914c65), name="盾 001", atk=0, def=5, recover=0, num=1
ID="s00030"(0x9ba72ee7), name="盾 003", atk=0, def=7, recover=0, num=1
                         title="グレード:C"
ID="s00050"(0xcdfd8961), name="盾 005", atk=0, def=9, recover=0, num=1
                         title="グレード:E"
ID="s00070"(0xffcbebe3), name="盾 007", atk=0, def=11, recover=0, num=1
                         title="グレード:G"
ID="s00090"(0x6148c66d), name="盾 009", atk=0, def=13, recover=0, num=1
                         title="グレード:I"
ID="s00110"(0xa8532652), name="盾 011", atk=0, def=15, recover=0, num=1
                         title="グレード:K"
ID="s00130"(0x9a6544d0), name="盾 013", atk=0, def=17, recover=0, num=1
                         title="グレード:M"
ID="s00150"(0xcc3fe356), name="盾 015", atk=0, def=19, recover=0, num=1
                         title="グレード:O"
ID="w00010"(0x32000e73), name="武器 001", atk=10, def=0, recover=0, num=1
ID="w00030"(0x00366cf1), name="武器 003", atk=12, def=1, recover=0, num=1
                         title="性能:+2"
ID="w00050"(0x566ccb77), name="武器 005", atk=14, def=2, recover=0, num=1
                         title="性能:+4"
ID="w00070"(0x645aa9f5), name="武器 007", atk=16, def=3, recover=0, num=1
                         title="性能:+6"
ID="w00090"(0xfad9847b), name="武器 009", atk=18, def=4, recover=0, num=1
                         title="性能:+8"
ID="w00110"(0x33c26444), name="武器 011", atk=20, def=5, recover=0, num=1
                         title="性能:+10"
ID="w00130"(0x01f406c6), name="武器 013", atk=22, def=6, recover=0, num=1
                         title="性能:+12"
ID="w00150"(0x57aea140), name="武器 015", atk=24, def=7, recover=0, num=1
                         title="性能:+14"
--- アビリティ ---
ID="a00020"(0xcca2bc36), name="特技 002", atk=8, def=6
ID="a00030"(0xd5b98d77), name="特技 003", atk=11, def=8
ID="a00040"(0x9af81bb0), name="特技 004", atk=14, def=10
ID="a00050"(0x83e32af1), name="特技 005", atk=17, def=12
ID="a00060"(0xa8ce7932), name="特技 006", atk=20, def=14
ID="a00070"(0xb1d54873), name="特技 007", atk=23, def=16
ID="a00080"(0x364d54bc), name="特技 008", atk=26, def=18
ID="a00090"(0x2f5665fd), name="特技 009", atk=29, def=20
ID="a00100"(0xff56b483), name="特技 010", atk=32, def=22
ID="a00110"(0xe64d85c2), name="特技 011", atk=35, def=24
ID="a00120"(0xcd60d601), name="特技 012", atk=38, def=26
ID="a00130"(0xd47be740), name="特技 013", atk=41, def=28
ID="a00140"(0x9b3a7187), name="特技 014", atk=44, def=30
ID="a00150"(0x822140c6), name="特技 015", atk=47, def=32
ID="a00160"(0xa90c1305), name="特技 016", atk=50, def=34
ID="a00170"(0xb0172244), name="特技 017", atk=53, def=36
ID="a00180"(0x378f3e8b), name="特技 018", atk=56, def=38
ID="a00190"(0x2e940fca), name="特技 019", atk=59, def=40
ID="a00200"(0xfd100ada), name="特技 020", atk=62, def=42
--- キャラ ---
ID="c00010"(0xaa474efe), name="たろう", level=11, atk=15, def=20
                         param1={111, 222}, param2={333, 444}
                         weapon=武器 003
                         shield=盾 003
                         [特技 003]
```

```
ID="c00020"(0x816a1d3d), name="じろう", level=21, atk=25, def=40
                        param1={999, 888}, param2={777, 666}
                        weapon=武器 005
                        shield=盾 001
ID="c00040"(0xd730babb), name="しろう", level=41, atk=55, def=3
                        param1={123, 456}, param2={987, 654}
                        weapon=武器 009
                        shield=盾 009
                        [特技 002]
ID="c00030"(0x98712c7c), name="さぶろう", level=31, atk=55, def=3
                        param1={123, 456}, param2={987, 654}
                        weapon=武器 007
                        shield=盾 005
                        [特技 005]
                        [特技 004]
--- 進行データ ---
pahse=4
flags=
 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
                                                    ※ここで「キャラ」のみを部分ロードするように設定
------------------------------------------------------------
【ファイル読み込み】                                  ←バイナリ形式セーブデータのファイル読み込み
file="save_data.bin", size=7117
------------------------------------------------------------
【デシリアライズ】                                    ←バイナリ形式デシリアライズ
デシリアライズ結果：
  致命的なエラー ... なし                             ↓キャラデータだけ以前の状態に復元している
------------------------------------------------------------    （他のデータはそのまま）
【現在のデータを表示】
--- インベントリ ---
ID="r00020"(0x49e0cc03), name="回復薬 002", atk=0, def=0, recover=7, num=10
ID="r00030"(0x50fbfd42), name="回復薬 003", atk=0, def=0, recover=9, num=10
ID="r00040"(0x1fba6b85), name="回復薬 004", atk=0, def=0, recover=11, num=10
ID="r00050"(0x06a15ac4), name="回復薬 005", atk=0, def=0, recover=13, num=10
ID="r00060"(0x2d8c0907), name="回復薬 006", atk=0, def=0, recover=15, num=10
ID="s00010"(0xa9914c65), name="盾 001", atk=0, def=5, recover=0, num=1
ID="s00030"(0x9ba72ee7), name="盾 003", atk=0, def=7, recover=0, num=1
                        title="グレード:C"
ID="s00050"(0xcdfd8961), name="盾 005", atk=0, def=9, recover=0, num=1
                        title="グレード:E"
ID="s00070"(0xffcbebe3), name="盾 007", atk=0, def=11, recover=0, num=1
                        title="グレード:G"
ID="s00090"(0x6148c66d), name="盾 009", atk=0, def=13, recover=0, num=1
                        title="グレード:I"
ID="s00110"(0xa8532652), name="盾 011", atk=0, def=15, recover=0, num=1
                        title="グレード:K"
ID="s00130"(0x9a6544d0), name="盾 013", atk=0, def=17, recover=0, num=1
                        title="グレード:M"
ID="s00150"(0xcc3fe356), name="盾 015", atk=0, def=19, recover=0, num=1
                        title="グレード:O"
ID="w00010"(0x32000e73), name="武器 001", atk=10, def=0, recover=0, num=1
ID="w00030"(0x00366cf1), name="武器 003", atk=12, def=1, recover=0, num=1
                        title="性能:+2"
ID="w00050"(0x566ccb77), name="武器 005", atk=14, def=2, recover=0, num=1
                        title="性能:+4"
ID="w00070"(0x645aa9f5), name="武器 007", atk=16, def=3, recover=0, num=1
                        title="性能:+6"
ID="w00090"(0xfad9847b), name="武器 009", atk=18, def=4, recover=0, num=1
                        title="性能:+8"
ID="w00110"(0x33c26444), name="武器 011", atk=20, def=5, recover=0, num=1
                        title="性能:+10"
ID="w00130"(0x01f406c6), name="武器 013", atk=22, def=6, recover=0, num=1
```

```
                              title="性能:+12"
ID="w00150"(0x57aea140), name="武器 015", atk=24, def=7, recover=0, num=1
                              title="性能:+14"
--- アビリティ ---
ID="a00020"(0xcca2bc36), name="特技 002", atk=8, def=6
ID="a00030"(0xd5b98d77), name="特技 003", atk=11, def=8
ID="a00040"(0x9af81bb0), name="特技 004", atk=14, def=10
ID="a00050"(0x83e32af1), name="特技 005", atk=17, def=12
ID="a00060"(0xa8ce7932), name="特技 006", atk=20, def=14
ID="a00070"(0xb1d54873), name="特技 007", atk=23, def=16
ID="a00080"(0x364d54bc), name="特技 008", atk=26, def=18
ID="a00090"(0x2f5665fd), name="特技 009", atk=29, def=20
ID="a00100"(0xff56b483), name="特技 010", atk=32, def=22
ID="a00110"(0xe64d85c2), name="特技 011", atk=35, def=24
ID="a00120"(0xcd60d601), name="特技 012", atk=38, def=26
ID="a00130"(0xd47be740), name="特技 013", atk=41, def=28
ID="a00140"(0x9b3a7187), name="特技 014", atk=44, def=30
ID="a00150"(0x822140c6), name="特技 015", atk=47, def=32
ID="a00160"(0xa90c1305), name="特技 016", atk=50, def=34
ID="a00170"(0xb0172244), name="特技 017", atk=53, def=36
ID="a00180"(0x378f3e8b), name="特技 018", atk=56, def=38
ID="a00190"(0x2e940fca), name="特技 019", atk=59, def=40
ID="a00200"(0xfd100ada), name="特技 020", atk=62, def=42
--- キャラ ---                                          ※直前まで存在していた「しろう」が消えている
ID="c00030"(0x98712c7c), name="さぶろう", level=30, atk=55, def=3   ※並び順が戻っている
                              param1={123, 456}, param2={987, 654}  ※各パラメータが戻っている
                              weapon=武器 001                       ※古いデータで装備していた「たろう」の武器と
                              shield=盾 003                           盾は、新しいインベントリに存在しないので
                              [特技 005]                              装備解除されている
ID="c00020"(0x816a1d3d), name="じろう", level=20, atk=25, def=40    ※古いデータで習得していた「たろう」の一部の
                              param1={999, 888}, param2={777, 666}    アビリティは、新しいアビリティリストに
                              weapon=武器 005                         存在しないので削除されている
                              shield=盾 001
ID="c00010"(0xaa474efe), name="たろう", level=10, atk=15, def=20
                              param1={111, 222}, param2={333, 444}
                              [特技 003]
                              [特技 002]
                              [特技 005]
--- 進行データ ---
pahse=4
flags=
 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
```

【セーブデータファイル：テキスト形式】※かなり大きなファイルになる

```
{"serializer": {
 "SaveData": {"crc": 0x3366f9bd, "itemType": "class CSaveData", "itemSize": 0, "isObj": 1, "isArr": 0, "isPtr": 0,
                                                                "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "0.0",
    "elem": {},
  "item": {"crc": 0x1f1b251e, "itemType": "struct ITEM_DATA", "itemSize": 80, "isObj": 1, "isArr": 0, "isPtr": 0,
                                                                "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "1.0",
     "elem": {
       "id": {"crc": 0xbf396750, "itemType": "class CStrWithCRC<8>", "itemSize": 16, "isObj": 1, "isArr": 0,
                                             "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "0.0",
           "elem": {
             "len": {"crc": 0x0c0c38eb, "itemType": "unsigned char", "itemSize": 1, "isObj": 0, "isArr": 0,
                                             "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 0x06},
             "str": {"crc": 0x5caea3f9, "itemType": "struct serial::str_t", "itemSize": 7, "isObj": 0,
                              "isArr": 0, "isPtr": 0, "isNul": 0, "isVLen": 1, "hasVer": 0, "data": "r00010"},
             "crc": {"crc": 0x7c6287fd, "itemType": "unsigned int", "itemSize": 4, "isObj": 0, "isArr": 0,
                                           "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 0x62cd9fc0}}},
       "name": {"crc": 0x5e237e06, "itemType": "class CStr<32>", "itemSize": 33, "isObj": 1, "isArr": 0,
                                             "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "0.0",
```

```
            "elem": {
              "len": {"crc": 0x0c0c38eb, "itemType": "unsigned char", "itemSize": 1, "isObj": 0, "isArr": 0,
                                          "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 0x09},
              "str": {"crc": 0x5caea3f9, "itemType": "struct serial::str_t", "itemSize": 10, "isObj": 0,
                                  "isArr": 0, "isPtr": 0, "isNul": 0, "isVLen": 1, "hasVer": 0, "data": "回復薬001"}}},
        "title": {"crc": 0x2b36786b, "itemType": "struct serial::str_t", "itemSize": 0, "isObj": 0,
                                    "isArr": 0, "isPtr": 1, "isNul": 1, "isVLen": 1, "hasVer": 0, "data": null},
        "basic": {"crc": 0x90797553, "itemType": "struct BASIC_DATA", "itemSize": 4, "isObj": 1, "isArr": 0,
                                          "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "1.0",
            "elem": {
              "atk": {"crc": 0x27677c27, "itemType": "short", "itemSize": 2, "isObj": 0, "isArr": 0,
                                          "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 0},
              "def": {"crc": 0x0cc4e161, "itemType": "short", "itemSize": 2, "isObj": 0, "isArr": 0,
                                          "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 0}}},
        "recover": {"crc": 0xf588f14d, "itemType": "short", "itemSize": 2, "isObj": 0, "isArr": 0, "isPtr": 0,
                                          "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 5},
        "num": {"crc": 0xdc43af6e, "itemType": "short", "itemSize": 2, "isObj": 0, "isArr": 0, "isPtr": 0,
                                          "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 10}}},
…（略）…（ここまで↑がアイテムデータ１件分のデータ）
    "phase_and_flags": {"crc": 0x4659bc29, "itemType": "class CPhaseAndFlags", "itemSize": 16, "isObj": 1,
                                "isArr": 0, "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "1.0",
        "elem": {
          "phase": {"crc": 0xb1bdd6cb, "itemType": "short", "itemSize": 2, "isObj": 0, "isArr": 0, "isPtr": 0,
                                                  "isNul": 0, "isVLen": 0, "hasVer": 0, "data": 7},
          "flags": {"crc": 0x0b0541ba, "itemType": "class std::bitset<64>", "itemSize": 8, "isObj": 1,
                                    "isArr": 0, "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0, "ver": "0.0",
            "elem": {
              "bitset": {"crc": 0x9dd4d1cb, "itemType": "struct serial::bin_t", "itemSize": 8, "isObj": 0,
                                        "isArr": 0, "isPtr": 0, "isNul": 0, "isVLen": 0, "hasVer": 0,
                                                  "data": [0x35,0x00,0x00,0x00,0x00,0x00,0x00,0x80]}}}}}}}
}, "terminator": "ok"}
```


▼ システムの依存関係

このサンプルプログラムは、別紙に掲載する他のサンプルプログラムを必要とする。
下記のプログラムである。

- プールアロケータクラス
  - 「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。
- スタックアロケータクラス／スタックアロケータアダプタークラス
  - 別紙の「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。
- 多態アロケータクラス／グローバル多態アロケータ
  - 別紙の「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。

サンプルプログラム内では STL も使用しているが、この多態アロケータにより、ヒープからメモリを取られず、固定バッファで動作するようにしている。

▼ 最適化のために

データ項目ごとに全てシリアライズを設定するとバージョン整合の点で柔軟なのはいい

がメモリサイズ肥大、ファイルサイズ肥大、低速化が懸念される。

そこで、各クラスに対してセーブデータとして保存する情報を集めたサブ構造体を作って、その 1 項目だけをシリアライズする構成にするのもよい。

これだけでも、本システムのメリットは十分に得られる。読み込み時に、構造変更によって一部のクラスに不整合が起きても、他のデータが正しく読み込めるためである。また、コレクターとディストリビュータを活用して、シリアライズ処理をすっきりとまとめる意義も大きい。

なお、この場合、メンバーの順序変更やサイズ変更はできなくなり、データ項目の追加は必ず最後方に行うという、旧来の保守スタイルが必要となる。

以下、具体例な対応例を示す。

【キャラデータ構造体】

```
//--------------------
//キャラデータ構造体
struct CHARA_ABILITY_DATA;
struct CHARA_DATA
{
	crc32_t getKey() const { return m_id.getCRC(); }//キーを取得
	struct PARAMANENT{//セーブ対象項目のサブ構造体
		ID_t m_id;//ID
		name_t m_name;//名前
		char m_level;//レベル
		BASIC_DATA m_basic;//基本データ
		const ITEM_DATA* m_weapon;//武器
		const ITEM_DATA* m_shield;//盾
		int m_param1[2];//ダミーパラメータ 1
		int m_param2[2];//ダミーパラメータ 2
	} m_parmanent;
	//これ以外はシリアライズ対象にしない
	crc32_t m_weaponId;//武器 ID
	crc32_t m_shieldId;//盾 ID
	CHARA_ABILITY_DATA* m_abilities;//キャラ習得アビリティ
};
```

【CHARA_DATA のシリアライズ／デシリアライズ共通処理】

```
namespace serial{
	//--------------------
	//シリアライズ処理：CHARA_DATA
	template<class Arc>
	struct serialize<Arc, CHARA_DATA> {
		void operator()(Arc& arc, const CHARA_DATA& obj, const CVersion& ver, const CVersion& now_ver)
		{
			arc & pairBin("id", obj.m_parmanen);//シリアライズするのはこれだけ（バイトデータ扱い）
		}
	};
}
```

■ 処理実装サンプル：シリアライズの実装

シリアライズ処理のサンプルを示す。
処理仕様で説明した通りの実装である。
なお、テキスト形式の読み込みは未実装。

▼ インクルードとネームスペース

本システムはネームスペースを使用する。以降のプログラムはすべて「serial」ネームスペース上に構成する。

なお、ユーザー定義の特殊化クラスもネームスペースを合わせて定義しなければならない。

【使用インクルード】

```
#include <memory.h>//memcpy 用
#include <string.h>//strcpy 用
#include <assert.h>//assert 用
#include <cstddef>//std::size_t 用
#include <stdarg.h>//va_list 用
#include <typeinfo.h>//type_id 用
#include <time.h>//time_t, tm 用
#include <map>//STL map 用

//その他、別紙のサンプルで作成したクラス・関数を使用
//crc32_t calcCRC32(const char* str);//CRC32 計算関数
//CStackAllocatorWithBuff//スタックアロケータ
//CTempPolyStackAllocator//多態アロケータ
```

【ネームスペース】

```
//-----------------------------------------------------------------------------
//シリアライズネームスペース
namespace serial
{
}
```

【マクロ定義】

```
//デシリアライズ時に、サイズが変更されたデータ項目を読み込む時の処理方法を設定する
#define IS_LITTLE_ENDIAN//リトルエンディアン
//#define IS_BIG_ENDIAN//ビッグエンディアン
```

▼ バージョンクラス

【バージョンクラス】

```
//--------------------
//バージョンクラス
template<unsigned short MAJOR, unsigned short MINOR>
class CVersionDefBase;
class CVersion
{
public:
	//定数
	static const unsigned int VER_FIGURE = 1000000;//合成バージョン計算用桁上げ数
```

```
	enum compareEnum : char
	{
		EQ = 0,//==:EQual : 同じ
		LT = -1,//<:Less Than : 小さい
		GT = 1,//>:Greater Than : 大きい
		MINOR_LT = -2,//マイナーバージョンのみ小さい（メジャーバージョンは一致）
		MINOR_GT = 2,//マイナーバージョンのみ大きい（メジャーバージョンは一致）
	};
public:
	//アクセッサ
	unsigned int getMajor() const { return m_majorVer; };//メジャーバージョン
	unsigned int getMinor() const { return m_minorVer; };//マイナーバージョン
	unsigned int getVer()const { return m_ver; };//合成バージョン
	const unsigned int* getVerPtr()const { return &m_ver; };//合成バージョンのポインタ
	std::size_t getVerSize()const { return sizeof(m_ver); };//合成バージョンのサイズ
public:
	//オペレータ
	bool operator==(const CVersion& rhs) const { return m_ver == rhs.m_ver; }
	bool operator!=(const CVersion& rhs) const { return m_ver != rhs.m_ver; }
	bool operator<(const CVersion& rhs) const { return m_ver < rhs.m_ver; }
	bool operator<=(const CVersion& rhs) const { return m_ver <= rhs.m_ver; }
	bool operator>(const CVersion& rhs) const { return m_ver > rhs.m_ver; }
	bool operator>=(const CVersion& rhs) const { return m_ver >= rhs.m_ver; }
public:
	//キャストオペレータ
	operator unsigned int() const { return m_ver; }
public:
	//メソッド
	//比較
	compareEnum compare(CVersion& rhs)
	{
		return	m_majorVer < rhs.m_majorVer ?
				LT :
				m_majorVer > rhs.m_majorVer ?
					GT :
					m_minorVer < rhs.m_minorVer ?
						MINOR_LT :
						m_minorVer > rhs.m_minorVer ?
							MINOR_GT :
							EQ;
	}
	//バージョンからジャーバージョンとマイナーバージョンを算出
	void calcFromVer()
	{
		*const_cast<unsigned short*>(&m_majorVer) = m_ver / VER_FIGURE;
		*const_cast<unsigned short*>(&m_minorVer) = m_ver % VER_FIGURE;
	}
public:
	//デフォルトコンストラクタ
	CVersion() :
		m_majorVer(0),
		m_minorVer(0),
		m_ver(0)
	{}
	//コンストラクタ
	CVersion(const unsigned short major, const unsigned short minor) :
		m_majorVer(major),
		m_minorVer(minor),
		m_ver(major * VER_FIGURE + minor)
	{}
	//テンプレートコンストラクタ
	template<unsigned short MAJOR, unsigned short MINOR>
	CVersion(CVersionDefBase<MAJOR, MINOR>) :
		CVersion(MAJOR, MINOR)
	{}
```

```
private:
	//フィールド
	const unsigned int m_ver;//バージョン
	const unsigned short m_majorVer;//メジャーバージョン
	const unsigned short m_minorVer;//マイナーバージョン
};
```

【バージョンテンプレート基底クラス】

```
//--------------------
//バージョンテンプレート基底クラス
template<unsigned short MAJOR, unsigned short MINOR>
class CVersionDefBase
{
public:
	//定数
	static const unsigned short MAJOR_VER = MAJOR;//メジャーバージョン
	static const unsigned short MINOR_VER = MINOR;//マイナーバージョン
	static const unsigned int VER = MAJOR_VER * CVersion::VER_FIGURE + MINOR_VER;//合成バージョン
public:
	//アクセッサ
	unsigned int getMajor() const { return MAJOR_VER; };//メジャーバージョン
	unsigned int getMinor() const { return MINOR_VER; };//マイナーバージョン
	unsigned int getVer() const { return VER; };//合成バージョン
};
```

【バージョンテンプレートクラス】

```
//--------------------
//バージョンテンプレートクラス
template<class T>
class CVersionDef : public CVersionDefBase<0, 0>{};//規定では0.0
```

【データクラス用のバージョン定義マクロ】

```
//--------------------
//データクラス用のバージョン定義マクロ
#define SERIALIZE_VERSION_DEF(T, MAJOR, MINOR) ¥
	namespace serial ¥
	{ ¥
		template<> ¥
		struct CVersionDef<T> : public CVersionDefBase<MAJOR, MINOR>{}; ¥
	}
```

▼ 型操作クラス

【文字列型／バイナリ型指定用構造体】

```
//--------------------
//汎用データ型指定用構造体
struct str_t{};//文字列型
struct bin_t{};//バイナリ型
```

【型操作基底クラス】

```
//--------------------
//型操作基底クラス
typedef std::size_t(*toMemFuncP)(void* mem, const std::size_t mem_size, const void* value_p,
                                 const std::size_t value_size);//メモリへコピー関数型
typedef std::size_t(*fromMemFuncP)(const void* mem, const std::size_t mem_size, void* value_p,
                                 const std::size_t value_size);//メモリからコピー関数型
typedef std::size_t(*toStrFuncP)(char* str, const std::size_t str_max, const void* value_p,
                                 const std::size_t value_size);//文字列へ変換関数型
typedef std::size_t(*fromStrFuncP)(const char* str, const std::size_t str_size, void* value_p,
                                 const std::size_t value_size_max);//文字列から変換関数型
class CTypeCtrlBase
{
```

```
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		if (mem_size < value_size)
		{
			//書き込み先のサイズの方が小さい場合
			//※書き込み先のサイズ分だけコピーする
			memcpy(mem, value_p, mem_size);
		}
		else if (mem_size > value_size)
		{
			//書き込み先のサイズの方が大きい場合
			//※後方をゼロクリア
			memcpy(mem, value_p, value_size);
			memset(reinterpret_cast<char*>(mem)+value_size, 0, mem_size - value_size);
		}
		else//if (mem_size == value_size)
		{
			//サイズが一致する場合
			//※そのままコピーするだけ
			memcpy(mem, value_p, mem_size);
		}
		return mem_size;
	}
	//※エンディアン調整版
	static std::size_t toMemAndAdjust(void* mem, const std::size_t mem_size, const void* value_p,
	                                  const std::size_t value_size)
	{
	#ifdef IS_BIG_ENDIAN
		if (mem_size < value_size)
		{
			//書き込み先のサイズの方が小さい場合
			//※書き込み先のサイズ分だけ後方をコピーする
			memcpy(mem, reinterpret_cast<const char*>(value_p)+(value_size - mem_size), mem_size);
		}
		else if (mem_size > value_size)
		{
			//書き込み先のサイズの方が大きい場合
			//※後詰めで前方をゼロクリア
			memcpy(reinterpret_cast<char*>(mem)+value_size, value_p, value_size);
			memset(mem, 0, mem_size - value_size);
		}
		else//if (mem_size == value_size)
		{
			//サイズが一致する場合
			//※そのままコピーするだけ
			memcpy(mem, value_p, mem_size);
		}
		return mem_size;
	#else//IS_BIG_ENDIAN
		return toMem(mem, mem_size, value_p, value_size);
	#endif//IS_BIG_ENDIAN
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                           const std::size_t value_size)
	{
		return toMem(value_p, value_size, mem, mem_size);
	}
	//※エンディアン調整版
	static std::size_t fromMemAndAdjust(const void* mem, const std::size_t mem_size, void* value_p,
	                                    const std::size_t value_size)
	{
```

```
		return toMemAndAdjust(value_p, value_size, mem, mem_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		std::size_t used = 0;
		std::size_t remain = str_max;
		char* write_p = str;
		if (remain > 3)
		{
			*(write_p++) = '[';
			++used;
			--remain;
		}
		const unsigned char* read_p = reinterpret_cast<const unsigned char*>(value_p);
		for (unsigned int i = 0; i < value_size && remain > 5 + 2; ++i)
		{
			const unsigned char c = *(read_p++);
			const unsigned char hi = c >> 4;
			const unsigned char lo = c & 0xf;
			if (i != 0)
			{
				*(write_p++) = ',';
				++used;
				--remain;
			}
			*(write_p++) = '0';
			*(write_p++) = 'x';
			*(write_p++) = hi >= 10 ? 'a' + hi - 10 : '0' + hi;
			*(write_p++) = lo >= 10 ? 'a' + lo - 10 : '0' + lo;
			used += 4;
			remain -= 4;
		}
		if (remain > 2)
		{
			*(write_p++) = ']';
			++used;
			--remain;
		}
		if (remain > 1)
			*(write_p) = '¥0';
		return used;
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//デフォルトコンストラクタ
	CTypeCtrlBase() :
		m_toMemFuncP(toMem),
		m_fromMemFuncP(fromMem),
		m_toStrFuncP(toStr),
		m_fromStrFuncP(fromStr)
	{}
	//コピーコンストラクタ
	CTypeCtrlBase(const CTypeCtrlBase& src) :
		m_toMemFuncP(src.m_toMemFuncP),
		m_fromMemFuncP(src.m_fromMemFuncP),
		m_toStrFuncP(src.m_toStrFuncP),
		m_fromStrFuncP(src.m_fromStrFuncP)
	{}
```

```
	//コンストラクタ
	CTypeCtrlBase(toMemFuncP to_mem_func_p, fromMemFuncP from_mem_func_p, toStrFuncP to_func_p,
	                                                                    fromStrFuncP from_func_p) :
		m_toMemFuncP(to_mem_func_p),
		m_fromMemFuncP(from_mem_func_p),
		m_toStrFuncP(to_func_p),
		m_fromStrFuncP(from_func_p)
	{}
	//デストラクタ
	~CTypeCtrlBase()
	{}
public:
	//フィールド
	toMemFuncP m_toMemFuncP;//メモリへコピー関数
	fromMemFuncP m_fromMemFuncP;//メモリからコピー関数
	toStrFuncP m_toStrFuncP;//文字列へ変換関数
	fromStrFuncP m_fromStrFuncP;//文字列から変換関数
};
```

【型操作テンプレートクラス】

```
//--------------------
//型操作テンプレートクラス
template<typename T>
class CTypeCtrl : public CTypeCtrlBase
{
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase()
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：int 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：int 型に特殊化
template<>
class CTypeCtrl<int> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                                                                    const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const int value = *reinterpret_cast<const int*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "%d", value);
	#else//USE_STRCPY_S
		return sprintf(str, "%d", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
```

```
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t data_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：unsigned int 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：unsigned int 型に特殊化
template<>
class CTypeCtrl<unsigned int> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                                                                        const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const unsigned int value = *reinterpret_cast<const unsigned int*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "0x%08x", value);
	#else//USE_STRCPY_S
		return sprintf(str, "0x%08x", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：long 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：long 型に特殊化
```

```
template<>
class CTypeCtrl<long> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
		const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const long value = *reinterpret_cast<const long*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "%ld", value);
	#else//USE_STRCPY_S
		return sprintf(str, "%ld", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
		const std::size_t data_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：unsigned long 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：unsigned long 型に特殊化
template<>
class CTypeCtrl<unsigned long> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
		const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
```

```
		const unsigned long value = *reinterpret_cast<const unsigned long*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "0x%08lx", value);
	#else//USE_STRCPY_S
		return sprintf(str, "0x%08lx", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：long long 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：long long 型に特殊化
template<>
class CTypeCtrl<long long> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                                                                        const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const long long value = *reinterpret_cast<const long long*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "%lld", value);
	#else//USE_STRCPY_S
		return sprintf(str, "%lld", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t data_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
```

```
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：unsigned long long 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：unsigned long long 型に特殊化
template<>
class CTypeCtrl<unsigned long long> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                           const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const unsigned long long value = *reinterpret_cast<const unsigned long long*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "0x%016llx", value);
	#else//USE_STRCPY_S
		return sprintf(str, "0x%016llx", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                           const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：short 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：short 型に特殊化
template<>
class CTypeCtrl<short> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
```

```
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                                                                        const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const short value = *reinterpret_cast<const short*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "%d", value);
	#else//USE_STRCPY_S
		return sprintf(str, "%d", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：unsigned short 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：unsigned short 型に特殊化
template<>
class CTypeCtrl<unsigned short> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                                                                        const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const unsigned short value = *reinterpret_cast<const unsigned short*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "0x%04x", value);
	#else//USE_STRCPY_S
		return sprintf(str, "0x%04x", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                                                                        const std::size_t value_size_max)
	{
```

```
            //未実装
            return 0;
      }
public:
      //コンストラクタ
      CTypeCtrl() :
            CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
      {}
      //デストラクタ
      ~CTypeCtrl()
      {}
};
```

【型操作テンプレートクラス：char 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：char 型に特殊化
template<>
class CTypeCtrl<char> : public CTypeCtrlBase
{
public:
      //メソッド
      //メモリへコピー
      static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
      {
            return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
      }
      //メモリからコピー
      static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
                                                                              const std::size_t value_size)
      {
            return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
      }
      //文字列へ変換
      static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
      {
            const char value = *reinterpret_cast<const char*>(value_p);
      #ifdef USE_STRCPY_S
            return sprintf_s(str, str_max, "%d", value);
      #else//USE_STRCPY_S
            return sprintf(str, "%d", value);
      #endif//USE_STRCPY_S
      }
      //文字列から変換
      static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
                                                                              const std::size_t value_size_max)
      {
            //未実装
            return 0;
      }
public:
      //コンストラクタ
      CTypeCtrl() :
            CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
      {}
      //デストラクタ
      ~CTypeCtrl()
      {}
};
```

【型操作テンプレートクラス：unsigned char 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：unsigned char 型に特殊化
template<>
class CTypeCtrl<unsigned char> : public CTypeCtrlBase
{
```

```
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
		const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		const unsigned char value = *reinterpret_cast<const unsigned char*>(value_p);
	#ifdef USE_STRCPY_S
		return sprintf_s(str, str_max, "0x%02x", value);
	#else//USE_STRCPY_S
		return sprintf(str, "0x%02x", value);
	#endif//USE_STRCPY_S
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
		const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：float 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：float 型に特殊化
template<>
class CTypeCtrl<float> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		//return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
		return CTypeCtrlBase::toMem(mem, mem_size, value_p, value_size);//エンディアン調整なしでコピー
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
		const std::size_t value_size)
	{
		if (mem_size == sizeof(double))
		{
			//double 型と予測
			double value_from;
			CTypeCtrlBase::fromMem(mem, mem_size, &value_from, sizeof(value_from));
			const float value_to = static_cast<float>(value_from);
			CTypeCtrlBase::toMem(value_p, value_size, &value_to, sizeof(value_to));
```

```
            }
            else if (mem_size == sizeof(float))
            {
                  //return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
                  return CTypeCtrlBase::fromMem(mem, mem_size, value_p, value_size);//エンディアン調整なしでコピー
            }
            else
            {
                  //整数型と予測
                  long long value_from;
                  CTypeCtrlBase::fromMem(mem, mem_size, &value_from, sizeof(value_from));
                  const float value_to = static_cast<float>(value_from);
                  CTypeCtrlBase::toMem(value_p, value_size, &value_to, sizeof(value_to));
            }
            return mem_size;
      }
      //文字列へ変換
      static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
      {
            const float value = *reinterpret_cast<const float*>(value_p);
      #ifdef USE_STRCPY_S
            return sprintf_s(str, str_max, "%f", value);
      #else//USE_STRCPY_S
            return sprintf(str, "%f", value);
      #endif//USE_STRCPY_S
      }
      //文字列から変換
      static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
                                                                              const std::size_t value_size_max)
      {
            //未実装
            return 0;
      }
public:
      //コンストラクタ
      CTypeCtrl() :
            CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
      {}
      //デストラクタ
      ~CTypeCtrl()
      {}
};
```

【型操作テンプレートクラス：double 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：double 型に特殊化
template<>
class CTypeCtrl<double> : public CTypeCtrlBase
{
public:
      //メソッド
      //メモリへコピー
      static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
      {
            //return CTypeCtrlBase::toMemAndAdjust(mem, mem_size, value_p, value_size);
            return CTypeCtrlBase::toMem(mem, mem_size, value_p, value_size);//エンディアン調整なしでコピー
      }
      //メモリからコピー
      static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
                                                                              const std::size_t value_size)
      {
            if (mem_size == sizeof(float))
            {
                  //float 型と予測
                  float value_from;
```

```
                CTypeCtrlBase::fromMem(mem, mem_size, &value_from, sizeof(value_from));
                const double value_to = static_cast<float>(value_from);
                CTypeCtrlBase::toMem(value_p, value_size, &value_to, sizeof(value_to));
            }
            else if (mem_size == sizeof(double))
            {
                //return CTypeCtrlBase::fromMemAndAdjust(mem, mem_size, value_p, value_size);
                return CTypeCtrlBase::fromMem(mem, mem_size, value_p, value_size);//エンディアン調整なしでコピー
            }
            else
            {
                //整数型と予測
                long long value_from;
                CTypeCtrlBase::fromMem(mem, mem_size, &value_from, sizeof(value_from));
                const double value_to = static_cast<double>(value_from);
                CTypeCtrlBase::toMem(value_p, value_size, &value_to, sizeof(value_to));
            }
            return mem_size;
        }
        //文字列へ変換
        static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
        {
            const double value = *reinterpret_cast<const double*>(value_p);
        #ifdef USE_STRCPY_S
            return sprintf_s(str, str_max, "%llf", value);
        #else//USE_STRCPY_S
            return sprintf(str, "%llf", value);
        #endif//USE_STRCPY_S
        }
        //文字列から変換
        static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
                                                                                const std::size_t value_size_max)
        {
            //未実装
            return 0;
        }
public:
        //コンストラクタ
        CTypeCtrl() :
            CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
        {}
        //デストラクタ
        ~CTypeCtrl()
        {}
};
```

【型操作テンプレートクラス：bin_t 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：バイナリ型（bin_t 型）に特殊化
template<>
class CTypeCtrl<bin_t> : public CTypeCtrlBase
{
public:
        //メソッド
        //メモリへコピー
        static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
        {
            return CTypeCtrlBase::toMem(mem, mem_size, value_p, value_size);
        }
        //メモリからコピー
        static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
                                                                                const std::size_t value_size)
        {
            return CTypeCtrlBase::fromMem(mem, mem_size, value_p, value_size);
        }
```

```
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toStr(str, str_max, value_p, value_size);
	}
	//文字列から変換
	static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
	                           const std::size_t value_size_max)
	{
		//未実装
		return 0;
	}
public:
	//コンストラクタ
	CTypeCtrl() :
		CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
	{}
	//デストラクタ
	~CTypeCtrl()
	{}
};
```

【型操作テンプレートクラス：str_t 型に特殊化】

```
//--------------------
//型操作テンプレートクラス：文字列型（str_t 型）に特殊化
template<>
class CTypeCtrl<str_t> : public CTypeCtrlBase
{
public:
	//メソッド
	//メモリへコピー
	static std::size_t toMem(void* mem, const std::size_t mem_size, const void* value_p, const std::size_t value_size)
	{
		return CTypeCtrlBase::toMem(mem, mem_size, value_p, value_size);
	}
	//メモリからコピー
	static std::size_t fromMem(const void* mem, const std::size_t mem_size, void* value_p,
	                           const std::size_t value_size)
	{
		return CTypeCtrlBase::fromMem(mem, mem_size, value_p, value_size);
	}
	//文字列へ変換
	static std::size_t toStr(char* str, const std::size_t str_max, const void* value_p, const std::size_t value_size)
	{
		std::size_t used = 0;
		std::size_t remain = str_max;
		char* write_p = str;
		if (remain > 1 + 1)
		{
			*(write_p++) = '¥"';
			++used;
			--remain;
		}
		const unsigned char* read_p = reinterpret_cast<const unsigned char*>(value_p);
		for (unsigned int i = 0; i < value_size && remain > 1 + 1; ++i)
		{
			const unsigned char c = *(read_p++);
			if (c == '¥0')
				break;
			*(write_p++) = c;
			++used;
			++remain;
		}
		if (remain > 1 + 1)
		{
```

```
                *(write_p++) = '¥"';
                ++used;
                --remain;
            }
            if (remain > 1)
                *(write_p) = '¥0';
            return used;
        }
        //文字列から変換
        static std::size_t fromStr(const char* str, const std::size_t str_size, void* value_p,
                                                                    const std::size_t value_size_max)
        {
            //未実装
            return 0;
        }
public:
        //コンストラクタ
        CTypeCtrl() :
            CTypeCtrlBase(toMem, fromMem, toStr, fromStr)
        {}
        //デストラクタ
        ~CTypeCtrl()
        {}
};
```

▼ シリアライズ関数オブジェクト用テンプレートクラス

【シリアライズ／デシリアライズ共通処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//シリアライズ／デシリアライズ共通処理用関数オブジェクトテンプレートクラス
//※シリアライズとデシリアライズ兼用共通処理
//※特殊化によりユーザー処理を実装
//※標準では何もしない
template<class Arc, class T>
struct serialize {
    typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
    void operator()(Arc& arc, const T& obj, const CVersion& ver, const CVersion& now_ver)
    {}
};
```

【セーブ処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//セーブ処理用関数オブジェクトテンプレートクラス
//※シリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※標準では何もしない
template<class Arc, class T>
struct save {
    typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
    void operator()(Arc& arc, const T& obj, const CVersion& ver)
    {}
};
```

【ロード処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//ロード処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※標準では何もしない
template<class Arc, class T>
struct load {
    typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
```

```
	void operator()(Arc& arc, const T& obj, const CVersion& ver, const CVersion& now_ver)
	{}
};
```

【ロード前処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//ロード前処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※obj のロードを開始する前に実行される
//※標準では何もしない
template<class Arc, class T>
struct beforeLoad {
	typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
	void operator()(Arc& arc, T& obj, const CVersion& ver, const CVersion& now_ver)
	{}
};
```

【ロード後処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//ロード後処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※obj のロードが一通り終わったあと実行される
//※noticeUnrecognizedItem, noticeUnloadedItem の後に実行される
//※標準では何もしない
template<class Arc, class T>
struct afterLoad {
	typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
	void operator()(Arc& arc, T& obj, const CVersion& ver, const CVersion& now_ver)
	{}
};
```

【セーブデータにしかなかったデータ項目通知用関数オブジェクトテンプレートクラス】

```
//--------------------
//セーブデータにはあったが、保存先の指定がなく、ロードできなかった項目の通知
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※見つかったら時点で知されるので、obj が不完全な状態である点に注意
//※委譲データ項目 delegate_item にデータ項目をセットして返すとリトライしてそこに読み込む
//※標準では何もしない
class CItemBase;//データ項目情報クラス宣言
template<class Arc, class T>
struct noticeUnrecognizedItem {
	typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
	void operator()(Arc& arc, T& obj, CVersion& ver, const CVersion& now_ver, const CItemBase& unrecognized_item)
	{}
};
```

【セーブデータになかったデータ項目通知用関数オブジェクトテンプレートクラス】

```
//--------------------
//保存先の指定があるが、セーブデータになくロードできなかった項目の通知
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※obj のロードが一通り終わったあと、まとめて通知する
//※noticeUnrecognizedItem の後、afterLoad より先に実行される
//※標準では何もしない
class CItemBase;//データ項目情報クラス宣言
template<class Arc, class T>
struct noticeUnloadedItem {
	typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
	void operator()(Arc& arc, T& obj, CVersion& ver, const CVersion& now_ver, const CItemBase& unloaded_item)
	{}
};
```

【コレクター（収集）用関数オブジェクトテンプレートクラス】

```
//--------------------
//データ収集処理用関数オブジェクトテンプレートクラス
//※シリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※処理の中で使えるのは operator<<() のみ
//  operator&() は禁止
//※標準では何もしない
template<class Arc, class T>
struct collector {
      typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
      void operator()(Arc& arc, const T& obj, const CVersion& ver)
      {}
};
```

【ディストリビュータ（分配）用関数オブジェクトテンプレートクラス】

```
//--------------------
//データ分配処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
//※処理の中で使えるのは operator>>() のみ
//  operator&() は禁止
//※収集処理で登録されたオブジェクトがセーブデータから見つかるごとに
//  何度も呼び出される
//※見つかったオブジェクトの情報（データ項目情報 = target_item）が渡される
//※基本オブジェクト（obj）は、配列だった場合、その先頭の要素が渡される
//※セーブデータ上の配列要素数と、ロードできた配列要素数（メモリ上の配列要素数）が渡される
//※標準では何もしない
class CItemBase;//データ項目情報クラス宣言
template<class Arc, class T>
struct distributor {
      typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
      void operator()(Arc& arc, T& obj, const std::size_t array_num_on_save_data, const std::size_t array_num_loaded,
                                    const CVersion& ver, const CVersion& now_ver, const CItemBase& target_item)
      {}
};
```

【ディストリビュート前処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//データ分配前処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
template<class Arc, class T>
struct beforeDistribute {
      typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
      void operator()(Arc& arc, T& obj, const std::size_t array_num_on_save_data, const std::size_t array_num_loaded,
                                                                  const CVersion& ver, const CVersion& now_ver)
      {}
};
```

【ディストリビュート後処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//データ分配後処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※特殊化によりユーザー処理を実装
template<class Arc, class T>
struct afterDistribute {
      typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
      void operator()(Arc& arc, T& obj, const std::size_t array_num_on_save_data, const std::size_t array_num_loaded,
                                                                  const CVersion& ver, const CVersion& now_ver)
      {}
};
```

【シリアライズ時に致命的エラー発生時処理用関数オブジェクトテンプレートクラス】

```
//--------------------
```

```
//シリアライズ時に致命的エラー発生時処理用関数オブジェクトテンプレートクラス
//※シリアライズ専用処理
//※エラーが発生したタイミングではなく、最後に呼び出される
//※特殊化によりユーザー処理を実装
template<class Arc, class T>
struct fatalSerializeErrorOccurred {
	typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
	void operator()(Arc& arc, const T& obj, const CVersion& ver)
	{}
};
```

【デシリアライズ時に致命的エラー発生時処理用関数オブジェクトテンプレートクラス】

```
//--------------------
//デシリアライズ時に致命的エラー発生時処理用関数オブジェクトテンプレートクラス
//※デシリアライズ専用処理
//※エラーが発生したタイミングではなく、最後に呼び出される
//※特殊化によりユーザー処理を実装
template<class Arc, class T>
struct fatalDeserializeErrorOccurred {
	typedef int IS_UNDEFINED;//SFINAE 用:関数オブジェクトの未定義チェック用の型定義
	void operator()(Arc& arc, const T& obj, const CVersion& ver, const CVersion& now_ver)
	{}
};
```

【ユーザー定義処理用特殊化テンプレートクラス定義済みチェック関数】

```
//--------------------
//ユーザー定義処理用特殊化テンプレートクラス定義済みチェック関数
//※SFINAE により、IS_UNDEFINED が定義されている型のオーバーロード関数が選ばれたら未定義とみなす
template<class F>
bool isDefinedFunctor(const typename F::IS_UNDEFINED)
{
	return false;//未定義
}
template<class F>
bool isDefinedFunctor(...)
{
	return true;//定義済み
}
```

【オブジェクト型判定用関数】

```
//--------------------
//オブジェクト型か？
//※いずれかの関数オブジェクトが登録されていればオブジェクト型とみなす
//※オブジェクト型はシリアライズの際にデータブロックとして扱う
class CArchiveDummy{};//アーカイブダミークラス
template<class T>
bool hasAnyFunctor()
{
	//ここでチェック：収集処理と分配処理は必ずワンセットで定義する必要あり
	assert((isDefinedFunctor<collector<CArchiveDummy, T> >(0)) ==
	                                        (isDefinedFunctor<distributor<CArchiveDummy, T> >(0)));
	//関数オブジェクトのどれか一つでも定義されているかチェック
	return isDefinedFunctor<serialize<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<save<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<load<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<beforeLoad<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<afterLoad<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<noticeUnrecognizedItem<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<noticeUnloadedItem<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<collector<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<distributor<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<beforeDistribute<CArchiveDummy, T> >(0) ||
		isDefinedFunctor<afterDistribute<CArchiveDummy, T> >(0);
		//isDefinedFunctor<fatalSerializeErrorOccurred<CArchiveDummy, T> >(0);//これは数えない
		//isDefinedFunctor<fatalDeserializeErrorOccurred<CArchiveDummy, T> >(0);//これは数えない
```

```
}
```

【データクラスのフレンド化マクロ】

```
//--------------------
//データクラスのフレンド化マクロ
#define FRIEND_SERIALIZE() ¥
	template<class Arc, class T> ¥
	friend struct serial::serialize; ¥
	template<class Arc, class T> ¥
	friend struct serial::save; ¥
	template<class Arc, class T> ¥
	friend struct serial::load; ¥
	template<class Arc, class T> ¥
	friend struct serial::beforeLoad; ¥
	template<class Arc, class T> ¥
	friend struct serial::afterLoad; ¥
	template<class Arc, class T> ¥
	friend struct serial::noticeUnrecognizedItem; ¥
	template<class Arc, class T> ¥
	friend struct serial::noticeUnloadedItem; ¥
	template<class Arc, class T> ¥
	friend struct serial::collector; ¥
	template<class Arc, class T> ¥
	friend struct serial::distributor; ¥
	template<class Arc, class T> ¥
	friend struct serial::beforeDistribute; ¥
	template<class Arc, class T> ¥
	friend struct serial::afterDistribute; ¥
	template<class Arc, class T> ¥
	friend struct serial::fatalSerializeErrorOccurred; ¥
	template<class Arc, class T> ¥
	friend struct fatalDeserializeErrorOccurred;
```

▼ データ項目管理クラス

【ポインタ型チェック用テンプレートクラス】※テンプレートの部分特殊化を活用してポインタを消去した型を得る

```
//--------------------
//ポインタ型チェック用クラス
//※テンプレートの部分特殊化を利用
template<class T>
struct isPtr
{
	static const bool IS_PTR = false;//ポインタ型か？ = 非ポインタ型
	typedef T TYPE;//通常型（非ポインタ型）変換用の型
	typedef T* PTR_TYPE;//ポインタ型変換用の型
	static const T& TO_VALUE(const T& var){ return var; }//値に変換
	static const T* TO_PTR(const T& var){ return reinterpret_cast<const T*>(&var); }//ポインタに変換
	static bool isNull(const T& var){ return false; }//ヌルか？
};
template<class T>
struct isPtr<T*>
{
	static const bool IS_PTR = true;//ポインタ型か？ = ポインタ型
	typedef T TYPE;//通常型（非ポインタ型）変換用の型
	typedef T* PTR_TYPE;//ポインタ型変換用の型
	static const T& TO_VALUE(const T* var){ return *var; }//値に変換
	static const T* TO_PTR(const T* var){ return var; }//ポインタに変換
	static bool isNull(const T* var){ return var == nullptr; }//ヌルか？
};
```

【データ項目属性】

```
//--------------------
```

```
//データ項目属性
enum itemAttrEnum : unsigned char
{
	IS_OBJECT = 0x01,//オブジェクト型
	IS_ARRAY = 0x02,//配列型
	IS_PTR = 0x04,//ポインタ型
	IS_NULL = 0x08,//ヌル
	IS_VLEN = 0x10,//可変長か？
	HAS_VER = 0x20,//バージョン情報あり
};
class CItemAttr
{
public:
	//型
	typedef unsigned char value_t;//属性型
public:
	//アクセッサ
	bool isObj() const { return (m_value & IS_OBJECT) ? true : false; }//オブジェクト型か？
	bool isArr() const { return (m_value & IS_ARRAY) ? true : false; }//配列型か？
	bool isPtr() const { return (m_value & IS_PTR) ? true : false; }//ポインタ型か？
	bool isNul() const { return (m_value & IS_NULL) ? true : false; }//ヌルポインタか？（ポインタ型の時だけ扱われる）
	bool isVLen() const { return (m_value & IS_VLEN) ? true : false; }//可変長か？（主に文字列ポインタの時に扱われる）
	bool hasVer() const { return (m_value & HAS_VER) ? true : false; }//バージョン情報があるか？
	void setHasVer() const { *const_cast<value_t*>(&m_value) = m_value | HAS_VER; }//バージョン情報ありにする
	void resetHasVer() const { *const_cast<value_t*>(&m_value) = m_value & ~HAS_VERS; }//バージョン情報なしにする
public:
	//オペレータ
	bool operator==(const CItemAttr& rhs) const { return m_value == rhs.m_value; }
	bool operator!=(const CItemAttr& rhs) const { return m_value != rhs.m_value; }
	//コピーオペレータ
	CItemAttr& operator=(const CItemAttr& src)
	{
		*const_cast<value_t*>(&m_value) = src.m_value;
		return *this;
	}
public:
	//メソッド
	//クリア
	void clear()
	{
		*const_cast<value_t*>(&m_value) = 0;
	}
public:
	//コンストラクタ
	CItemAttr(const value_t info) :
		m_value(info)
	{}
	CItemAttr(const bool is_object, const bool is_array, const bool is_ptr, const bool is_null, const bool is_vlen) :
		m_value(
			(is_object ? IS_OBJECT : 0) |
			(is_array ? IS_ARRAY : 0) |
			(is_ptr ? IS_PTR : 0) |
			(is_ptr && is_null ? IS_NULL : 0) |
			(is_vlen ? IS_VLEN : 0)
		)
	{}
	//デストラクタ
	~CItemAttr()
	{}
public://直接アクセス許可
	//フィールド
	const value_t m_value;//属性
};
```

【データ項目情報基底クラス】

```
//--------------------
//データ項目情報基底クラス
class CItemBase
{
public:
	//アクセッサ
	bool isObj() const { return m_attr.isObj(); }//オブジェクト型か？
	bool isArr() const { return m_attr.isArr(); }//配列型か？
	bool isPtr() const { return m_attr.isPtr(); }//ポインタ型か？
	bool isNul() const { return m_attr.isNul(); }//ヌルか？
	bool isVLen() const { return m_attr.isVLen(); }//可変長か？
	//【参考用】（セーブデータ上の）配列要素数を取得
	//※アーカイブ形式によっては正しくない可能性がある
	std::size_t getElemNum() const
	{
		return	m_arrNum == 0 ?
				1 :
				m_arrNum;
	}
	//現在の配列要素数を取得
	std::size_t getNowElemNum() const
	{
		return	m_hasNowInfo ?
				m_nowArrNum == 0 ?
					1 :
					m_nowArrNum :
				0;
	}
	//【参考用】最小の配列要素数を取得
	//※アーカイブ形式によっては正しくない可能性がある
	std::size_t getMinimumElemNum() const
	{
		return	m_hasNowInfo ?
				m_arrNum < m_nowArrNum ?
					m_arrNum == 0 ?
						1 :
						m_arrNum :
					m_nowArrNum == 0 ?
						1 :
						m_nowArrNum :
				m_arrNum == 0 ?
					1 :
					m_arrNum;
	}
	bool nowIsObj() const { return m_hasNowInfo && m_nowAttr.isObj(); }//現在のデータはオブジェクト型か？
	bool nowIsArr() const { return m_hasNowInfo && m_nowAttr.isArr(); }//現在のデータは配列型か？
	bool nowIsPtr() const { return m_hasNowInfo && m_nowAttr.isPtr(); }//現在のデータはポインタ型か？
	bool nowIsNul() const { return m_hasNowInfo && m_nowAttr.isNul(); }//現在のデータはヌルか？
	bool nowIsVLen() const { return m_hasNowInfo && m_nowAttr.isVLen(); }//現在のデータは可変長か？
	bool nowAndSaveDataIsSameAttr() const { return m_hasNowInfo && m_nowAttr == m_attr; }
	                                                    //現在のデータとセーブデータの属性が一致するか？
	bool nowAndSaveDataIsDifferentAttr() const { return m_hasNowInfo && m_nowAttr != m_attr; }
	                                                    //現在のデータとセーブデータの属性が一致しないか？
	bool nowAndSaveDataIsObj() const { return m_hasNowInfo && m_nowAttr.isObj() && m_attr.isObj(); }
	                                                    //現在のデータもセーブデータもオブジェクト型か？
	bool nowAndSaveDataIsArr() const { return m_hasNowInfo && m_nowAttr.isArr() && m_attr.isArr(); }
	                                                    //現在のデータもセーブデータも配列型か？
	bool nowAndSaveDataIsPtr() const { return m_hasNowInfo && m_nowAttr.isPtr() && m_attr.isPtr(); }
	                                                    //現在のデータもセーブデータもポインタ型か？
	bool nowAndSaveDataIsNul() const { return m_hasNowInfo && m_nowAttr.isNul() && m_attr.isNul(); }
	                                                    //現在のデータもセーブデータもはヌルか？
	bool nowIsObjButSaveDataIsNot() const { return m_hasNowInfo && m_nowAttr.isObj() && !m_attr.isObj(); }
	                                                //現在のデータはオブジェクト型だがセーブデータはそうではないか？
	bool nowIsArrButSaveDataIsNot() const { return m_hasNowInfo && m_nowAttr.isArr() && !m_attr.isArr(); }
```

```
                                                //現在のデータは配列型だがセーブデータはそうではないか？
    bool nowIsPtrButSaveDataIsNot() const { return m_hasNowInfo && m_nowAttr.isPtr() && !m_attr.isPtr(); }
                                                //現在のデータはポインタ型だがセーブデータはそうではないか？
    bool nowIsNulButSaveDataIsNot() const { return m_hasNowInfo && m_nowAttr.isNul() && !m_attr.isNul(); }
                                                //現在のデータはヌルだがセーブデータはそうではないか？
    bool nowIsNotObjButSaveDataIs() const { return m_hasNowInfo && !m_nowAttr.isObj() && m_attr.isObj(); }
                                                //現在のデータはオブジェクト型ではないがセーブデータはそうか？
    bool nowIsNotArrButSaveDataIs() const { return m_hasNowInfo && !m_nowAttr.isArr() && m_attr.isArr(); }
                                                //現在のデータは配列型ではないがセーブデータはそうか？
    bool nowIsNotPtrButSaveDataIs() const { return m_hasNowInfo && !m_nowAttr.isPtr() && m_attr.isPtr(); }
                                                //現在のデータはポインタ型ではないがセーブデータはそうか？
    bool nowIsNotNulButSaveDataIs() const { return m_hasNowInfo && !m_nowAttr.isNul() && m_attr.isNul(); }
                                                //現在のデータはヌルではないがセーブデータはそうか？
    bool nowSizeIsSame() const { return !isObj() && !isVLen() && m_hasNowInfo && m_nowItemSize == m_itemSize; }
                                                //現在のサイズの方とセーブデータのサイズが同じか？
    bool nowSizeIsSamall() const { return !isObj() && !isVLen() && m_hasNowInfo && m_nowItemSize < m_itemSize; }
                                                //現在のサイズの方がセーブデータのサイズより小さいか？
    bool nowSizeIsLarge() const { return !isObj() && !isVLen() && m_hasNowInfo && m_nowItemSize > m_itemSize; }
                                                //現在のサイズの方がセーブデータのサイズより大きいか？
    bool nowArrIsSame() const { return m_hasNowInfo && m_nowArrNum == m_arrNum; }
                                                //現在の配列要素数とセーブデータの配列要素数が同じか？
    bool nowArrIsSmall() const { return m_hasNowInfo && m_nowArrNum < m_arrNum; }
                                                //現在の配列要素数の方がセーブデータの配列要素数より小さいか？
    bool nowArrIsLarge() const { return m_hasNowInfo && m_nowArrNum > m_arrNum; }
                                                //現在の配列要素数の方がセーブデータの配列要素数より大きいか？
    bool hasNowInfo() const { return m_hasNowInfo; }//現在の情報コピー済み取得
    bool isOnlyOnSaveData() const{ return m_isOnlyOnSaveData; }//セーブデータ上にのみ存在するデータか？
    void setIsOnlyOnSaveData() const { m_isOnlyOnSaveData = true; m_isOnlyOnMem = false; }
                                                        //セーブデータ上にのみ存在するデータか？を更新
    void setIsOnlyOnSaveData(const bool enabled) const { if (enabled) setIsOnlyOnSaveData(); }
                                                        //セーブデータ上にのみ存在するデータか？を更新
    void resetIsOnlyOnSaveData() const { m_isOnlyOnSaveData = false; }
                                                         //セーブデータ上にのみ存在するデータか？をリセット
    bool isOnlyOnMem() const { return m_isOnlyOnMem; }//セーブデータ上にないデータか？
    void setIsOnlyOnMem() const { m_isOnlyOnMem = true; m_isOnlyOnSaveData = false; }
                                                    //セーブデータ上にないデータか？を更新
    void setIsOnlyOnMem(const bool enabled) const { if (enabled) setIsOnlyOnMem(); }
                                                   //セーブデータ上にないデータか？を更新
    void resetIsOnlyOnMem() const { m_isOnlyOnMem = false; }//セーブデータ上にないデータか？をリセット
    bool isAlready() const { return m_isAlready; }//処理済みか？
    void setIsAlready() const { m_isAlready = true; }//処理済みにする
    void resetIsAlready() const { m_isAlready = false; }//処理済みを解除する
public:
    //オペレータ
    bool operator==(const CItemBase& rhs) const { return m_nameCrc == rhs.m_nameCrc; }//データ項目名 CRC で一致判定
    bool operator!=(const CItemBase& rhs) const { return m_nameCrc != rhs.m_nameCrc; }//データ項目名 CRC で不一致判定
    bool operator==(const crc32_t name_crc) const { return m_nameCrc == name_crc; }//データ項目名 CRC で一致判定
    bool operator!=(const crc32_t name_crc) const { return m_nameCrc != name_crc; }//データ項目名 CRC で不一致判定
    bool operator==(const char* name) const { return m_nameCrc == calcCRC32(name); }//データ項目名 CRC で一致判定
    bool operator!=(const char* name) const { return m_nameCrc != calcCRC32(name); }//データ項目名 CRC で不一致判定
public:
    //キャストオペレータ
    operator crc32_t() const { return m_nameCrc; }
    operator const char*() const { return m_name; }
    operator const std::type_info& () const { return *m_itemType; }
public:
    //メソッド
    template<typename T>//値取得
    T& get(){ return *static_cast<T*>(const_cast<void*>(m_itemP)); }
    template<typename T>//値取得（配列要素）
    T& get(const int index){ return static_cast<T*>(const_cast<void*>(m_itemP))[index]; }
    template<typename T>//const で値取得
    const T& get() const { return *static_cast<const T*>(m_itemP); }
    template<typename T>//const で値取得（配列要素）
    const T& get(const int index) const { return static_cast<const T*>(m_itemP)[index]; }
```

```
	template<typename T>//const で値取得
	const T& getConst() const { return *static_cast<const T*>(m_itemP); }
	template<typename T>//const で値取得 (配列要素)
	const T& getConst(const int index) const { return static_cast<const T*>(m_itemP)[index]; }
	//読み込み情報をクリア
	void clearForLoad()
	{
		//m_name = nullptr;//データ項目名
		*const_cast<crc32_t*>(&m_nameCrc) = 0;//データ項目名 CRC
		//m_itemP;//データの参照ポインタ
		//m_itemType;//データの型情報
		*const_cast<std::size_t*>(&m_itemSize) = 0;//データサイズ
		*const_cast<std::size_t*>(&m_arrNum) = 0;//データの配列要素数
		const_cast<CItemAttr*>(&m_attr)->clear();//属性
		m_nowItemSize = 0;//現在のデータサイズ
		m_nowArrNum = 0;//現在のデータの配列要素数
		m_nowAttr.clear();//現在の属性
		m_hasNowInfo = false;//現在の情報コピー済み
		m_isOnlyOnSaveData = false;//セーブデータ上にのみ存在するデータ
		m_isOnlyOnMem = false;//セーブデータ上にないデータ
		m_isAlready = false;//処理済み
	}
	//現在の情報をコピー
	void copyFromOnMem(const CItemBase& src)
	{
		//assert(m_nameCrc == src.m_nameCrc);//CRC チェック⇒しない
		m_name = src.m_name;//データ項目名
		*const_cast<crc32_t*>(&m_nameCrc) = src.m_nameCrc;//データ項目名 CRC ※委譲対応のため CRC も更新
		m_itemP = src.m_itemP;//データの参照ポインタ
		m_itemType = src.m_itemType;//データの型情報
		m_nowItemSize = src.m_itemSize;//現在のデータサイズ
		m_nowArrNum = src.m_arrNum;//現在のデータの配列要素数
		m_nowAttr = src.m_attr;//現在の属性
		m_nowTypeCtrl = src.m_typeCtrl;//型操作
		m_hasNowInfo = true;//現在の情報コピー済み
		m_isOnlyOnSaveData = false;//セーブデータ上にのみ存在するデータ
		m_isOnlyOnMem = false;//セーブデータ上にないデータ
		src.resetIsOnlyOnMem();//コピー元の「セーブデータ上にないデータ」をリセット
	}
	//強制的に全情報をコピー
	void copyForce(const CItemBase& src)
	{
		memcpy(this, &src, sizeof(*this));
	}
public:
	//コピーコンストラクタ
	CItemBase(const CItemBase& src) :
		m_name(src.m_name),
		m_nameCrc(src.m_nameCrc),
		m_itemP(src.m_itemP),
		m_itemType(src.m_itemType),
		m_itemSize(src.m_itemSize),
		m_arrNum(src.m_arrNum),
		m_attr(src.m_attr),
		m_typeCtrl(src.m_typeCtrl),
		m_nowItemSize(src.m_nowItemSize),
		m_nowArrNum(src.m_nowArrNum),
		m_nowAttr(src.m_nowAttr),
		m_nowTypeCtrl(src.m_nowTypeCtrl),
		m_hasNowInfo(src.m_hasNowInfo),
		m_isOnlyOnSaveData(src.m_isOnlyOnSaveData),
		m_isOnlyOnMem(src.m_isOnlyOnMem),
		m_isAlready(src.m_isAlready)
	{}
	//デフォルトコンストラクタ
```

```
	CItemBase() :
		m_name(nullptr),
		m_nameCrc(0),
		m_itemP(nullptr),
		m_itemType(&typeid(void)),
		m_itemSize(0),
		m_arrNum(0),
		m_attr(false, false, false, false, false),
		m_typeCtrl(CTypeCtrlBase()),
		m_nowItemSize(0),
		m_nowArrNum(0),
		m_nowAttr(false, false, false, false, false),
		m_nowTypeCtrl(CTypeCtrlBase()),
		m_hasNowInfo(false),
		m_isOnlyOnSaveData(false),
		m_isOnlyOnMem(false),
		m_isAlready(false)
	{}
	//コンストラクタ
	CItemBase(const char* name, const void* item_p, const std::type_info& item_type, const std::size_t item_size,
	const std::size_t arr_num, const bool is_object, const bool is_ptr, const bool is_vlen, CTypeCtrlBase type_ctrl) :
		m_name(name),
		m_nameCrc(calcCRC32(name)),
		m_itemP(item_p),
		m_itemType(&item_type),
		m_itemSize(item_size),
		m_arrNum(arr_num),
		m_attr(is_object, arr_num > 0, is_ptr, item_p == nullptr, is_vlen),
		m_typeCtrl(type_ctrl),
		m_nowItemSize(0),
		m_nowArrNum(0),
		m_nowAttr(false, false, false, false, false),
		m_nowTypeCtrl(CTypeCtrlBase()),
		m_hasNowInfo(false),
		m_isOnlyOnSaveData(false),
		m_isOnlyOnMem(false),
		m_isAlready(false)
	{}
	//コンストラクタ
	CItemBase(const CItemBase& src, const void* item_p, const std::type_info& item_type, const std::size_t item_size,
	const std::size_t arr_num, const bool is_object, const bool is_ptr, const bool is_vlen, CTypeCtrlBase type_ctrl) :
		m_name(src.m_name),
		m_nameCrc(src.m_nameCrc),
		m_itemP(item_p),
		m_itemType(&item_type),
		m_itemSize(item_size),
		m_arrNum(arr_num),
		m_attr(is_object, arr_num > 0, is_ptr, item_p == nullptr, is_vlen),
		m_typeCtrl(type_ctrl),
		m_nowItemSize(0),
		m_nowArrNum(0),
		m_nowAttr(false, false, false, false, false),
		m_nowTypeCtrl(CTypeCtrlBase()),
		m_hasNowInfo(false),
		m_isOnlyOnSaveData(false),
		m_isOnlyOnMem(false),
		m_isAlready(false)
	{}
	//デストラクタ
	~CItemBase()
	{}
public://フィールドを公開して直接操作
	//フィールド
	const char* m_name;//データ項目名
	const crc32_t m_nameCrc;//データ項目名 CRC
```

```
	const void* m_itemP;//データの参照ポインタ
	const std::type_info* m_itemType;//データの型情報
	const std::size_t m_itemSize;//データサイズ
	const std::size_t m_arrNum;//データの配列要素数
	const CItemAttr m_attr;//属性
	CTypeCtrlBase m_typeCtrl;//型操作
	std::size_t m_nowItemSize;//データサイズ　※現在のサイズ（デシリアライズ処理用）
	std::size_t m_nowArrNum;//データの配列要素数　※現在のサイズ（デシリアライズ処理用）
	CItemAttr m_nowAttr;//属性　※現在の状態（デシリアライズ処理用）
	CTypeCtrlBase m_nowTypeCtrl;//型操作　※現在の状態（デシリアライズ処理用）
	bool m_hasNowInfo;//現在の情報コピー済み
	mutable bool m_isOnlyOnSaveData;//セーブデータ上にのみ存在するデータ
	mutable bool m_isOnlyOnMem;//セーブデータ上にないデータ
	mutable bool m_isAlready;//処理済み
};
```

【データ項目情報テンプレートクラス】

```
//--------------------
//データ項目情報テンプレートクラス
template<typename T>
class CItem : public CItemBase
{
public:
	//コンストラクタ
	CItem(const char* name, const T* item_p, const std::size_t arr_num, const bool is_ptr, const bool is_vlen) :
		CItemBase(name, item_p, typeid(T), sizeof(T), arr_num, hasAnyFunctor<T>(), is_ptr, is_vlen, CTypeCtrl<T>())
	{}
	CItem(const char* name, const std::size_t size, const T* item_p, const bool is_ptr, const bool is_vlen) :
		CItemBase(name, item_p, typeid(T), size, 0, hasAnyFunctor<T>(), is_ptr, is_vlen, CTypeCtrl<T>())
	{}
	CItem(const char* name, const std::size_t size) :
		CItemBase(name, nullptr, typeid(T), size, 0, hasAnyFunctor<T>(), false, false, CTypeCtrl<T>())
	{}
	CItem(const char* name) :
		CItemBase(name, nullptr, typeid(T), 0, 0, hasAnyFunctor<T>(), false, false, CTypeCtrl<T>())
	{}
	CItem(const CItemBase& src) :
		CItemBase(src)
	{}
	//デストラクタ
	~CItem()
	{}
};
```

【データ項目情報作成テンプレート関数】

```
//--------------------
//データ項目情報作成テンプレート関数
template<class T>
CItem<typename isPtr<T>::TYPE> pair(const char* name, const T& item)
{
	CItem<typename isPtr<T>::TYPE> item_obj(name, isPtr<T>::TO_PTR(item), 0, isPtr<T>::IS_PTR, false);
	return item_obj;
}
```

【データ項目情報作成テンプレート関数：配列自動判定用】

```
//--------------------
//データ項目情報作成テンプレート関数（配列自動判定用）
template<class T, std::size_t N>
CItem<T> pair(const char* name, const T(&item)[N])
{
	CItem<T> item_obj(name, item, N, false, false);
	return item_obj;
}
```

【データ項目情報作成テンプレート関数：要素数指定型】※ポインタを配列扱いにしたい場合に使用

```
//--------------------
//データ項目情報作成テンプレート関数（ポインタを配列扱いにしたい場合に使用）
template<class T>
CItem<typename isPtr<T>::TYPE> pairArray(const char* name, const T& item, const std::size_t N)
{
	CItem<typename isPtr<T>::TYPE> item_obj(name, isPtr<T>::TO_PTR(item), N, isPtr<T>::IS_PTR, false);
	return item_obj;
}
```

【データ項目情報作成テンプレート関数：バイナリデータ型】

```
//--------------------
//データ項目情報作成テンプレート関数（バイナリ用）
//※operator&()専用
//※operator<<() / operator>>() には使用禁止
template<class T>
CItem<bin_t> pairBin(const char* name, const T& item)
{
	const std::size_t item_size = sizeof(T);
	const bin_t* item_p = reinterpret_cast<const bin_t*>(&item);
	CItem<bin_t> item_obj(name, item_size, item_p, false, false);
	return item_obj;
}
```

【データ項目情報作成テンプレート関数：文字列型】

```
//--------------------
//データ項目情報作成テンプレート関数（文字列用）
//※operator&()専用
//※operator<<() / operator>>() には使用禁止
template<std::size_t N>
CItem<str_t> pairStr(const char* name, const char (&item)[N])
{
	const std::size_t item_size = sizeof(char)* N;
	const str_t* item_p = reinterpret_cast<const str_t*>(&item);
	CItem<str_t> item_obj(name, item_size, item_p, false, true);
	return item_obj;
}
//※ポインタ用
CItem<str_t> pairStr(const char* name, char* item)
{
	const std::size_t item_size = 0;
			//アーカイブ読み込み用のサイズ（アーカイブ書き込み時は実際の文字列長+1 に更新する／ヌルなら 0）
	const str_t* item_p = reinterpret_cast<const str_t*>(item);
	CItem<str_t> item_obj(name, item_size, item_p, true, true);
	return item_obj;
}
```

【データ項目情報作成テンプレート関数：初回用】※データオブジェクト（インスタンス）不要

```
//--------------------
//データ項目情報作成テンプレート関数（初回用）
template<class T>
CItem<T> pair(const char* name, const std::size_t size)
{
	CItem<T> item_obj(name, size);
	return item_obj;
}
template<class T>
CItem<T> pair(const char* name)
{
	CItem<T> item_obj(name, 0);
	return item_obj;
}
```

▼ 処理結果クラス

【処理結果クラス】

```
//--------------------
//処理結果クラス
class CResult
{
public:
	//アクセッサ
	bool hasFatalError() const { return m_hasFatalError; }//致命的なエラーあり
	void setHasFatalError(){ m_hasFatalError = true; }//致命的なエラーあり
	void setHasFatalError(const bool enabled){ if (enabled) setHasFatalError(); }//致命的なエラーあり
	int getNumSmallerSizeItem() const { return m_numSmallerSizeItem; }//サイズが縮小されたデータ項目の数を取得
	int getNumLargerSizeItem() const { return m_numLargerSizeItem; }//サイズが拡大されたデータ項目の数を取得
	int getNumSmallerArrItem() const { return m_numSmallerArrItem; }//配列要素数が縮小されたデータ項目の数を取得
	int getNumLargerArrItem() const { return m_numLargerArrItem; }//配列要素数が拡大されたデータ項目の数を取得
	int getNumIsOnlyOnSaveData() const { return m_numIsOnlyOnSaveData; }
                                                    //セーブデータ上にのみ存在するデータ項目の数を取得
	int getNumIsOnlyOnMem() const { return m_numIsOnlyOnMem; }//セーブデータ上にないデータ項目の数を取得
	int getNumIsObjOnSaveDataOnly() const { return m_numIsObjOnSaveDataOnly; }
                                //現在オブジェクト型ではないが、セーブデータ上ではそうだったデータ項目の数を取得
	int getNumIsObjOnMemOnly() const { return m_numIsObjOnMemOnly; }
                                //現在オブジェクト型だが、セーブデータ上ではそうではなかったデータ項目の数を取得
	int getNumIsArrOnSaveDataOnly() const { return m_numIsArrOnSaveDataOnly; }
                                //現在配列型ではないが、セーブデータ上ではそうだったデータ項目の数を取得
	int getNumIsArrOnMemOnly() const { return m_numIsArrOnMemOnly; }
                                //現在配列型だが、セーブデータ上ではそうではなかったデータ項目の数を取得
	int getNumIsPtrOnSaveDataOnly() const { return m_numIsPtrOnSaveDataOnly; }
                                //現在ポインタ型ではないが、セーブデータ上ではそうだったデータ項目の数を取得
	int getNumIsPtrOnMemOnly() const { return m_numIsPtrOnMemOnly; }
                                //現在ポインタ型だが、セーブデータ上ではそうではなかったデータ項目の数を取得
	int getNumIsNulOnSaveDataOnly() const { return m_numIsNulOnSaveDataOnly; }
                                //現在ヌルではないが、セーブデータ上ではそうだったデータ項目の数を取得
	int getNumIsNulOnMemOnly() const { return m_numIsNulOnMemOnly; }
                                //現在ヌルだが、セーブデータ上ではそうではなかったデータ項目の数を取得
	void addNumSmallerSizeItem(){ ++m_numSmallerSizeItem; }//サイズが縮小されたデータ項目の数をカウントアップ
	void addNumLargerSizeItem(){ ++m_numLargerSizeItem; }//サイズが拡大されたデータ項目の数をカウントアップ
	void addNumSmallerArrItem(){ ++m_numSmallerArrItem; }//配列要素数が縮小されたデータ項目の数をカウントアップ
	void addNumLargerArrItem(){ ++m_numLargerArrItem; }//配列要素数が拡大されたデータ項目の数をカウントアップ
	void addNumIsOnlyOnSaveData(){ ++m_numIsOnlyOnSaveData; }
                                                    //セーブデータ上にのみ存在するデータ項目の数をカウントアップ
	void addNumIsOnlyOnMem(){ ++m_numIsOnlyOnMem; }//セーブデータ上にないデータ項目の数をカウントアップ
	void addNumIsObjOnSaveDataOnly(){ ++m_numIsObjOnSaveDataOnly; }
                    //現在オブジェクト型ではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsObjOnMemOnly(){ ++m_numIsObjOnMemOnly; }
                    //現在オブジェクト型だが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumIsArrOnSaveDataOnly(){ ++m_numIsArrOnSaveDataOnly; }
                    //現在配列型ではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsArrOnMemOnly(){ ++m_numIsArrOnMemOnly; }
                    //現在配列型だが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumIsPtrOnSaveDataOnly(){ ++m_numIsPtrOnSaveDataOnly; }
                    //現在ポインタ型ではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsPtrOnMemOnly(){ ++m_numIsPtrOnMemOnly; }
                    //現在ポインタ型だが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumIsNulOnSaveDataOnly(){ ++m_numIsNulOnSaveDataOnly; }
                    //現在ヌルではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsNulOnMemOnly(){ ++m_numIsNulOnMemOnly; }
                    //現在ヌルだが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumSmallerSizeItem(const bool enabled){ if (enabled) addNumSmallerSizeItem(); }
                                                    //サイズが縮小されたデータ項目の数をカウントアップ
	void addNumLargerSizeItem(const bool enabled){ if (enabled) addNumLargerSizeItem(); }
                                                    //サイズが拡大されたデータ項目の数をカウントアップ
	void addNumSmallerArrItem(const bool enabled){ if (enabled) addNumSmallerArrItem(); }
                                                    //配列要素数が縮小されたデータ項目の数をカウントアップ
```

```
	void addNumLargerArrItem(const bool enabled){ if (enabled) addNumLargerArrItem(); }
                                                        //配列要素数が拡大されたデータ項目の数をカウントアップ
	void addNumIsOnlyOnSaveData(const bool enabled){ if (enabled) addNumIsOnlyOnSaveData(); }
                                                    //セーブデータ上にのみ存在するデータ項目の数をカウントアップ
	void addNumIsOnlyOnMem(const bool enabled){ if (enabled) addNumIsOnlyOnMem(); }
                                                     //セーブデータ上にないデータ項目の数をカウントアップ
	void addNumIsObjOnSaveDataOnly(const bool enabled){ if (enabled) addNumIsObjOnSaveDataOnly(); }
                    //現在オブジェクト型ではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsObjOnMemOnly(const bool enabled){ if (enabled) addNumIsObjOnMemOnly(); }
                    //現在オブジェクト型だが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumIsArrOnSaveDataOnly(const bool enabled){ if (enabled) addNumIsArrOnSaveDataOnly(); }
                    //現在配列型ではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsArrOnMemOnly(const bool enabled){ if (enabled) addNumIsArrOnMemOnly(); }
                    //現在配列型だが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumIsPtrOnSaveDataOnly(const bool enabled){ if (enabled) addNumIsPtrOnSaveDataOnly(); }
                    //現在ポインタ型ではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsPtrOnMemOnly(const bool enabled){ if (enabled) addNumIsPtrOnMemOnly(); }
                    //現在ポインタ型だが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	void addNumIsNulOnSaveDataOnly(const bool enabled){ if (enabled) addNumIsNulOnSaveDataOnly(); }
                    //現在ヌルではないが、セーブデータ上ではそうだったデータ項目の数をカウントアップ
	void addNumIsNulOnMemOnly(const bool enabled){ if (enabled) addNumIsNulOnMemOnly(); }
                    //現在ヌルだが、セーブデータ上ではそうではなかったデータ項目の数をカウントアップ
	std::size_t getCopiedSize() const { return m_copiedSize; }//コピー済みサイズ取得
	std::size_t getSaveDataSize() const { return m_hasFatalError ? 0 : m_saveDataSize; }//セーブデータサイズ取得
	void setSaveDataSize(const std::size_t save_data_size){ m_saveDataSize = save_data_size; }
                                                                                    //セーブデータサイズをセット
public:
	//メソッド
	//コピー済みサイズ追加
	std::size_t addCopiedSize(const std::size_t size)
	{
		m_copiedSize += size;
		return m_copiedSize;
	}
	//処理結果に加算
	void addResult(const CResult& src)
	{
		setHasFatalError(src.m_hasFatalError);//致命的なエラーあり
		m_numSmallerSizeItem += src.m_numSmallerSizeItem;//サイズが縮小されたデータ項目の数
		m_numLargerSizeItem += src.m_numLargerSizeItem;//サイズが拡大されたデータ項目の数
		m_numSmallerArrItem += src.m_numSmallerArrItem;//配列要素数が縮小されたデータ項目の数
		m_numLargerArrItem += src.m_numLargerArrItem;//配列要素数が拡大されたデータ項目の数
		m_numIsOnlyOnSaveData += src.m_numIsOnlyOnSaveData;//セーブデータ上にのみ存在するデータ項目の数
		m_numIsOnlyOnMem += src.m_numIsOnlyOnMem;//セーブデータ上にないデータ項目の数
		m_numIsObjOnSaveDataOnly += src.m_numIsObjOnSaveDataOnly;
                                //現在オブジェクト型ではないが、セーブデータ上ではそうだったデータ項目の数
		m_numIsObjOnMemOnly += src.m_numIsObjOnMemOnly;
                                //現在オブジェクト型だが、セーブデータ上ではそうではなかったデータ項目の数
		m_numIsArrOnSaveDataOnly += src.m_numIsArrOnSaveDataOnly;
                                //現在配列型ではないが、セーブデータ上ではそうだったデータ項目の数
		m_numIsArrOnMemOnly += src.m_numIsArrOnMemOnly;
                                //現在配列型だが、セーブデータ上ではそうではなかったデータ項目の数
		m_numIsPtrOnSaveDataOnly += src.m_numIsPtrOnSaveDataOnly;
                                //現在ポインタ型ではないが、セーブデータ上ではそうだったデータ項目の数
		m_numIsPtrOnMemOnly += src.m_numIsPtrOnMemOnly;
                                //現在ポインタ型だが、セーブデータ上ではそうではなかったデータ項目の数
		m_numIsNulOnSaveDataOnly += src.m_numIsNulOnSaveDataOnly;
                                //現在ヌルではないが、セーブデータ上ではそうだったデータ項目の数
		m_numIsNulOnMemOnly += src.m_numIsNulOnMemOnly;
                                //現在ヌルだが、セーブデータ上ではそうではなかったデータ項目の数
		m_copiedSize += src.m_copiedSize;//コピー済みサイズ
	}
	//処理結果を計上
	void addResult(const CItemBase& src)
	{
```

```
            addNumSmallerSizeItem(src.nowSizeIsSamall());//サイズが縮小されたデータ項目の数
            addNumLargerSizeItem(src.nowSizeIsLarge());//サイズが拡大されたデータ項目の数
            addNumSmallerArrItem(src.nowArrIsSmall());//配列要素数が縮小されたデータ項目の数
            addNumLargerArrItem(src.nowArrIsLarge());//配列要素数が拡大されたデータ項目の数
            addNumIsOnlyOnSaveData(src.isOnlyOnSaveData());//セーブデータ上にのみ存在するデータ項目の数
            addNumIsOnlyOnMem(src.isOnlyOnMem());//セーブデータ上にないデータ項目の数
            addNumIsObjOnSaveDataOnly(src.nowIsNotObjButSaveDataIs());
                                    //現在オブジェクト型ではないが、セーブデータ上ではそうだったデータ項目の数
            addNumIsObjOnMemOnly(src.nowIsObjButSaveDataIsNot());
                                    //現在オブジェクト型だが、セーブデータ上ではそうではなかったデータ項目の数
            addNumIsArrOnSaveDataOnly(src.nowIsNotArrButSaveDataIs());
                                    //現在配列型ではないが、セーブデータ上ではそうだったデータ項目の数
            addNumIsArrOnMemOnly(src.nowIsArrButSaveDataIsNot());
                                    //現在配列型だが、セーブデータ上ではそうではなかったデータ項目の数
            addNumIsPtrOnSaveDataOnly(src.nowIsNotPtrButSaveDataIs());
                                    //現在ポインタ型ではないが、セーブデータ上ではそうだったデータ項目の数
            addNumIsPtrOnMemOnly(src.nowIsPtrButSaveDataIsNot());
                                    //現在ポインタ型だが、セーブデータ上ではそうではなかったデータ項目の数
            addNumIsNulOnSaveDataOnly(src.nowIsNotNulButSaveDataIs());
                                    //現在ヌルではないが、セーブデータ上ではそうだったデータ項目の数
            addNumIsNulOnMemOnly(src.nowIsNulButSaveDataIsNot());
                                    //現在ヌルだが、セーブデータ上ではそうではなかったデータ項目の数
    }
public:
    //コンストラクタ
    CResult() :
            m_hasFatalError(false),
            m_numSmallerSizeItem(0),
            m_numLargerSizeItem(0),
            m_numSmallerArrItem(0),
            m_numLargerArrItem(0),
            m_numIsOnlyOnSaveData(0),
            m_numIsOnlyOnMem(0),
            m_numIsObjOnSaveDataOnly(0),
            m_numIsObjOnMemOnly(0),
            m_numIsArrOnSaveDataOnly(0),
            m_numIsArrOnMemOnly(0),
            m_numIsPtrOnSaveDataOnly(0),
            m_numIsPtrOnMemOnly(0),
            m_numIsNulOnSaveDataOnly(0),
            m_numIsNulOnMemOnly(0),
            m_copiedSize(0),
            m_saveDataSize(0)
    {}
    //デストラクタ
    ~CResult()
    {}
private:
    //フィールド
    bool m_hasFatalError;//致命的なエラーあり
    short m_numSmallerSizeItem;//サイズが縮小されたデータ項目の数
    short m_numLargerSizeItem;//サイズが拡大されたデータ項目の数
    short m_numSmallerArrItem;//配列要素数が縮小されたデータ項目の数
    short m_numLargerArrItem;//配列要素数が拡大されたデータ項目の数
    short m_numIsOnlyOnSaveData;//セーブデータ上にのみ存在するデータ項目の数
    short m_numIsOnlyOnMem;//セーブデータ上にないデータ項目の数
    short m_numIsObjOnSaveDataOnly;//現在オブジェクト型ではないが、セーブデータ上ではそうだったデータ項目の数
    short m_numIsObjOnMemOnly;//現在オブジェクト型だが、セーブデータ上ではそうではなかったデータ項目の数
    short m_numIsArrOnSaveDataOnly;//現在配列型ではないが、セーブデータ上ではそうだったデータ項目の数
    short m_numIsArrOnMemOnly;//現在配列型だが、セーブデータ上ではそうではなかったデータ項目の数
    short m_numIsPtrOnSaveDataOnly;//現在ポインタ型ではないが、セーブデータ上ではそうだったデータ項目の数
    short m_numIsPtrOnMemOnly;//現在ポインタ型だが、セーブデータ上ではそうではなかったデータ項目の数
    short m_numIsNulOnSaveDataOnly;//現在ヌルではないが、セーブデータ上ではそうだったデータ項目の数
    short m_numIsNulOnMemOnly;//現在ヌルだが、セーブデータ上ではそうではなかったデータ項目の数
    std::size_t m_copiedSize;//コピー済みサイズ
```

```
	std::size_t m_saveDataSize;//セーブデータサイズ
};
```

▼ アーカイブ読み書きクラス

【アーカイブ読み書き基底クラス】

```
//--------------------
//アーカイブ読み書き基底クラス
typedef unsigned char byte;//バッファ用
class CIOArchiveBase
{
	friend class CIOArchiveHelper;
	friend class COArchiveHelper;
	friend class CIArchiveHelper;
public:
	//型
	typedef std::map<crc32_t, const CItemBase> itemList_t;//データ項目リスト型
public:
	//アクセッサ
	CResult& getResult(){ return m_result; }//処理結果を取得
	const CResult& getResult() const { return m_result; }//処理結果を取得
	bool hasFatalError() const { return m_result.hasFatalError(); }//致命的なエラーありか？
	std::size_t getSaveDataSize() const { return m_result.getSaveDataSize(); }//セーブデータサイズ取得
	const void* getSaveData() const { return m_buff; }//セーブデータバッファの先頭ポインタを取得
protected:
	const byte* getBuffPtr() const { return m_buff; }//セーブデータバッファの先頭ポインタを取得
	const std::size_t getBuffSize() const { return m_buffSize; }//セーブデータバッファのサイズを取得
	const std::size_t getBuffUsed() const { return m_buffPos; }//セーブデータバッファの使用量を取得
	const std::size_t getBuffPos() const { return m_buffPos; }//セーブデータバッファの現在位置を取得
	const std::size_t getBuffRemain() const { return m_buffSize - m_buffPos; }//セーブデータバッファの残量を取得
	byte* getBuffNowPtr(){ return m_buff + m_buffPos; }//セーブデータバッファの現在位置のポインタを取得
	bool buffIsFull() const { return m_buffPos >= m_buffSize; }//バッファの現在位置が末端に到達したか？
	std::size_t getItemListNum() const { return m_itemList->size(); }//データ項目リスト
protected:
	//メソッド
	//処理結果を合成
	void addResult(const CResult& src)
	{
		m_result.addResult(src);
	}
	//リストにデータ項目追加
	void addItem(const CItemBase& item)
	{
		//ワーク領域を設定（グローバル多態アロケータ）
		//※関数を抜けると自動的に元に戻る
		CTempPolyStackAllocator alloc(m_workBuff);

		//データ項目を追加
		assert(m_itemList->find(item.m_nameCrc) == m_itemList->end());//重複チェック
		m_itemList->emplace(item.m_nameCrc, item);
	}
	//リストからデータ項目を検索
	const CItemBase* findItem(const crc32_t name_crc) const
	{
		auto ite = m_itemList->find(name_crc);
		if (ite == m_itemList->end())//見つかったか？
			return nullptr;//見つからなかった
		return &ite->second;
	}
private:
	//データリスト作成
	void createItemList()
```

```
	{
		//ワーク領域を設定（グローバル多態アロケータ）
		//※関数を抜けると自動的に元に戻る
		CTempPolyStackAllocator alloc(m_workBuff);

		//データ項目リストを生成
		m_itemList = new itemList_t;
	}
	//データリスト破棄
	void destroyItemList()
	{
		//ワーク領域を設定（グローバル多態アロケータ）
		//※関数を抜けると自動的に元に戻る
		CTempPolyStackAllocator alloc(m_workBuff);

		//データ項目リストを破棄
		if (m_itemList)
		{
			delete m_itemList;
			m_itemList = nullptr;
		}

		//ワークバッファ用スタックアロケータクリア
		m_workBuff.clearN();
	}
protected:
	//コンストラクタ
	CIOArchiveBase(void* buff, const std::size_t buff_size, void* work_buff, std::size_t work_buff_size) :
		m_nestLevel(0),
		m_buff(reinterpret_cast<byte*>(buff)),
		m_buffSize(buff_size),
		m_buffPos(0),
		m_workBuff(work_buff, work_buff_size),
		m_itemList(nullptr)
	{
		//データリスト作成
		createItemList();
	}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	CIOArchiveBase(CIOArchiveBase& parent) :
		m_nestLevel(parent.m_nestLevel + 1),
		m_buff(parent.getBuffNowPtr()),
		m_buffSize(parent.getBuffRemain()),
		m_buffPos(0),
		m_workBuff(const_cast<IStackAllocator::byte*>(parent.m_workBuff.getNowPtrN()),
		                                                    parent.m_workBuff.getRemain()),
		m_itemList(nullptr)
	{
		//データリスト作成
		createItemList();
	}
	//デストラクタ
	~CIOArchiveBase()
	{
		//データリスト破棄
		destroyItemList();
	}
protected:
	//フィールド
	CResult m_result;//処理結果
	int m_nestLevel;//データのネストレベル
	byte* m_buff;//セーブデータバッファ
	const std::size_t m_buffSize;//セーブデータバッファのサイズ
	std::size_t m_buffPos;//セーブデータバッファの処理位置
```

```
	CStackAllocator m_workBuff;//ワークバッファ用スタックアロケータ
	itemList_t* m_itemList;//データ項目リスト
};
```

【アーカイブ書き込みクラス】

```
//--------------------
//アーカイブ書き込みクラス
template<class STYLE>
class COArchive : public CIOArchiveBase
{
public:
	//定数
	static const bool is_read = false;//読み込みクラスか？
	static const bool is_write = true;//書き込みクラスか？
public:
	//型
	typedef STYLE arcStyle_t;//アーカイブ形式型
private:
	//定数
	//ステート
	enum stateEnum : int
	{
		st_SERIALIZE,//シリアライズ中
		st_SERIALIZE_PHASE_BLOCK,//シリアライズ：ブロック処理フェーズ
		st_SERIALIZE_PHASE_ARRAY,//シリアライズ：配列ブロック処理フェーズ
		st_SERIALIZE_PHASE_ELEMENT,//シリアライズ：要素処理フェーズ
		st_SERIALIZE_PHASE_SAVE_DATA,//シリアライズ：データのセーブフェーズ
		st_SERIALIZE_PHASE_COLLECT,//シリアライズ：収集フェーズ
		st_SERIALIZE_PHASE_COLLECT_END,//シリアライズ：収集フェーズ終了
		st_SERIALIZE_END,//シリアライズ終了
	};
private:
	//アクセッサ
	stateEnum getState() const { return m_state; }//ステートを取得
	stateEnum setState(const stateEnum state){ stateEnum prev = m_state; m_state = state; return prev; }
                                                                    //ステートを変更　※変更前のステートを返す
public:
	//オペレータ
	//「&」オペレータ
	//※データ項目指定＆データ書き込み用処理
	template<class T>
	COArchive& operator&(const CItem<T> item_obj)
	{
		if (m_result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return *this;

		//現在処理中のアーカイブオブジェクト
		COArchive& arc = *this;

		//st_SERIALIZE_PHASE_SAVE_DATA 時以外にこの処理に来るのはプログラムに間違いがある
		//※収集処理で operator&() を使っている時など
		assert(arc.getState() == st_SERIALIZE_PHASE_SAVE_DATA);

		//文字列型専用処理
		if (typeid(T) == typeid(serial::str_t))
		{
			if (item_obj.m_itemP)
			{
				//文字列長を格納し直す
				*const_cast<std::size_t*>(&item_obj.m_itemSize) =
                                        strlen(reinterpret_cast<const char*>(item_obj.m_itemP)) + 1;
			}
			else
			{
				//ヌル時は０に
```

```
                        *const_cast<std::size_t*>(&item_obj.m_itemSize) = 0;
                }
        }

        //データ項目を記録
        //※重複チェックのため
        addItem(item_obj);

        //データ書き込み
        if (item_obj.isObj())
        {
                //オブジェクトなら operator<<() で書き込み
                arc << item_obj;
        }
        else
        {
                //データをアーカイブに記録
                arc.m_style.writeDataItem(arc, item_obj, item_obj);

                //データ書き込み済み
                item_obj.setIsAlready();
        }

        return *this;
}
//「<<」オペレータ
//※データ書き込み
template<class T>
COArchive& operator<<(const CItem<T> item_obj)
{
        if (m_result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                return *this;

        //現在処理中のアーカイブオブジェクト
        COArchive& arc = *this;

        //st_SERIALIZE, st_SERIALIZE_PHASE_SAVE_DATA, st_SERIALIZE_PHASE_COLLECT 時以外に
        //この処理に来るのはプログラムに間違いがある
        assert(arc.getState() == st_SERIALIZE ||
                arc.getState() == st_SERIALIZE_PHASE_SAVE_DATA ||
                arc.getState() == st_SERIALIZE_PHASE_COLLECT);

        //オブジェクトでなければダメ
        //※何かしらのシリアライズ用関数オブジェクトを実装した型か？
        assert(item_obj.isObj() == true);

        //ネストレベルが 0 ならシグネチャーを書き込み
        if (m_nestLevel == 0)
                arc.m_style.writeSignature(arc);//シグネチャー書き込み

        //バージョン取得
        CVersionDef<T> ver_def;
        CVersion ver(ver_def);

        //ブロック開始情報書き込み
        arc.m_style.writeBlockHeader(arc, item_obj, ver);

        //ブロック開始
        std::size_t block_size = 0;//ブロックのサイズ

        {
                //ブロック処理用の子アーカイブオブジェクトを生成
                COArchive& parent_arc = arc;
                COArchive arc(parent_arc, st_SERIALIZE_PHASE_BLOCK);
```

```
//ヌルでなければ処理する
if (!item_obj.isNul() && !arc.hasFatalError())
{
	//配列要素数取得
	const std::size_t array_elem_num = item_obj.m_arrNum;

	//配列ブロック開始情報書き込み
	arc.m_style.writeArrayHeader(arc, item_obj, array_elem_num);

	//配列ブロック開始
	std::size_t array_block_size = 0;//配列ブロックのサイズ

	{
		//配列ブロック処理用の孫アーカイブオブジェクトを生成
		COArchive& parent_arc = arc;
		COArchive arc(parent_arc, st_SERIALIZE_PHASE_ARRAY);

		//データ書き込み先の先頭ポインタ（配列の先頭）を記憶
		//※配列ループ処理中に item_obj.m_itemP を書き換えるので、元に戻せるようにしておく
		const void* item_p_top = item_obj.m_itemP;

		//要素ループ
		const std::size_t loop_elem_num = item_obj.getElemNum();
											//配列の要素数もしくは（配列じゃなければ）1 を返す
		for (std::size_t index = 0; index < loop_elem_num && !arc.hasFatalError(); ++index)
		{
			//要素開始情報書き込み
			arc.m_style.writeElemHeader(arc, item_obj, index);

			//要素開始
			short items_num = 0;//データ項目数
			std::size_t elem_size = 0;//この要素のデータサイズ
			{
				//要素処理用のひ孫アーカイブオブジェクトを生成
				COArchive& parent_arc = arc;
				COArchive arc(parent_arc, st_SERIALIZE_PHASE_ELEMENT);

				//データのセーブフェーズに変更
				arc.setState(st_SERIALIZE_PHASE_SAVE_DATA);

				//シリアライズ処理（シリアライズ&デシリアライズ兼用処理）呼び出し
				{
					serialize<COArchive, T> functor;
					functor(arc, item_obj.template getConst<T>(), ver, ver);
				}

				//セーブ処理（シリアライズ専用処理）呼び出し
				{
					save<COArchive, T> functor;
					functor(arc, item_obj.template getConst<T>(), ver);
				}

				//要素終了情報書き込み
				//※例えば、バイナリスタイルでは、要素のヘッダ部にデータ項目数と
				//  データサイズを書き込み、要素の最後までシークする
				arc.m_style.writeElemFooter(parent_arc, arc, item_obj, index, items_num,
												elem_size);

				//データ書き込み先のポインタを配列の次の要素に更新
				if (item_obj.m_itemP)
				{
					*const_cast<void**>(&item_obj.m_itemP) =
						reinterpret_cast<T*>(const_cast<void*>(item_obj.m_itemP)) + 1;
				}
			}
```

```
                        }

                        //配列の先頭ポインタ（元のポインタ）に戻す
                        *const_cast<const void**>(&item_obj.m_itemP) = item_p_top;

                        //配列ブロック終了情報書き込み
                        //※例えば、バイナリスタイルでは、配列ブロックのヘッダ部に配列要素数とデータサイズを
                        //  書き込み、要素の最後までシークする
                        arc.m_style.writeArrayFooter(parent_arc, arc, item_obj, array_block_size);
                  }
            }

            //データ収集処理（シリアライズ専用処理）呼び出し
            arc.setState(st_SERIALIZE_PHASE_COLLECT);
            {
                  collector<COArchive, T> functor;
                  functor(arc, item_obj.template getConst<T>(), ver);
            }
            arc.setState(st_SERIALIZE_PHASE_COLLECT_END);

            //ブロック終了情報書き込み
            //※例えば、バイナリスタイルでは、ブロックのヘッダ部にデータサイズを書き込み、
            //  ブロックの最後までシークする
            arc.m_style.writeBlockFooter(parent_arc, arc, item_obj, block_size);
      }

      //ネストレベルが 0 ならターミネータを書き込み
      if (m_nestLevel == 0)
            arc.m_style.writeTerminator(arc);//ターミネータ書き込み

      //データ書き込み済みにする
      item_obj.setIsAlready();

      //処理結果にセーブデータサイズをセットする
      m_result.setSaveDataSize(getBuffUsed());

      //最終的に致命的なエラーがあれば呼び出し
      if (m_nestLevel == 0 && arc.hasFatalError())
      {
            fatalSerializeErrorOccurred<COArchive, T> functor;
            functor(arc, item_obj.template get<T>(), ver);
      }

      return *this;
  }
public:
  //コンストラクタ
  COArchive(void* buff, const std::size_t buff_size, void* work_buff, std::size_t work_buff_size) :
        CIOArchiveBase(buff, buff_size, work_buff, work_buff_size),
        m_parent(nullptr),
        m_style(),
        m_state(st_SERIALIZE)
  {}
  template<typename BUFF_T, std::size_t BUFF_SIZE, typename WORK_T, std::size_t WORK_SIZE>
  COArchive(BUFF_T(&buff)[BUFF_SIZE], WORK_T(&work_buff)[WORK_SIZE]) :
        CIOArchiveBase(buff, BUFF_SIZE, work_buff, WORK_SIZE),
        m_parent(nullptr),
        m_style(),
        m_state(st_SERIALIZE)
  {}
  template<typename WORK_T, std::size_t WORK_SIZE>
  COArchive(void* buff, const std::size_t buff_size, WORK_T(&work_buff)[WORK_SIZE]) :
        CIOArchiveBase(buff, buff_size, work_buff, WORK_SIZE),
        m_parent(nullptr),
        m_style(),
```

```
            m_state(st_SERIALIZE)
      {}
      //親を受け取るコンストラクタ
      //※処理階層が深くなるごとにコピーが行われる
      COArchive(COArchive& parent, const stateEnum state) :
            CIOArchiveBase(parent),
            m_parent(&parent),
            m_style(parent.m_style),
            m_state(state)
      {}
      //デストラクタ
      ~COArchive()
      {}
private:
      //フィールド
      COArchive* m_parent;//親アーカイブオブジェクト
      arcStyle_t m_style;//アーカイブスタイルオブジェクト
      stateEnum m_state;//ステート
};
```

【アーカイブ読み込みクラス】

```
//--------------------
//アーカイブ読み込みクラス
template<class STYLE>
class CIArchive : public CIOArchiveBase
{
public:
      //定数
      static const bool is_read = true;//読み込みクラスか？
      static const bool is_write = false;//書き込みクラスか？
public:
      //型
      typedef STYLE arcStyle_t;//アーカイブ形式型
private:
      //定数
      //ステート
      enum stateEnum : int
      {
            st_DESERIALIZE,//デシリアライズ中
            st_DESERIALIZE_PHASE_BLOCK,//シリアライズ：ブロック処理フェーズ
            st_DESERIALIZE_PHASE_ARRAY,//シリアライズ：配列ブロック処理フェーズ
            st_DESERIALIZE_PHASE_ELEMENT,//シリアライズ：要素処理フェーズ
            st_DESERIALIZE_PHASE_BEFORE_LOAD,//デシリアライズ：ロード前処理フェーズ
            st_DESERIALIZE_PHASE_MAKE_LIST,//デシリアライズ：対象データ項目リスト作成フェーズ
            st_DESERIALIZE_PHASE_LOAD_DATA,//デシリアライズ：データのロードフェーズ
            st_DESERIALIZE_PHASE_LOAD_OBJECT,//デシリアライズ：オブジェクトのロードフェーズ
            st_DESERIALIZE_PHASE_LOAD_OBJECT_END,//デシリアライズ：オブジェクトのロードフェーズ終了
            st_DESERIALIZE_PHASE_NOTICE_UNRECOGNIZED,//デシリアライズ：認識できなかったデータ項目の通知フェーズ
            st_DESERIALIZE_PHASE_NOTICE_UNLOADED,//デシリアライズ：ロードできなかったデータ項目の通知フェーズ
            st_DESERIALIZE_PHASE_AFTEER_LOAD,//デシリアライズ：ロード後処理フェーズ
            st_DESERIALIZE_PHASE_BEFORE_DISTRIBUTE,//デシリアライズ：分配前フェーズ
            st_DESERIALIZE_PHASE_DISTRIBUTE,//デシリアライズ：分配フェーズ
            st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT,//デシリアライズ：オブジェクトの分配フェーズ
            st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT_END,//デシリアライズ：オブジェクトの分配フェーズ終了
            st_DESERIALIZE_PHASE_AFTER_DISTRIBUTE,//デシリアライズ：分配後フェーズ
            st_DESERIALIZE_END,//デシリアライズ終了
      };
private:
      //アクセッサ
      stateEnum getState() const { return m_state; }//ステートを取得
      stateEnum setState(const stateEnum state){ stateEnum prev = m_state; m_state = state; return prev; }
                                                      //ステートを変更　※変更前のステートを返す
      CItemBase* getTargetObjItem(){ return m_targetObjItem; }//オブジェクト処理の対象データ項目をセット
      const CItemBase* getTargetObjItem() const { return m_targetObjItem; }//オブジェクト処理の対象データ項目をセット
      void setTargetObjItem(CItemBase& item){ m_targetObjItem = &item; }//オブジェクト処理の対象データ項目をセット
```

```
	void resetTargetObjItem(){ m_targetObjItem = nullptr; }//オブジェクト処理の対象データ項目をリセット
	CItemBase* getTargetObjItemDelegate(){ return m_targetObjItemDelegate; }
                                                    //オブジェクト処理の対象データ項目の委譲データ項目をセット
	const CItemBase* getTargetObjItemDelegate() const { return m_targetObjItemDelegate; }
                                                    //オブジェクト処理の対象データ項目の委譲データ項目をセット
	void setTargetObjItemDelegate(CItemBase* item){ m_targetObjItemDelegate = item; }
                                                    //オブジェクト処理の対象データ項目の委譲データ項目をセット
	void resetTargetObjItemDelegate(){ m_targetObjItemDelegate = nullptr; }
                                                   //オブジェクト処理の対象データ項目の委譲データ項目をリセット
public:
	//オペレータ
	//「&」オペレータ
	//※データ項目指定用処理
	template<class T>
	CIArchive& operator&(const CItem<T> item_obj)
	{
		if (m_result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return *this;

		//現在処理中のアーカイブオブジェクト
		CIArchive& arc = *this;

		//st_DESERIALIZE_PHASE_MAKE_LIST, st_DESERIALIZE_PHASE_LOAD_OBJECT,
		//st_DESERIALIZE_PHASE_LOAD_OBJECT_END 時以外にこの処理に来るのはプログラムに間違いがある
		//※ロード前処理やロード後処理、通知処理、分配処理で operator&() を使っている時など
		assert(arc.getState() == st_DESERIALIZE_PHASE_MAKE_LIST ||
			arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT ||
			arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT_END ||
			arc.getState() == st_DESERIALIZE_PHASE_NOTICE_UNRECOGNIZED);

		//オブジェクト処理モード終了時はなにもしない
		if (arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT_END)
		{
			return *this;
		}
		//オブジェクト処理モード時は対象データ項目のみ処理する
		else if (arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT)
		{
			if (*m_targetObjItem == item_obj)
			{
				arc >> item_obj;//オブジェクト読み込み
				arc.setState(st_DESERIALIZE_PHASE_LOAD_OBJECT_END);//オブジェクト処理モード終了
			}
			return *this;
		}
		//認識できなかったオブジェクトの通知モード時は委譲データ項目をセットして終了する
		else if (arc.getState() == st_DESERIALIZE_PHASE_NOTICE_UNRECOGNIZED)
		{
			if (m_targetObjItemDelegate)
			{
				m_targetObjItemDelegate->copyForce(item_obj);
			}
			return *this;
		}

		//一旦は「セーブデータに存在しないデータ項目」という扱いにしておく
		//※ロード終了後もそのままならロードできなかったデータ項目として処理する
		item_obj.setIsOnlyOnMem();

		//データ項目を記録
		//※全ての記録が終わった後、データを読み込みながらデータ項目に書き込んでいく
		addItem(item_obj);

		return *this;
	}
```

```
//「>>」オペレータ
//※データ読み込み
template<class T>
CIArchive& operator>>(CItem<T> item_obj_now)
{
	if (m_result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return *this;

	//現在処理中のアーカイブオブジェクト
	CIArchive& arc = *this;

	//st_DESERIALIZE, st_DESERIALIZE_PHASE_MAKE_LIST, st_DESERIALIZE_PHASE_LOAD_DATA,
	//st_DESERIALIZE_PHASE_LOAD_OBJECT,st_DESERIALIZE_PHASE_LOAD_OBJECT_END 時以外に
	//この処理に来るのはプログラムに間違いがある
	//※ロード処理（ロード前後処理含む）、通知処理で operator>>() を使っている時など
	assert(arc.getState() == st_DESERIALIZE ||
		arc.getState() == st_DESERIALIZE_PHASE_MAKE_LIST ||
		arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT ||
		arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT_END ||
		arc.getState() == st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT ||
		arc.getState() == st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT_END);

	//データ項目リトライ処理⇒ブロック読み込みのための委譲データ項目を準備
	CItemBase* delebate_item = nullptr;

	//対象データ項目リスト作成フェーズ中なら operator&() に処理を回す
	if (arc.getState() == st_DESERIALIZE_PHASE_MAKE_LIST)
	{
		return *this & item_obj_now;
	}
	//オブジェクト処理モード時は委譲データ項目を取得する
	else if (arc.getState() == st_DESERIALIZE_PHASE_LOAD_OBJECT)
	{
		//委譲データ項目を取得
		delebate_item = m_targetObjItemDelegate;
	}
	//分配オブジェクト処理モード終了時はなにもしない
	else if (arc.getState() == st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT_END)
	{
		return *this;
	}
	//分配オブジェクトモード時は対象データ項目のみ処理する
	else if (arc.getState() == st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT)
	{
		if (*m_targetObjItem == item_obj_now)
		{
			//オブジェクト処理モード終了状態にして処理続行
			arc.setState(st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT_END);
		}
	}

	//パラメータチェック
	//assert(!item_obj_now.isOnlyOnSaveData());
	//※セーブデータにしか存在しないデータは処理不可
	//※配信処理を間違えて以内限り、そのような状態にはならないはず

	//ネストレベルが 0 ならパース
	if (m_nestLevel == 0)
		arc.m_style.parse(arc);//パース

	//ネストレベルが 0 ならシグネチャーを読み込み
	if (m_nestLevel == 0)
		arc.m_style.readSignature(arc);//シグネチャー読み込み
```

```
		//バージョン取得
		CVersionDef<T> now_ver_def;
		CVersion now_ver(now_ver_def);//現在のバージョン
		CVersion ver;//読み込み用のバージョン
		//ブロック開始情報読み込み
		std::size_t block_size = 0;
		CItem<T> item_obj(item_obj_now);
		arc.m_style.readBlockHeader(arc, item_obj_now, delebate_item, now_ver, item_obj, ver, block_size);

		//ブロック開始
		{
			//ブロック処理用の子アーカイブオブジェクトを生成
			CIArchive& parent_arc = arc;
			CIArchive arc(parent_arc, st_DESERIALIZE_PHASE_BLOCK);

			//集計準備
			const std::size_t elem_num = item_obj.getElemNum();//要素数（配列要素ではなく、非配列なら 1）
			std::size_t elem_num_loaded = 0;//実際に読み込んだ要素数（配列要素ではなく、非配列なら 1）

			if (!item_obj.isNul() && !arc.hasFatalError())//【セーブデータ上の】要素がヌルでなければ処理する
			{

				//配列ブロック開始情報読み込み
				std::size_t array_elem_num = 0;//配列要素数
				std::size_t array_block_size = 0;//配列ブロックサイズ
				arc.m_style.readArrayHeader(arc, item_obj, array_elem_num, array_block_size);

				//配列ブロック開始
				{
					//配列ブロック処理用の孫アーカイブオブジェクトを生成
					CIArchive& parent_arc = arc;
					CIArchive arc(parent_arc, st_DESERIALIZE_PHASE_ARRAY);

					//データ読み込み先の先頭ポインタ（配列の先頭）を記憶
					//※配列ループ処理中に item_obj.m_itemP を書き換えるので、元に戻せるようにしておく
					const void* item_p_top = item_obj.m_itemP;

					//配列ループ
					//※ループ終了条件判定に index < elem_num は含めない
					//　（アーカイブスタイルによっては事前に正確な配列要素数が得られない可能性があるため）
					for (std::size_t index = 0; !arc.hasFatalError(); ++index)
														//【セーブデータ上の】配列要素数分データ読み込み
					{
						if (!item_obj.isArr())
						{
							//非配列要素の終了判定
							if (index == 1)
								break;
						}
						else
						{
							//配列ブロック終了判定
							bool is_array_block_end = true;
							arc.m_style.tryAndReadArrayFooter(arc, item_obj, is_array_block_end);
							if (is_array_block_end)
								break;//配列ブロックの終了を検出したらループから抜ける
						}

						//デバッグ用の変数キャスト
						const T* debug_p = reinterpret_cast<const T*>(item_obj.m_itemP);

						//有効な配列要素か？（有効でなければ処理せず読み込むだけ）
						const bool is_valid_element = (!item_obj.nowIsNul() &&
														index < item_obj.getNowElemNum());
						if (is_valid_element)
```

```
		++elem_num_loaded;

	//要素開始情報読み込み
	std::size_t elem_size = 0;
	short items_num = 0;
	arc.m_style.readElemHeader(arc, item_obj, index, items_num, elem_size);

	//要素開始
	{
		//新しいアーカイブオブジェクトを生成
		CIArchive& parent_arc = arc;
		CIArchive arc(parent_arc, st_DESERIALIZE_PHASE_ELEMENT);

		//ロード前処理フェーズに変更
		arc.setState(st_DESERIALIZE_PHASE_BEFORE_LOAD);

		//ロード前処理（デシリアライズ専用処理）呼び出し
		if (is_valid_element)
		{
			beforeLoad<CIArchive, T> functor;
			functor(arc, item_obj.template get<T>(), ver, now_ver);
		}

		//対象データ項目リスト作成フェーズに変更
		arc.setState(st_DESERIALIZE_PHASE_MAKE_LIST);

		//デシリアライズ処理（シリアライズ＆デシリアライズ兼用処理）呼び出し
		//※データ項目リストを記録するだけ
		if (is_valid_element)
		{
			serialize<CIArchive, T> functor;
			functor(arc, item_obj.template getConst<T>(), ver, now_ver);
		}

		//ロード処理（デシリアライズ専用処理）呼び出し
		//※データ項目リストを記録するだけ
		if (is_valid_element)
		{
			load<CIArchive, T> functor;
			functor(arc, item_obj.template get<T>(), ver, now_ver);
		}

		//データのロードフェーズに変更
		arc.setState(st_DESERIALIZE_PHASE_LOAD_DATA);

		//実際のロード処理
		//※ループ終了条件判定に item_idx < items_num は含めない
		//  （アーカイブスタイルによっては事前に正確な項目数が得られない可能性が
		//    あるため）
		for (short item_idx = 0; !arc.hasFatalError(); ++item_idx)
		{
			//要素終了情報読み込み
			bool is_elem_end = true;
			arc.m_style.tryAndReadElemFooter(arc, item_obj, index, is_elem_end);
			if (is_elem_end)
				break;//要素の終了を検出したらループから抜ける

			bool is_init = true;//初回処理
			bool is_required_retry = false;//リトライ処理要求
			bool is_already_retry = false;//リトライ処理済み
			CItemBase delegate_child_item_for_retry;//リトライ用の委譲項目（実体）
			CItemBase* delegate_child_item_now = nullptr;//リトライ用の委譲項目
			while (is_init || (is_required_retry && !is_already_retry))
													//リトライは一回だけ有効
			{
```

```
is_init = false;//初回処理終了
if (is_required_retry)//リトライ時
      is_already_retry = true;
                  //リトライ済みにする（リトライは一回だけ）

//要素の読み込み
CItemBase child_item;
arc.m_style.readDataItem(arc, item_obj, delegate_child_item_now,
            child_item, is_valid_element, is_required_retry);

//ロードしたが保存先の指定がなかった項目の処理
//※既にリトライしている場合は実行しない
if (child_item.isOnlyOnSaveData() && !is_already_retry)
{
      //通知フェーズに変更
      arc.setState(st_DESERIALIZE_PHASE_NOTICE_UNRECOGNIZED);

      //通知
      if (is_valid_element)
      {
            //委譲データ項目の受け取り用にインスタンスをセット
            arc.setTargetObjItemDelegate(
                  &delegate_child_item_for_retry);

            //通知処理呼び出し
            noticeUnrecognizedItem<CIArchive, T> functor;
            functor(arc, item_obj.template get<T>(), ver,
                                    now_ver, child_item);

            //委譲データ項目の受け取り終了
            arc.resetTargetObjItemDelegate();

            //委譲データ項目の受け取り状態を判定
            if (delegate_child_item_for_retry.m_nameCrc != 0 &&
                        delegate_child_item_for_retry.m_itemP)
            {
                  //委譲データ項目が設定されたのでリトライ
                  delegate_child_item_now =
                              &delegate_child_item_for_retry;
                  is_required_retry = true;
            }
      }

      //データのロードフェーズに戻す
      arc.setState(st_DESERIALIZE_PHASE_LOAD_DATA);

      //リトライ
      //※オブジェクトの処理に入る前にリトライを実行
      if (is_required_retry)
            continue;
}

//オブジェクトの場合の処理
if (child_item.isObj())
{
      //オブジェクトのロードフェーズに変更
      arc.setState(st_DESERIALIZE_PHASE_LOAD_OBJECT);

      //オブジェクト処理対象データ項目をセット
      arc.setTargetObjItem(child_item);
      arc.setTargetObjItemDelegate(delegate_child_item_now);

      //通知処理呼び出し
      //※委譲データ項目がセットされている時のみ
      //if (arc.getState() !=
```

```
                                        //                st_DESERIALIZE_PHASE_LOAD_OBJECT_END)
                                        if (delegate_child_item_now)
                                        {
                                              noticeUnrecognizedItem<CIArchive, T> functor;
                                              functor(arc, item_obj.template get<T>(), ver,
                                                                        now_ver, child_item);
                                        }

                                        //デシリアライズ処理（シリアライズ＆デシリアライズ兼用処理）
                                        //呼び出し
                                        //※対象オブジェクトアイテムを処理する
                                        if (arc.getState() != st_DESERIALIZE_PHASE_LOAD_OBJECT_END)
                                        {
                                              serialize<CIArchive, T> functor;
                                              functor(arc, item_obj.template getConst<T>(), ver,
                                                                                  now_ver);
                                        }

                                        //ロード処理（デシリアライズ専用処理）呼び出し
                                        //※対象オブジェクトアイテムを処理する
                                        if (arc.getState() != st_DESERIALIZE_PHASE_LOAD_OBJECT_END)
                                        {
                                              load<CIArchive, T> functor;
                                              functor(arc, item_obj.template get<T>(), ver,
                                                                                   now_ver);
                                        }

                                        //オブジェクト処理対象データ項目をリセット
                                        arc.resetTargetObjItem();
                                        arc.resetTargetObjItemDelegate();
                                        delegate_child_item_now = nullptr;

                                        //未処理のままだったらブロックをスキップする
                                        if (arc.getState() != st_DESERIALIZE_PHASE_LOAD_OBJECT_END)
                                        {
                                              //オブジェクトのブロックをスキップ
                                              arc.m_style.skipReadBlock(arc);

                                              //処理済みにする
                                              child_item.setIsAlready();
                                        }

                                        //データのロードフェーズに戻す
                                        arc.setState(st_DESERIALIZE_PHASE_LOAD_DATA);
                                  }
                            }
                            //リトライ用の委譲項目リセット
                            delegate_child_item_now = nullptr;
                      }

                      //処理されなかったデータ項目の処理
                      for (auto& pair : *arc.m_itemList)
                      {
                            const CItemBase& child_item = pair.second;
                            //保存先の指定があるが、セーブデータになくロードできなかった項目
                            if (child_item.isOnlyOnMem())
                            {
                                  //通知フェーズに変更
                                  arc.setState(st_DESERIALIZE_PHASE_NOTICE_UNLOADED);

                                  //処理結果計上
                                  arc.getResult().addResult(child_item);

                                  //通知処理呼び出し
                                  if (is_valid_element)
```

```
                                        {
                                              noticeUnloadedItem<CIArchive, T> functor;
                                              functor(arc, item_obj.template get<T>(), ver, now_ver,
                                                                                    child_item);
                                        }
                                  }
                            }

                            //ロード後処理フェーズに変更
                            arc.setState(st_DESERIALIZE_PHASE_AFTEER_LOAD);

                            //ロード後処理（デシリアライズ専用処理）呼び出し
                            if (is_valid_element)
                            {
                                 afterLoad<CIArchive, T> functor;
                                 functor(arc, item_obj.template get<T>(), ver, now_ver);
                            }

                            //要素終了
                            arc.m_style.finishReadElem(parent_arc, arc);
                      }

                      //データ読み込み先のポインタを配列の次の要素に更新
                      if (item_obj.m_itemP)
                      {
                            *const_cast<void**>(&item_obj.m_itemP) =
                                  reinterpret_cast<T*>(const_cast<void*>(item_obj.m_itemP)) + 1;
                      }
                }

                //配列の先頭ポインタ（元のポインタ）に戻す
                item_obj.m_itemP = item_p_top;

                //配列ブロック終了
                arc.m_style.finishReadArray(parent_arc, arc);
          }
    }

    //分配前フェーズに変更
    arc.setState(st_DESERIALIZE_PHASE_BEFORE_DISTRIBUTE);

    //分配前処理（デシリアライズ専用処理）呼び出し
    {
          beforeDistribute<CIArchive, T> functor;
          functor(arc, item_obj.template get<T>(), elem_num, elem_num_loaded, ver, now_ver);
    }

    //分配フェーズに変更
    arc.setState(st_DESERIALIZE_PHASE_DISTRIBUTE);

    //分配処理（デシリアライズ専用処理）呼び出し
    while (!arc.hasFatalError())
    {
          //ブロック終了判定
          bool is_block_end = true;
          arc.m_style.tryAndReadBlockFooter(arc, item_obj, is_block_end);
          if (is_block_end)
                break;//ブロックの終了を検出したらループから抜ける

          //オブジェクトブロック開始
          {
                //オブジェクトブロック処理用の子アーカイブオブジェクトを生成
                CIArchive& parent_arc = arc;
                CIArchive arc(parent_arc, st_DESERIALIZE_PHASE_DISTRIBUTE);
```

```
                        //ブロック開始情報を仮読みし、分配処理に回す
                        CItemBase require_item;
                        std::size_t require_block_size = 0;
                        bool is_found_next_block = false;
                        arc.m_style.requireNextBlockHeader(arc, require_item, require_block_size,
                                                                                is_found_next_block);
                        if (!is_found_next_block)
                              break;//ブロックが見つからなかった場合もループから抜ける（ありえない？）

                        //オブジェクト分配処理フェーズに変更
                        arc.setState(st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT);

                        //オブジェクト処理対象データ項目をセット
                        arc.setTargetObjItem(require_item);

                        //分配処理（デシリアライズ専用処理）呼び出し
                        {
                              distributor<CIArchive, T> functor;
                              functor(arc, item_obj.template get<T>(), elem_num, elem_num_loaded, ver, now_ver,
                                                                                        require_item);
                        }

                        //オブジェクト処理対象データ項目をリセット
                        arc.resetTargetObjItem();

                        //未処理のままだったらブロックをスキップする
                        if (arc.getState() != st_DESERIALIZE_PHASE_DISTRIBUTE_OBJECT_END)
                        {
                              //オブジェクトのブロックをスキップ
                              arc.m_style.skipReadBlock(arc);

                              //処理済みにする
                              require_item.setIsAlready();
                        }

                        //オブジェクトブロック終了
                        arc.m_style.finishReadBlock(parent_arc, arc);
                  }
            }

            //分配後フェーズに変更
            arc.setState(st_DESERIALIZE_PHASE_AFTER_DISTRIBUTE);

            //分配後処理（デシリアライズ専用処理）呼び出し
            {
                  afterDistribute<CIArchive, T> functor;
                  functor(arc, item_obj.template get<T>(), elem_num, elem_num_loaded, ver, now_ver);
            }

            //ブロック終了
            arc.m_style.finishReadBlock(parent_arc, arc);
      }

      //ネストレベルが０ならターミネータを読み込み
      if (m_nestLevel == 0)
            arc.m_style.readTerminator(arc);//ターミネータ読み込み

      //処理結果にセーブデータサイズをセットする
      m_result.setSaveDataSize(getBuffUsed());

      //最終的に致命的なエラーがあれば呼び出し
      if (m_nestLevel == 0 && arc.hasFatalError())
      {
            fatalDeserializeErrorOccurred<CIArchive, T> functor;
            functor(arc, item_obj.template get<T>(), ver, now_ver);
```

```
            }

            return *this;
      }
public:
      //コンストラクタ
      CIArchive(const void* buff, const std::size_t buff_size, void* work_buff, std::size_t work_buff_size) :
            CIOArchiveBase(const_cast<void*>(buff), buff_size, work_buff, work_buff_size),
            m_style(),
            m_state(st_DESERIALIZE),
            m_parent(nullptr),
            m_targetObjItem(nullptr),
            m_targetObjItemDelegate(nullptr)
      {}
      template<typename BUFF_T, std::size_t BUFF_SIZE, typename WORK_T, std::size_t WORK_SIZE>
      CIArchive(const BUFF_T(&buff)[BUFF_SIZE], WORK_T(&work_buff)[WORK_SIZE]) :
            CIOArchiveBase(const_cast<BUFF_T*>(&buff[0]), BUFF_SIZE, work_buff, WORK_SIZE),
            m_style(),
            m_state(st_DESERIALIZE),
            m_parent(nullptr),
            m_targetObjItem(nullptr),
            m_targetObjItemDelegate(nullptr)
      {}
      template<typename WORK_T, std::size_t WORK_SIZE>
      CIArchive(const void* buff, const std::size_t buff_size, WORK_T(&work_buff)[WORK_SIZE]) :
            CIOArchiveBase(const_cast<void*>(buff), buff_size, work_buff, WORK_SIZE),
            m_parent(nullptr),
            m_style(),
            m_state(st_DESERIALIZE),
            m_targetObjItem(nullptr),
            m_targetObjItemDelegate(nullptr)
      {}
      //親を受け取るコンストラクタ
      //※処理階層が深くなるごとにコピーが行われる
      CIArchive(CIArchive& parent, const stateEnum state) :
            CIOArchiveBase(parent),
            m_parent(&parent),
            m_style(parent.m_style),
            m_state(state),
            m_targetObjItem(nullptr),
            m_targetObjItemDelegate(nullptr)
      {}
      //デストラクタ
      ~CIArchive()
      {}
private:
      //フィールド
      CIArchive* m_parent;//親アーカイブオブジェクト
      arcStyle_t m_style;//アーカイブスタイルオブジェクト
      stateEnum m_state;//ステート
      CItemBase* m_targetObjItem;//オブジェクト処理の対象データ項目
      CItemBase* m_targetObjItemDelegate;//オブジェクト処理の対象データ項目の委譲データ項目
};
```

▼ アーカイブ読み書き用ヘルパークラス

　アーカイブ形式クラスがアーカイブ読み書きクラスのセーブデータバッファを操作するためのクラス。

【アーカイブ読み書き基本操作ヘルパークラス】

```
//--------------------
```

```
//アーカイブ読み書き基本操作ヘルパークラス
class CIOArchiveHelper
{
public:
	CResult& getResult(){ return m_arc.getResult(); }//処理結果を取得
	const CResult& getResult() const { return m_arc.getResult(); }//処理結果を取得
	bool hasFatalError() const { return m_arc.hasFatalError(); }//致命的なエラーありか？
	const byte* getBuffPtr() const { return m_arc.getBuffPtr(); }//セーブデータバッファの先頭ポインタを取得
	const std::size_t getBuffSize() const { return m_arc.getBuffSize(); }//セーブデータバッファのサイズを取得
	const std::size_t getBuffUsed() const { return m_arc.getBuffUsed(); }//セーブデータバッファの使用量を取得
	const std::size_t getBuffPos() const { return m_arc.getBuffPos(); }//セーブデータバッファの現在位置を取得
	const std::size_t getBuffRemain() const { return m_arc.getBuffRemain(); }//セーブデータバッファの残量を取得
	byte* getBuffNowPtr(){ return m_arc.getBuffNowPtr(); }//セーブデータバッファの現在位置のポインタを取得
	bool buffIsFull() const { return m_arc.buffIsFull(); }//バッファの現在位置が末端に到達したか？
	std::size_t getItemListNum() const { return m_arc.getItemListNum(); }//データ項目リスト
public:
	//メソッド
	//処理結果を合成
	void addResult(const CResult& src){ m_arc.addResult(src); }
	//リストからデータ項目を検索
	const CItemBase* findItem(const crc32_t name_crc) const { return m_arc.findItem(name_crc); }
	//バッファのカレントポインタを移動
	//※範囲外への移動が要求されたら端まで移動して false を返す
	bool seek(const int seek_, int& real_seek)
	{
		const int used = static_cast<int>(m_arc.m_buffPos);
		const int remain = static_cast<int>(m_arc.getBuffRemain());
		real_seek = seek_ < 0 ?
			used >(-seek_) ?
		seek_ :
			-used :
			remain > seek_ ?
		  seek_ :
				remain;
		m_arc.m_buffPos = static_cast<std::size_t>(used + real_seek);
		return real_seek == seek_;
	}
	//※処理結果オブジェクト使用版
	bool seek(CResult& result, const int seek_)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		int real_seek = 0;
		//カレントポインタ移動
		const bool result_now = seek(seek_, real_seek);
		//処理結果記録
		if (!result_now)
			result.setHasFatalError();
		return result_now;
	}
protected:
	//コンストラクタ
	CIOArchiveHelper(CIOArchiveBase& arc) :
		m_arc(arc)
	{}
	//デストラクタ
	~CIOArchiveHelper()
	{}
protected:
	//フィールド
	CIOArchiveBase& m_arc;//アーカイブオブジェクト（セーブデータイメージバッファ）
};
```

【アーカイブ書き込み基本操作ヘルパークラス】

```
//--------------------
```

```
//アーカイブ書き込み基本操作ヘルパークラス
class COArchiveHelper : public CIOArchiveHelper
{
	template<class STYLE>
	friend class COArchive;
public:
	//メソッド
	//バッファへのデータ書き込み
	//※要求サイズが全て書き込めなかったら false を返す
	bool write(const void* data, const std::size_t size, std::size_t& written_size)
	{
		const std::size_t remain = m_arc.getBuffRemain();
		written_size = remain > size ? size : remain;
		if (data)//データがヌルならサイズ分 0 で埋める
			memcpy(m_arc.m_buff + m_arc.m_buffPos, data, written_size);
		else//データコピー
			memset(m_arc.m_buff + m_arc.m_buffPos, 0, written_size);
		m_arc.m_buffPos += written_size;
		return written_size == size;
	}
	//※処理結果オブジェクト使用版
	bool write(CResult& result, const void* data, const std::size_t size, std::size_t* written_size = nullptr)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//データ書き込み
		std::size_t written_size_tmp = 0;
		const bool result_now = write(data, size, written_size_tmp);
		//処理結果記録
		result.addCopiedSize(written_size_tmp);
		if (!result_now)
			result.setHasFatalError();
		if (written_size)
			*written_size += written_size_tmp;
		return result_now;
	}
	//バッファへの文字列書き込み
	std::size_t print(CResult& result, const char* fmt, ...)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		const std::size_t remain = m_arc.getBuffRemain();
		va_list args;
		va_start(args, fmt);
	#ifdef USE_STRCPY_S
		const std::size_t ret = vsprintf_s(reinterpret_cast<char*>(m_arc.m_buff + m_arc.m_buffPos), remain, fmt,
	                                                                                                    args);
	#else//USE_STRCPY_S
		const std::size_t ret = vsprintf(reinterpret_cast<char*>(m_arc.m_buff + m_arc.m_buffPos), fmt, args);
	#endif//USE_STRCPY_S
		va_end(args);
		m_arc.m_buffPos += ret;
		return ret;
	}
	//バッファへの文字列書き込み
	std::size_t printWithFunc(CResult& result, toStrFuncP to_str_func, const void* data_p, const std::size_t data_size)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		const std::size_t remain = m_arc.getBuffRemain();
		const std::size_t ret = to_str_func(reinterpret_cast<char*>(m_arc.m_buff + m_arc.m_buffPos), remain, data_p,
	                                                                                                    data_size);
		m_arc.m_buffPos += ret;
		return ret;
	}
```

```
	//バッファへのインデント書き込み
	void printIndent(CResult& result, const int add)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return;
		const int remain = static_cast<int>(m_arc.getBuffRemain());
		int space_len = (static_cast<int>(m_arc.m_nestLevel) + add) * 2 + 1;
		space_len = space_len > 0 ? space_len : 0;
		space_len = space_len < remain ? space_len : remain;
		if (space_len >= 1)
		{
			memset(m_arc.m_buff + m_arc.m_buffPos, ' ', space_len);
			m_arc.m_buffPos += space_len;
			*(m_arc.m_buff + m_arc.m_buffPos) = '¥0';
		}
	}
public:
	//コンストラクタ
	COArchiveHelper(CIOArchiveBase& arc) :
		CIOArchiveHelper(arc)
	{}
	//デストラクタ
	~COArchiveHelper()
	{}
};
```

【アーカイブ読み込み基本操作ヘルパークラス】

```
//--------------------
//アーカイブ読み込み基本操作ヘルパークラス
class CIArchiveHelper : public CIOArchiveHelper
{
	template<class STYLE>
	friend class CIArchive;
public:
	//メソッド
	//バッファからのデータ読み込み
	//※要求サイズが全て書き込めなかったら false を返す
	bool read(void* data, const std::size_t size, std::size_t& read_size)
	{
		const std::size_t remain = m_arc.getBuffRemain();
		read_size = remain > size ? size : remain;
		if (data)//data がヌルならコピーしないがポインタは進める
			memcpy(data, m_arc.m_buff + m_arc.m_buffPos, read_size);
		m_arc.m_buffPos += read_size;
		return read_size == size;
	}
	//※処理結果オブジェクト使用版
	bool read(CResult& result, void* data, const std::size_t size, std::size_t* read_size = nullptr)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//データ読み込み
		std::size_t read_size_tmp = 0;
		const bool result_now = read(data, size, read_size_tmp);
		//処理結果記録
		if (!result_now)
			result.setHasFatalError();
		if (read_size)
			*read_size += read_size_tmp;
		return result_now;
	}
	//バッファからのデータ読み込み
	//※要求サイズが全て書き込めなかったら false を返す
	bool readWithFunc(fromMemFuncP from_mem_func, void* data, const std::size_t dst_size, const std::size_t src_size,
	                                                                                      std::size_t& read_size)
```

```
	{
		const std::size_t remain = m_arc.getBuffRemain();
		read_size = remain > src_size ? src_size : remain;
		if (data)//data がヌルならコピーしないがポインタは進める
		{
			from_mem_func(m_arc.m_buff + m_arc.m_buffPos, read_size, data, dst_size);
		}
		m_arc.m_buffPos += read_size;
		return read_size == src_size;
	}
	//※処理結果オブジェクト使用版
	bool readWithFunc(CResult& result, fromMemFuncP from_mem_func, void* data, const std::size_t dst_size,
	                                                   const std::size_t src_size, std::size_t* read_size = nullptr)
	{
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//データ読み込み
		std::size_t read_size_tmp = 0;
		const bool result_now = readWithFunc(from_mem_func, data, dst_size, src_size, read_size_tmp);
		//処理結果記録
		if (!result_now)
			result.setHasFatalError();
		if (read_size)
			*read_size += read_size_tmp;
		return result_now;
	}
public:
	//コンストラクタ
	CIArchiveHelper(CIOArchiveBase& arc) :
		CIOArchiveHelper(arc)
	{}
	//デストラクタ
	~CIArchiveHelper()
	{}
};
```

▼ アーカイブ形式クラス

【アーカイブ形式基底クラス】

```
//--------------------
//アーカイブ形式基底クラス
class CArchiveStyleBase
{
public:
	//コンストラクタ
	CArchiveStyleBase()
	{}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	CArchiveStyleBase(CArchiveStyleBase& parent)
	{}
	//デストラクタ
	~CArchiveStyleBase()
	{}
};
```

【アーカイブ形式クラスプロトタイプ（アーカイブ書き込み用）】

```
//--------------------
//アーカイブ形式プロトタイプ（アーカイブ書き込み用）
//※必要なメソッドを定義しているだけのサンプル（クラス追加のための参考用）
//※実際には使用しない
class COArchiveStyleProto : public CArchiveStyleBase
```

```
{
public:
	//メソッド
	//シグネチャ書き込み
	bool writeSignature(COArchiveHelper arc){ return true; }
	//ブロックヘッダー書き込み
	bool writeBlockHeader(COArchiveHelper arc, const CItemBase& item, const CVersion& ver){ return true; }
	//配列ブロックヘッダー書き込み
	bool writeArrayHeader(COArchiveHelper arc, const CItemBase& item, const std::size_t array_elem_num){ return true; }
	//要素ヘッダー書き込み
	bool writeElemHeader(COArchiveHelper arc, const CItemBase& item, const std::size_t index){ return true; }
	//データ項目書き込み
	bool writeDataItem(COArchiveHelper arc, const CItemBase& item, const CItemBase& child_item){ return true; }
	//要素フッター書き込み
	bool writeElemFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
		const std::size_t index, short& items_num, std::size_t& elem_size){ return true; }
	//配列ブロックフッター書き込み
	bool writeArrayFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
		std::size_t& array_block_size){ return true; }
	//ブロックフッター書き込み
	bool writeBlockFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
		std::size_t& block_size){ return true; }
	//ターミネータ書き込み
	bool writeTerminator(COArchiveHelper arc){ return true; }
public:
	//コンストラクタ
	COArchiveStyleProto() :
		CArchiveStyleBase()
	{}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	COArchiveStyleProto(COArchiveStyleProto& parent) :
		CArchiveStyleBase(parent)
	{}
	//デストラクタ
	~COArchiveStyleProto()
	{}
};
```

【アーカイブ形式プロトタイプ（アーカイブ読み込み用）】

```
//--------------------
//アーカイブ形式プロトタイプ（アーカイブ読み込み用）
//※必要なメソッドを定義しているだけのサンプル（クラス追加のための参考用）
//※実際には使用しない
class CIArchiveStyleProto : public CArchiveStyleBase
{
public:
	//メソッド
	//パース
	bool parse(CIArchiveHelper arc){ return true; }
	//シグネチャ読み込み
	bool readSignature(CIArchiveHelper arc){ return true; }
	//ブロックヘッダー読み込み
	//※読み込んだオブジェクトの型情報とバージョンを返す
	bool readBlockHeader(CIArchiveHelper arc, const CItemBase& item, const CItemBase* delegate_item,
		const CVersion& ver, CItemBase& input_item, CVersion& input_ver, std::size_t& block_size){ return true; }
	//配列ブロックヘッダー読み込み
	bool readArrayHeader(CIArchiveHelper arc, const CItemBase& item, std::size_t& array_elem_num,
		std::size_t& array_block_size){ return true; }
	//要素ヘッダー読み込み
	bool readElemHeader(CIArchiveHelper arc, const CItemBase& item, const std::size_t index,
		short& items_num, std::size_t& elem_size){ return true; }
	//データ項目読み込み
	bool readDataItem(CIArchiveHelper arc, const CItemBase& item, const CItemBase* delegate_child_item_now,
		CItemBase& child_item, const bool item_is_valid, const bool is_required_retry){ return true; }
```

```
	//要素フッター読み込み
	//※読み込みテストの結果、要素フッターでなければデータ項目の読み込みを継続する
	bool tryAndReadElemFooter(CIArchiveHelper child_arc, const CItemBase& item, const std::size_t index,
	                                                                                bool& is_elem_end){ return true; }
	//要素読み込み終了
	bool finishReadElem(CIArchiveHelper parent_arc, CIArchiveHelper child_arc){ return true; }
	//配列ブロックフッター読み込み
	//※読み込みテストの結果、配列ブロックフッターでなければデータ項目の読み込みを継続する
	bool tryAndReadArrayFooter(CIArchiveHelper arc, const CItemBase& item, bool& is_array_block_end){ return true; }
	//配列ブロック読み込み終了
	bool finishReadArray(CIArchiveHelper parent_arc, CIArchiveHelper child_arc){ return true; }
	//ブロックの読み込みをスキップ
	bool skipReadBlock(CIArchiveHelper arc){ return true; }
	//ブロックフッター読み込み
	//※読み込みテストの結果、ブロックフッターでなければデータ項目（オブジェクト）の読み込みを継続する
	bool tryAndReadBlockFooter(CIArchiveHelper arc, const CItemBase& item, bool& is_block_end){ return true; }
	//ブロック読み込み終了
	bool finishReadBlock(CIArchiveHelper parent_arc, CIArchiveHelper child_arc){ return true; }
	//次のブロックヘッダー問い合わせ（先行読み込み）
	//※処理を進めず、次の情報を読み取るのみ
	//※（例えば、バイナリスタイルなら、読み込みバッファのポインタを進めない）
	bool requireNextBlockHeader(CIArchiveHelper arc, CItemBase& require_item, std::size_t& child_block_size,
	                                                                        bool& is_found_next_block){ return true; }
	//ターミネータ読み込み
	bool readTerminator(CIArchiveHelper arc){ return true; }
public:
	//コンストラクタ
	CIArchiveStyleProto() :
		CArchiveStyleBase()
	{}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	CIArchiveStyleProto(CIArchiveStyleProto& parent) :
		CArchiveStyleBase(parent)
	{}
	//デストラクタ
	~CIArchiveStyleProto()
	{}
};
```

▼ アーカイブ形式クラス：バイナリ形式

【バイナリ形式アーカイブクラス（共通）】

```
//--------------------
//バイナリ形式アーカイブクラス（共通）
class CArchiveStyleBinaryBase : public CArchiveStyleBase
{
public:
	//定数
	static const std::size_t SIGNATURE_SIZE = 16;//シグネチャサイズ
	static const std::size_t TERMINATOR_SIZE = 16;//ターミネータサイズ
	static const std::size_t BEGIN_MARK_SIZE = 2;//各種始端マークサイズ
	static const std::size_t END_MARK_SIZE = 2;//各種終端マークサイズ
	static const unsigned char SIGNATURE[SIGNATURE_SIZE];//シグネチャ
	static const unsigned char TERMINATOR[TERMINATOR_SIZE];//ターミネータ
	static const unsigned char BLOCK_BEGIN[BEGIN_MARK_SIZE];//ブロック始端
	static const unsigned char BLOCK_END[END_MARK_SIZE];//ブロック終端
	static const unsigned char ARRAY_BEGIN[BEGIN_MARK_SIZE];//配列始端
	static const unsigned char ARRAY_END[END_MARK_SIZE];//配列終端
	static const unsigned char ELEM_BEGIN[BEGIN_MARK_SIZE];//要素始端
	static const unsigned char ELEM_END[END_MARK_SIZE];//要素終端
	static const unsigned char ITEM_BEGIN[BEGIN_MARK_SIZE];//データ項目始端
```

```
	static const unsigned char ITEM_END[END_MARK_SIZE];//データ項目終端
public:
	//コンストラクタ
	CArchiveStyleBinaryBase():
		CArchiveStyleBase()
	{}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	CArchiveStyleBinaryBase(CArchiveStyleBinaryBase& parent) :
		CArchiveStyleBase(parent)
	{}
	//デストラクタ
	~CArchiveStyleBinaryBase()
	{}
};
```

【バイナリ形式アーカイブクラス（共通）の静的変数をインスタンス化】

```
//--------------------
//バイナリ形式アーカイブクラス（共通）：静的変数インスタンス化
const unsigned char CArchiveStyleBinaryBase::SIGNATURE[CArchiveStyleBinaryBase::SIGNATURE_SIZE] =
		{ 0x00, 0xff, 's', 'e', 'r', 'i', 'a', 'l', ':', ':', 'C', 'B', 'i', 'n', 0xff, 0x00 };//シグネチャ
const unsigned char CArchiveStyleBinaryBase::TERMINATOR[CArchiveStyleBinaryBase::TERMINATOR_SIZE] =
		{ 0xff, 0x00, 's', 'e', 'r', 'i', 'a', 'l', ':', ':', 'C', 'B', 'i', 'n', 0x00, 0xff };//ターミネータ
const unsigned char CArchiveStyleBinaryBase::BLOCK_BEGIN[CArchiveStyleBinaryBase::BEGIN_MARK_SIZE] = { 0xfb, 0xff };
                                                                                    //ブロック始端
const unsigned char CArchiveStyleBinaryBase::BLOCK_END[CArchiveStyleBinaryBase::END_MARK_SIZE] = { 0xff, 0xfb };
                                                                                    //ブロック終端
const unsigned char CArchiveStyleBinaryBase::ARRAY_BEGIN[CArchiveStyleBinaryBase::BEGIN_MARK_SIZE] = { 0xfa, 0xff };
                                                                                    //配列始端
const unsigned char CArchiveStyleBinaryBase::ARRAY_END[CArchiveStyleBinaryBase::END_MARK_SIZE] = { 0xff, 0xfa };
                                                                                    //配列終端
const unsigned char CArchiveStyleBinaryBase::ELEM_BEGIN[CArchiveStyleBinaryBase::BEGIN_MARK_SIZE] = { 0xfe, 0xff };
                                                                                    //要素始端
const unsigned char CArchiveStyleBinaryBase::ELEM_END[CArchiveStyleBinaryBase::END_MARK_SIZE] = { 0xff, 0xfe };
                                                                                    //要素終端
const unsigned char CArchiveStyleBinaryBase::ITEM_BEGIN[CArchiveStyleBinaryBase::BEGIN_MARK_SIZE] = { 0xfd, 0xff };
                                                                                    //データ項目始端
const unsigned char CArchiveStyleBinaryBase::ITEM_END[CArchiveStyleBinaryBase::END_MARK_SIZE] = { 0xff, 0xfd };
                                                                                    //データ項目終端
```

【バイナリ形式アーカイブクラス（アーカイブ書き込み用)】

```
//--------------------
//バイナリ形式アーカイブクラス（アーカイブ書き込み用）
class COArchiveStyleBinary : public CArchiveStyleBinaryBase
{
public:
	//メソッド
	//シグネチャ書き込み
	bool writeSignature(COArchiveHelper arc)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		arc.write(result, SIGNATURE, SIGNATURE_SIZE);//シグネチャ書き込み
		return !result.hasFatalError();
	}
	//ブロックヘッダー書き込み
	bool writeBlockHeader(COArchiveHelper arc, const CItemBase& item, const CVersion& ver)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		arc.write(result, BLOCK_BEGIN, BEGIN_MARK_SIZE);//ブロック始端書き込み
		arc.write(result, &item.m_nameCrc, sizeof(item.m_nameCrc));//名前 CRC 書き込み
		item.m_attr.setHasVer();//バージョン情報ありにする
```

```
	arc.write(result, &item.m_attr.m_value, sizeof(item.m_attr.m_value));//属性書き込み
	arc.write(result, ver.getVerPtr(), ver.getVerSize());//バージョン書き込み
	const std::size_t block_size = 0;
	arc.write(result, &block_size, sizeof(block_size));//ブロックサイズ仮書き込み※ブロック終了時に書き換える
	return !result.hasFatalError();
}
//配列ブロックヘッダー書き込み
bool writeArrayHeader(COArchiveHelper arc, const CItemBase& item, const std::size_t array_elem_num)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	if (item.isArr())
	{
		arc.write(result, ARRAY_BEGIN, BEGIN_MARK_SIZE);//配列ブロック始端書き込み
		const std::size_t array_block_size = 0;
		arc.write(result, &array_elem_num, sizeof(array_elem_num));//配列要素数仮書き込み
		arc.write(result, &array_block_size, sizeof(array_block_size));
                                                //配ブロック列サイズ仮書き込み　※配列終了時に書き換える
	}
	return !result.hasFatalError();
}
//要素ヘッダー書き込み
bool writeElemHeader(COArchiveHelper arc, const CItemBase& item, const std::size_t index)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	arc.write(result, ELEM_BEGIN, BEGIN_MARK_SIZE);//要素始端書き込み
	const short items_num = 0;
	const std::size_t elem_size = 0;
	arc.write(result, &items_num, sizeof(items_num));//データ項目数仮書き込み　※要素終了時に書き換える
	arc.write(result, &elem_size, sizeof(elem_size));//要素サイズ仮書き込み　※要素終了時に書き換える
	return !result.hasFatalError();
}
//データ項目書き込み
bool writeDataItem(COArchiveHelper arc, const CItemBase& item, const CItemBase& child_item)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	arc.write(result, ITEM_BEGIN, BEGIN_MARK_SIZE);//データ項目始端書き込み
	arc.write(result, &child_item.m_nameCrc, sizeof(child_item.m_nameCrc));//名前 CRC 書き込み
	child_item.m_attr.resetHasVer();//バージョン情報なしにする
	arc.write(result, &child_item.m_attr.m_value, sizeof(child_item.m_attr.m_value));//属性書き込み
	arc.write(result, &child_item.m_itemSize, sizeof(child_item.m_itemSize));//データサイズ書き込み
	if (!child_item.isNul())//ヌル時はここまでの情報で終わり
	{
		if (child_item.isArr())//配列か？
			arc.write(result, &child_item.m_arrNum, sizeof(child_item.m_arrNum));//配列要素数書き込み
		unsigned char* p = reinterpret_cast<unsigned char*>(const_cast<void*>(child_item.m_itemP));
		const std::size_t elem_num = child_item.getElemNum();
		for (std::size_t index = 0; index < elem_num && !result.hasFatalError(); ++index)
                                                                        //配列要素数分データ書き込み
		{
			arc.write(result, p, child_item.m_itemSize);//データ書き込み
			p += child_item.m_itemSize;
		}
	}
	arc.write(result, ITEM_END, END_MARK_SIZE);//データ項目終端書き込み
	return !result.hasFatalError();
}
//要素フッター書き込み
bool writeElemFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
                                        const std::size_t index, short& items_num, std::size_t& elem_size)
```

```
{
	CResult& result = child_arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
	{
		parent_arc.addResult(result);//親に処理結果を計上
		return false;
	}
	items_num = child_arc.getItemListNum();//データ項目数取得
	elem_size = child_arc.getBuffUsed();//データサイズ取得
	parent_arc.seek(result, -static_cast<int>(sizeof(elem_size)));
                                                    //要素サイズ情報の分､バッファのカレントポインタを戻す
	parent_arc.seek(result, -static_cast<int>(sizeof(items_num)));
                                                  //データ項目数情報の分、バッファのカレントポインタを戻す
	parent_arc.write(result, &items_num, sizeof(items_num));//要素サイズを更新（書き込み）
	parent_arc.write(result, &elem_size, sizeof(elem_size));//要素サイズを更新（書き込み）
	parent_arc.seek(result, static_cast<int>(elem_size));
                                        //要素サイズ分（要素の終わりまで)、バッファのカレントポインタを進める
	parent_arc.write(result, ELEM_END, END_MARK_SIZE);//要素終端書き込み
	parent_arc.addResult(result);//親に処理結果を計上
	return !result.hasFatalError();
}
//配列ブロックフッター書き込み
bool writeArrayFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
                                                                    std::size_t& array_block_size)
{
	CResult& result = child_arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
	{
		parent_arc.addResult(result);//親に処理結果を計上
		return false;
	}
	array_block_size = child_arc.getBuffUsed();//データサイズ取得
	if (item.isArr())
	{
		parent_arc.seek(result, -static_cast<int>(sizeof(array_block_size)));
                                        //配列ブロックサイズ情報の分、バッファのカレントポインタを戻す
		parent_arc.write(result, &array_block_size, sizeof(array_block_size));
                                                                    //配列ブロックサイズを更新（書き込み）
	}
	parent_arc.seek(result, static_cast<int>(array_block_size));
                                //配列ブロックサイズ分（要素の終わりまで)、バッファのカレントポインタを進める
	if (item.isArr())
	{
		parent_arc.write(result, ARRAY_END, END_MARK_SIZE);//配列ブロック終端書き込み
	}
	parent_arc.addResult(result);//親に処理結果を計上
	return !result.hasFatalError();
}
//ブロックフッター書き込み
bool writeBlockFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
                                                                    std::size_t& block_size)
{
	CResult& result = child_arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
	{
		parent_arc.addResult(result);//親に処理結果を計上
		return false;
	}
	block_size = child_arc.getBuffUsed();//データサイズ取得
	int real_seek = 0;
	parent_arc.seek(result, -static_cast<int>(sizeof(block_size)));
                                                  //ブロックサイズ情報の分、バッファのカレントポインタを戻す
	parent_arc.write(result, &block_size, sizeof(block_size));//ブロックサイズを更新（書き込み）
	parent_arc.seek(result, static_cast<int>(block_size));
                                //ブロックサイズ分（ブロックの終わりまで)、バッファのカレントポインタを進める
```

```
            parent_arc.write(result, BLOCK_END, END_MARK_SIZE);//ブロック終端書き込み
            parent_arc.addResult(result);//親に処理結果を計上
            return !result.hasFatalError();
      }
      //ターミネータ書き込み
      bool writeTerminator(COArchiveHelper arc)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            arc.write(result, TERMINATOR, TERMINATOR_SIZE);//ターミネータ書き込み
            return !result.hasFatalError();
      }
public:
      //コンストラクタ
      COArchiveStyleBinary() :
            CArchiveStyleBinaryBase()
      {}
      //親を受け取るコンストラクタ
      //※処理階層が深くなるごとにコピーが行われる
      COArchiveStyleBinary(COArchiveStyleBinary& parent) :
            CArchiveStyleBinaryBase(parent)
      {}
      //デストラクタ
      ~COArchiveStyleBinary()
      {}
};
```

【バイナリ形式アーカイブクラス（アーカイブ読み込みよう）】

```
//--------------------
//バイナリ形式アーカイブクラス（アーカイブ読み込み用）
class CIArchiveStyleBinary : public CArchiveStyleBinaryBase
{
public:
      //メソッド
      //パース
      bool parse(CIArchiveHelper arc)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            //何もしない
            return !result.hasFatalError();
      }
      //シグネチャ読み込み
      bool readSignature(CIArchiveHelper arc)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            char signature[SIGNATURE_SIZE];//シグネチャ読み込みバッファ
            arc.read(result, signature, SIGNATURE_SIZE);//シグネチャ読み込み
            if (memcmp(signature, SIGNATURE, SIGNATURE_SIZE) != 0)//シグネチャチェック
                  result.setHasFatalError();
            return !result.hasFatalError();
      }
      //ブロックヘッダー読み込み
      //※読み込んだオブジェクトの型情報とバージョンを返す
      bool readBlockHeader(CIArchiveHelper arc, const CItemBase& item, const CItemBase* delegate_item,
                           const CVersion& ve, CItemBase& input_item, CVersion& input_ver, std::size_t& block_size)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            char begin_mark[BEGIN_MARK_SIZE];//ブロック始端読み込みバッファ
```

```
		arc.read(result, begin_mark, BEGIN_MARK_SIZE);//ブロック始端読み込み
		if (memcmp(begin_mark, BLOCK_BEGIN, BEGIN_MARK_SIZE) != 0)//ブロック始端チェック
		{
			result.setHasFatalError();
			return false;
		}
		input_item.clearForLoad();//読み込み情報を一旦クリア
		arc.read(result, const_cast<crc32_t*>(&input_item.m_nameCrc), sizeof(input_item.m_nameCrc));
                                                                                //名前CRC読み込み
		arc.read(result, const_cast<CItemAttr::value_t*>(&input_item.m_attr.m_value),
                                                                sizeof(input_item.m_attr.m_value));//属性読み込み
		if (input_item.m_attr.hasVer())//バージョン情報があるか？
		{
			arc.read(result, const_cast<unsigned int*>(input_ver.getVerPtr()), input_ver.getVerSize());
                                                                                //バージョン読み込み
			input_ver.calcFromVer();
		}
		arc.read(result, const_cast<std::size_t*>(&input_item.m_itemSize), sizeof(input_item.m_itemSize));
                                                                                //ブロックサイズ読み込み
		block_size = input_item.m_itemSize;//ブロックサイズ
		//委譲アイテムの指定があれば、名前をチェックせずに入力データに情報をコピーする
		if (delegate_item)
		{
			input_item.copyFromOnMem(*delegate_item);//セーブデータの情報に現在の情報をコピー（統合）
			input_item.setIsOnlyOnSaveData();//セーブデータ上のみのデータ扱いにする（集計のため）
		}
		else
		{
			//名前CRCをチェックして情報を統合
			//※違っていたら致命的エラー（セーブデータが適合していない）
			if (input_item == item)
			{
				//両者に存在する
				input_item.copyFromOnMem(item);//セーブデータの情報に現在の情報をコピー（統合）
			}
			else
			{
				//一致しない
				input_item.setIsOnlyOnSaveData();//セーブデータにしか存在しないデータ項目
				result.setHasFatalError();//致命的エラー設定
			}
		}
		//ヌルの時はこの時点で処理済みにする
		//※ヌルじゃない時は配列読み込みが済んだら処理済みにする
		if (input_item.m_attr.isNul())
		{
			item.setIsAlready();//処理済みにする
			result.addResult(input_item);//結果を計上
		}
		return !result.hasFatalError();
	}
	//配列ブロックヘッダー読み込み
	bool readArrayHeader(CIArchiveHelper arc, const CItemBase& item, std::size_t& array_elem_num,
                                                                        std::size_t& array_block_size)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		if (item.isArr())//配列型の時だけ読み込む
		{
			char begin_mark[BEGIN_MARK_SIZE];//配列ブロック始端読み込みバッファ
			arc.read(result, begin_mark, BEGIN_MARK_SIZE);//配列ブロック始端読み込み
			if (memcmp(begin_mark, ARRAY_BEGIN, BEGIN_MARK_SIZE) != 0)//配列ブロック始端チェック
			{
				result.setHasFatalError();
```

```
                    return false;
              }
              arc.read(result, const_cast<std::size_t*>(&item.m_arrNum), sizeof(item.m_arrNum));
                                                                              //配列要素数読み込み
              arc.read(result, const_cast<std::size_t*>(&array_block_size), sizeof(array_block_size));
                                                                          //配列ブロックサイズ読み込み
              array_elem_num = item.m_arrNum;//配列要素数
        }
        else
        {
              *const_cast<std::size_t*>(&item.m_arrNum) = 0;//配列要素数
              array_elem_num = 0;//配列要素数
              array_block_size = 0;//配列ブロックサイズ
        }
        item.setIsAlready();//処理済みにする
        result.addResult(item);//結果を計上
        return !result.hasFatalError();
  }
  //要素ヘッダー読み込み
  bool readElemHeader(CIArchiveHelper arc, const CItemBase& item, const std::size_t index, short& items_num,
                                                                                  std::size_t& elem_size)
  {
        CResult& result = arc.getResult();
        items_num = 0;
        if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
              return false;
        char begin_mark[BEGIN_MARK_SIZE];//配列始端読み込みバッファ
        arc.read(result, begin_mark, BEGIN_MARK_SIZE);//配列始端読み込み
        if (memcmp(begin_mark, ELEM_BEGIN, BEGIN_MARK_SIZE) != 0)//要素始端チェック
        {
              result.setHasFatalError();
              return false;
        }
        arc.read(result, &items_num, sizeof(items_num));//データ項目数読み込み
        arc.read(result, &elem_size, sizeof(elem_size));//要素サイズ読み込み
        return !result.hasFatalError();
  }
  //データ項目読み込み
  bool readDataItem(CIArchiveHelper arc, const CItemBase& item, const CItemBase* delegate_child_item_now,
                              CItemBase& child_item, const bool item_is_valid, const bool is_required_retry)
  {
        CResult& result = arc.getResult();
        if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
              return false;
        if (is_required_retry)//リトライ要求
        {
              //記憶している位置に戻す
              arc.seek(result, -static_cast<int>(m_readSizeForPrevDataItem));
                                                                      //【リトライ用】前回のデータ読み込みサイズ
        }
        m_readSizeForPrevDataItem = 0;//【リトライ用】前回のデータ読み込みサイズをリセット
        std::size_t read_size = 0;//この処理中で計上する読み込みサイズ
        char begin_mark[BEGIN_MARK_SIZE];//データ項目／ブロック始端読み込みバッファ
        arc.read(result, begin_mark, BEGIN_MARK_SIZE, &read_size);//データ項目／ブロック始端読み込み
        bool is_block_begin = false;
        if (memcmp(begin_mark, BLOCK_BEGIN, BEGIN_MARK_SIZE) == 0)//ブロック始端チェック
              is_block_begin = true;
        else
        {
              if (memcmp(begin_mark, ITEM_BEGIN, BEGIN_MARK_SIZE) != 0)//データ項目始端チェック
              {
                    result.setHasFatalError();
                    return false;
              }
        }
```

```
		arc.read(result, const_cast<crc32_t*>(&child_item.m_nameCrc), sizeof(child_item.m_nameCrc), &read_size);
                                                                                    //名前 CRC 読み込み
		arc.read(result, const_cast<CItemAttr::value_t*>(&child_item.m_attr.m_value),
                                                sizeof(child_item.m_attr.m_value), &read_size);//属性読み込み
		const CItemBase* child_item_now = nullptr;
		if (delegate_child_item_now)//委譲データ項目があればそれを優先的に使用
		{
			child_item_now = delegate_child_item_now;//委譲データ項目
			//child_item の名前と CRC を書き換える
			child_item.m_name = delegate_child_item_now->m_name;//名前
			*const_cast<crc32_t*>(&child_item.m_nameCrc) = delegate_child_item_now->m_nameCrc;//名前の CRC
		}
		else
			child_item_now = arc.findItem(child_item.m_nameCrc);//対応するデータ項目情報を検索
		if (child_item_now)//対応するデータ項目が見つかったか？
			child_item.copyFromOnMem(*child_item_now);//現在の情報をセーブデータの情報にコピー（統合）
		else
			child_item.setIsOnlyOnSaveData();//対応するデータがない：セーブデータにしかデータが存在しない
		//対象データ項目はオブジェクト型か？
		if (child_item.isObj())
		{
			//対象がオブジェクト（ブロック）なら、オブジェクトとして処理をやり直すために、この時点で処理終了
			arc.seek(result, -static_cast<int>(read_size));
                                                        //やり直しのために、バッファのカレントポインタを先頭に戻す
			//ブロック始端を検出していなかったらデータ不整合でエラー終了
			if (!is_block_begin)
			{
				result.setHasFatalError();
				return false;
			}
			return !result.hasFatalError();
		}
		//通常データの読み込み処理
		assert(!child_item.m_attr.hasVer());//オブジェクトでもないのにバージョン情報があれば NG
		arc.read(result, const_cast<std::size_t*>(&child_item.m_itemSize), sizeof(child_item.m_itemSize),
                                                                                &read_size);//データサイズ読み込み
		if (!child_item.isNul())//【セーブデータ上の】データがヌルでなければ処理する
		{
			if (child_item.isArr())//配列か？
				arc.read(result, const_cast<std::size_t*>(&child_item.m_arrNum),
                                                sizeof(child_item.m_arrNum), &read_size);//配列要素数読み込み
			unsigned char* p = reinterpret_cast<unsigned char*>(const_cast<void*>(child_item.m_itemP));
			const std::size_t elem_num = child_item.getElemNum();
			for (std::size_t index = 0; index < elem_num && !result.hasFatalError(); ++index)
                                                                //【セーブデータ上の】配列要素数分データ書き込み
			{
				const bool element_is_valid =//有効なデータか？
					item_is_valid && //親のデータが有効か？
					!child_item.isOnlyOnSaveData() && //セーブデータにしかないデータではないか？
					!child_item.nowIsNul() && //現在の（コピー先の）データがヌルではないか？
					index < child_item.getNowElemNum();//現在の（コピー先の）配列の範囲内か？
				void* p_tmp = element_is_valid ? p : nullptr;
                                                        //有効なデータでなければ nullptr を渡し、空読み込みする
				arc.readWithFunc(result, child_item.m_nowTypeCtrl.m_fromMemFuncP, p_tmp,
                                        child_item.m_nowItemSize, child_item.m_itemSize, &read_size);//データ読み込み
				if (p)
					p += child_item.m_nowItemSize;//次の要素
			}
		}
		char end_mark[END_MARK_SIZE];//データ項目終端読み込みバッファ
		arc.read(result, end_mark, END_MARK_SIZE, &read_size);//データ項目終端読み込み
		if (memcmp(end_mark, ITEM_END, END_MARK_SIZE) != 0)//データ項目始端チェック
		{
			result.setHasFatalError();
			return false;
```

```
		}
		child_item.setIsAlready();//処理済みにする
		result.addResult(child_item);//結果を計上
		m_readSizeForPrevDataItem = read_size;//【リトライ用】前回のデータ読み込みサイズを記憶
		return true;
	}
	//要素フッター読み込み
	//※読み込みテストの結果、要素フッターでなければデータ項目の読み込みを継続する
	bool tryAndReadElemFooter(CIArchiveHelper arc, const CItemBase& item, const std::size_t index, bool& is_elem_end)
	{
		is_elem_end = true;//要素終了扱い
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		std::size_t read_size = 0;
		char end_mark[END_MARK_SIZE];
		arc.read(result, end_mark, END_MARK_SIZE, &read_size);//要素終端読み込み
		if (memcmp(end_mark, ELEM_END, END_MARK_SIZE) != 0)//要素終端チェック
		{
			is_elem_end = false;//要素終了ではない
			arc.seek(result, -static_cast<int>(read_size));
												//要素の終端ではなかったため、バッファのカレントポインタを戻す
		}
		return !result.hasFatalError();
	}
	//要素読み込み終了
	bool finishReadElem(CIArchiveHelper parent_arc, CIArchiveHelper child_arc)
	{
		parent_arc.addResult(child_arc.getResult());//親に処理結果を計上
		CResult& result = child_arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		parent_arc.seek(result, child_arc.getBuffUsed());
		return !result.hasFatalError();
	}
	//配列ブロックフッター読み込み
	//※読み込みテストの結果、配列ブロックフッターでなければデータ項目の読み込みを継続する
	bool tryAndReadArrayFooter(CIArchiveHelper arc, const CItemBase& item, bool& is_array_block_end)
	{
		is_array_block_end = true;//配列ブロック終了扱い
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		if (item.isArr())
		{
			std::size_t read_size = 0;
			char end_mark[END_MARK_SIZE];
			arc.read(result, end_mark, END_MARK_SIZE, &read_size);//配列ブロック終端読み込み
			if (memcmp(end_mark, ARRAY_END, END_MARK_SIZE) != 0)//要素終端チェック
			{
				is_array_block_end = false;//配列ブロック終了ではない
				arc.seek(result, -static_cast<int>(read_size));
												//要素の終端ではなかったため、バッファのカレントポインタを戻す
			}
		}
		return !result.hasFatalError();
	}
	//配列ブロック読み込み終了
	bool finishReadArray(CIArchiveHelper parent_arc, CIArchiveHelper child_arc)
	{
		parent_arc.addResult(child_arc.getResult());//親に処理結果を計上
		CResult& result = child_arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		parent_arc.seek(result, child_arc.getBuffUsed());
```

```
	return !result.hasFatalError();
}
//ブロックの読み込みをスキップ
bool skipReadBlock(CIArchiveHelper arc)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	char begin_mark[BEGIN_MARK_SIZE];//ブロック始端読み込みバッファ
	arc.read(result, begin_mark, BEGIN_MARK_SIZE);//ブロック始端読み込み
	if (memcmp(begin_mark, BLOCK_BEGIN, BEGIN_MARK_SIZE) != 0)//ブロック始端チェック
	{
		result.setHasFatalError();
		return false;
	}
	crc32_t name_crc = 0;
	CItemAttr attr(false, false, false, false, false);
	arc.read(result, &name_crc, sizeof(name_crc));//名前 CRC 読み込み
	arc.read(result, const_cast<CItemAttr::value_t*>(&attr.m_value), sizeof(attr.m_value));//属性書き込み
	if (attr.hasVer())//バージョン情報があるか？
	{
		CVersion input_ver_dummy;
		arc.read(result, const_cast<unsigned int*>(input_ver_dummy.getVerPtr()),
                                                    input_ver_dummy.getVerSize());//バージョン読み込み
	}
	if (!attr.isNul())//ヌル時はここまでの情報で終わり
	{
		std::size_t item_size;
		arc.read(result, &item_size, sizeof(item_size));//ブロックサイズ読み込み
		arc.seek(result, static_cast<int>(item_size));//ブロックサイズ分、バッファのカレントポインタを進める
	}
	//ブロックフッター読み込み
	char end_mark[END_MARK_SIZE];
	arc.read(result, end_mark, END_MARK_SIZE);//ブロック終端読み込み
	if (memcmp(end_mark, BLOCK_END, END_MARK_SIZE) != 0)//ブロック終端チェック
	{
		result.setHasFatalError();
		return false;
	}
	return !result.hasFatalError();
}
//ブロックフッター読み込み
//※読み込みテストの結果、ブロックフッターでなければデータ項目（オブジェクト）の読み込みを継続する
bool tryAndReadBlockFooter(CIArchiveHelper arc, const CItemBase& item, bool& is_block_end)
{
	is_block_end = true;//ブロック終了扱い
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	std::size_t read_size = 0;
	char end_mark[END_MARK_SIZE];
	arc.read(result, end_mark, END_MARK_SIZE, &read_size);//ブロック終端読み込み
	if (memcmp(end_mark, BLOCK_END, END_MARK_SIZE) != 0)//ブロック終端チェック
	{
		is_block_end = false;//ブロック終了ではない
		arc.seek(result, -static_cast<int>(read_size));
                                        //ブロックの終端ではなかったため、バッファのカレントポインタを戻す
	}
	return !result.hasFatalError();
}
//ブロック読み込み終了
bool finishReadBlock(CIArchiveHelper parent_arc, CIArchiveHelper child_arc)
{
	parent_arc.addResult(child_arc.getResult());//親に処理結果を計上
	CResult& result = child_arc.getResult();
```

```
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	parent_arc.seek(result, child_arc.getBuffUsed());
	return !result.hasFatalError();
}
//※読み込みテストの結果、ブロックフッターでなければデータ項目（オブジェクト）の読み込みを継続する
//次のブロックヘッダー問い合わせ（先行読み込み）
//※処理を進めず、次の情報を読み取るのみ
//※（例えば、バイナリスタイルなら、読み込みバッファのポインタを進めない）
bool requireNextBlockHeader(CIArchiveHelper arc, CItemBase& require_item, std::size_t& require_block_size,
                                                                                bool& is_found_next_block)
{
	is_found_next_block = false;//ブロック情報が見つかってないことにする
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	std::size_t read_size = 0;//読み込みサイズ
	require_item.clearForLoad();//読み込み情報を一旦クリア
	char begin_mark[BEGIN_MARK_SIZE];//ブロック始端読み込みバッファ
	arc.read(result, begin_mark, BEGIN_MARK_SIZE, &read_size);//ブロック始端読み込み
	if (memcmp(begin_mark, BLOCK_BEGIN, BEGIN_MARK_SIZE) != 0)//ブロック始端チェック
	{
		arc.seek(result, -static_cast<int>(read_size));//仮読みした分、バッファのカレントポインタを戻す
		return !result.hasFatalError();
	}
	is_found_next_block = true;//見つかった
	arc.read(result, const_cast<crc32_t*>(&require_item.m_nameCrc), sizeof(require_item.m_nameCrc),
                                                                        &read_size);//名前 CRC 書き込み
	arc.read(result, const_cast<CItemAttr::value_t*>(&require_item.m_attr.m_value),
                                                sizeof(require_item.m_attr.m_value), &read_size);//属性書き込み
	if (require_item.m_attr.hasVer())//バージョン情報があるか？
	{
		CVersion input_ver_dummy;
		arc.read(result, const_cast<unsigned int*>(input_ver_dummy.getVerPtr()),
                                                input_ver_dummy.getVerSize(), &read_size);//バージョン読み込み
	}
	require_block_size = read_size;//ブロックサイズ一旦計上
	if (!require_item.isNul())//ヌル時はここまでの情報で終わり
	{
		arc.read(result, const_cast<std::size_t*>(&require_item.m_itemSize),
                                            sizeof(require_item.m_itemSize), &read_size);//ブロックサイズ読み込み
		require_block_size += require_item.m_itemSize;//ブロックサイズ計上
		if (require_item.isArr())//配列時は配列要素数も読み込み
		{
			char begin_mark[BEGIN_MARK_SIZE];//配列ブロック始端読み込みバッファ
			arc.read(result, begin_mark, BEGIN_MARK_SIZE);//配列ブロック始端読み込み
			if (memcmp(begin_mark, ARRAY_BEGIN, BEGIN_MARK_SIZE) == 0)//配列ブロック始端チェック
			{
				//std::size_t array_block_size = 0;
				arc.read(result, const_cast<std::size_t*>(&require_item.m_arrNum),
                                                        sizeof(require_item.m_arrNum));//配列要素数読み込み
				//arc.read(result, const_cast<std::size_t*>(&array_block_size),
                                                    sizeof(array_block_size));//配列ブロックサイズ読み込み
			}
		}
	}
	arc.seek(result, -static_cast<int>(read_size));//仮読みした分、バッファのカレントポインタを戻す
	return !result.hasFatalError();
}
//ターミネータ読み込み
bool readTerminator(CIArchiveHelper arc)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
```

```
            char terminator[TERMINATOR_SIZE];
            arc.read(result, terminator, TERMINATOR_SIZE);//ターミネータ読み込み
            if (memcmp(terminator, TERMINATOR, TERMINATOR_SIZE) != 0)//ターミネータチェック
                  result.setHasFatalError();
            return true;
      }
public:
      //コンストラクタ
      CIArchiveStyleBinary() :
            CArchiveStyleBinaryBase(),
            m_readSizeForPrevDataItem(0)
      {}
      //親を受け取るコンストラクタ
      //※処理階層が深くなるごとにコピーが行われる
      CIArchiveStyleBinary(CIArchiveStyleBinary& parent) :
            CArchiveStyleBinaryBase(parent),
            m_readSizeForPrevDataItem(0)
      {}
      //デストラクタ
      ~CIArchiveStyleBinary()
      {}
private:
      //フィールド
      std::size_t m_readSizeForPrevDataItem;//【リトライ用】前回のデータ読み込みサイズ
};
```

▼ アーカイブ形式クラス：テキスト形式

【テキスト形式アーカイブクラス（共通）】

```
//--------------------
//テキスト形式アーカイブクラス（共通）
class CArchiveStyleTextBase : public CArchiveStyleBase
{
public:
      //コンストラクタ
      CArchiveStyleTextBase() :
            CArchiveStyleBase()
      {}
      //親を受け取るコンストラクタ
      //※処理階層が深くなるごとにコピーが行われる
      CArchiveStyleTextBase(CArchiveStyleTextBase& parent) :
            CArchiveStyleBase(parent)
      {}
      //デストラクタ
      ~CArchiveStyleTextBase()
      {}
};
```

【テキスト形式アーカイブクラス（アーカイブ書き込み用）】

```
//--------------------
//テキスト形式アーカイブクラス（アーカイブ書き込み用）
class COArchiveStyleText : public CArchiveStyleTextBase
{
public:
      //定数
      enum PROCESS
      {
            PROCESS_UNKNOWN = 0,
            PROCESS_TOP,
            PROCESS_BLOCK,
            PROCESS_ARRAY_BLOCK,
            PROCESS_ELEM,
```

```
            PROCESS_ITEM,
      };
public:
      //メソッド
      //シグネチャ書き込み
      bool writeSignature(COArchiveHelper arc)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            m_process = PROCESS_TOP;
            arc.print(result, "{¥"serializer¥": {");
            m_blockIndex = 0;
            return !result.hasFatalError();
      }
      //ブロックヘッダー書き込み
      bool writeBlockHeader(COArchiveHelper arc, const CItemBase& item, const CVersion& ver)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            if (m_blockIndex != 0 || m_process == PROCESS_ARRAY_BLOCK ||
                                                      (m_process == PROCESS_ELEM && m_itemIndex != 0))
                  arc.print(result, ",");
            if (m_process == PROCESS_ELEM)
                  ++m_itemIndex;
            arc.print(result, "¥n");
            m_process = PROCESS_BLOCK;
            arc.printIndent(result, 0);
            arc.print(result, "¥"%s¥": {", item.m_name);
            arc.print(result, "¥"crc¥": 0x%08x, ", item.m_nameCrc);
            arc.print(result, "¥"itemType¥": ¥"%s¥", ", item.m_itemType->name());
            arc.print(result, "¥"itemSize¥": %d, ", item.m_itemSize);
            arc.print(result, "¥"isObj¥": %d, ", item.isObj());
            arc.print(result, "¥"isArr¥": %d, ", item.isArr());
            arc.print(result, "¥"isPtr¥": %d, ", item.isPtr());
            arc.print(result, "¥"isNul¥": %d, ", item.isNul());
            arc.print(result, "¥"isVLen¥": %d, ", item.isVLen());
            arc.print(result, "¥"hasVer¥": %d, ", item.m_attr.hasVer());
            arc.print(result, "¥"ver¥": ¥"%d.%d¥"", ver.getMajor(), ver.getMinor());
            m_arrayBlockIndex = 0;
            return !result.hasFatalError();
      }
      //配列ブロックヘッダー書き込み
      bool writeArrayHeader(COArchiveHelper arc, const CItemBase& item, const std::size_t array_elem_num)
      {
            CResult& result = arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
                  return false;
            if (item.isArr())
            {
                  arc.print(result, ",");
                  arc.print(result, "¥n");
                  m_process = PROCESS_ARRAY_BLOCK;
                  arc.printIndent(result, 0);
                  arc.print(result, "¥"arrayNum¥": %d, ¥"array¥": [", array_elem_num);
                  m_elemIndex = 0;
            }
            else
            {
                  m_process = PROCESS_ARRAY_BLOCK;
                  m_elemIndex = -1;
            }
            return !result.hasFatalError();
      }
```

```
//要素ヘッダー書き込み
bool writeElemHeader(COArchiveHelper arc, const CItemBase& item, const std::size_t index)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	if (!item.isArr() || (item.isArr() && m_elemIndex != 0))
		arc.print(result, ",");
	arc.print(result, "¥n");
	m_process = PROCESS_ELEM;
	arc.printIndent(result, 0);
	if (item.isArr())
	{
		arc.print(result, "[");
	}
	else
	{
		arc.print(result, "¥"elem¥": {");
	}
	m_itemIndex = 0;
	return !result.hasFatalError();
}
//データ項目書き込み
bool writeDataItem(COArchiveHelper arc, const CItemBase& item, const CItemBase& child_item)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	if (m_itemIndex != 0)
		arc.print(result, ",");
	arc.print(result, "¥n");
	m_process = PROCESS_ITEM;
	arc.printIndent(result, 0);
	arc.print(result, "¥"%s¥": {", child_item.m_name);
	arc.print(result, "¥"crc¥": 0x%08x, ", child_item.m_nameCrc);
	arc.print(result, "¥"itemType¥": ¥"%s¥", ", child_item.m_itemType->name());
	arc.print(result, "¥"itemSize¥": %d, ", child_item.m_itemSize);
	arc.print(result, "¥"isObj¥": %d, ", child_item.isObj());
	arc.print(result, "¥"isArr¥": %d, ", child_item.isArr());
	arc.print(result, "¥"isPtr¥": %d, ", child_item.isPtr());
	arc.print(result, "¥"isNul¥": %d, ", child_item.isNul());
	arc.print(result, "¥"isVLen¥": %d, ", child_item.isVLen());
	arc.print(result, "¥"hasVer¥": %d, ", child_item.m_attr.hasVer());
	if (child_item.isArr())
		arc.print(result, "¥"arrNum¥": %d, ", child_item.m_arrNum);
	arc.print(result, "¥"data¥": ");
	{
		if (child_item.isNul())
			arc.print(result, "null");
		else
		{
			unsigned char* p = reinterpret_cast<unsigned char*>(const_cast<void*>(child_item.m_itemP));
			const std::size_t elem_num = child_item.getElemNum();
			if (child_item.isArr())
				arc.print(result, "[");
			for (std::size_t index = 0; index < elem_num && !result.hasFatalError(); ++index)
                                                                        //配列要素数分データ書き込み
			{
				if (index != 0)
					arc.print(result, ",");
				arc.printWithFunc(result, child_item.m_typeCtrl.m_toStrFuncP, p, child_item.m_itemSize);
				p += child_item.m_itemSize;
			}
			if (child_item.isArr())
				arc.print(result, "]");
```

```
                }
            }
            arc.print(result, "}");
            ++m_itemIndex;
            m_process = m_parentProcess;
            return !result.hasFatalError();
        }
        //要素フッター書き込み
        bool writeElemFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
                                                    const std::size_t index, short& items_num, std::size_t& elem_size)
        {
            CResult& result = child_arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
            {
                parent_arc.addResult(result);//親に処理結果を計上
                return false;
            }
            child_arc.print(result, "}");
            parent_arc.seek(result, child_arc.getBuffUsed());
            parent_arc.addResult(result);//親に処理結果を計上
            if (m_parent)
                ++m_parent->m_elemIndex;
            m_process = m_parentProcess;
            return !result.hasFatalError();
        }
        //配列ブロックフッター書き込み
        bool writeArrayFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc, const CItemBase& item,
                                                                        std::size_t& array_block_size)
        {
            CResult& result = child_arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
            {
                parent_arc.addResult(result);//親に処理結果を計上
                return false;
            }
            if (item.isArr())
            {
                child_arc.print(result, "]");
            }
            parent_arc.seek(result, child_arc.getBuffUsed());
            parent_arc.addResult(result);//親に処理結果を計上
            if (m_parent)
                ++m_parent->m_arrayBlockIndex;
            m_process = m_parentProcess;
            return !result.hasFatalError();
        }
        //ブロックフッター書き込み
        bool writeBlockFooter(COArchiveHelper parent_arc, COArchiveHelper child_arc,
                                                                    const CItemBase& item, std::size_t& block_size)
        {
            CResult& result = child_arc.getResult();
            if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
            {
                parent_arc.addResult(result);//親に処理結果を計上
                return false;
            }
            child_arc.print(result, "}");
            parent_arc.seek(result, child_arc.getBuffUsed());
            parent_arc.addResult(result);//親に処理結果を計上
            if (m_parent)
                ++m_parent->m_blockIndex;
            m_process = m_parentProcess;
            return !result.hasFatalError();
        }
        //ターミネータ書き込み
```

```
	bool writeTerminator(COArchiveHelper arc)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		arc.print(result, "¥n");
		arc.print(result, "], ¥"terminator¥": ¥"ok¥"]¥n");
		return !result.hasFatalError();
	}
public:
	//コンストラクタ
	COArchiveStyleText() :
		CArchiveStyleTextBase(),
		m_parent(nullptr),
		m_nestLevel(0),
		m_process(PROCESS_UNKNOWN),
		m_blockIndex(0),
		m_arrayBlockIndex(0),
		m_elemIndex(0),
		m_itemIndex(0),
		m_parentProcess(PROCESS_UNKNOWN),
		m_parentBlockIndex(0),
		m_parentArrayBlockIndex(0),
		m_parentElemIndex(0),
		m_parentItemIndex(0)
	{}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	COArchiveStyleText(COArchiveStyleText& src) :
		CArchiveStyleTextBase(src),
		m_parent(&src),
		m_nestLevel(src.m_nestLevel + 1),
		m_process(src.m_process),
		m_blockIndex(0),
		m_arrayBlockIndex(0),
		m_elemIndex(0),
		m_itemIndex(0),
		m_parentProcess(src.m_process),
		m_parentBlockIndex(src.m_blockIndex),
		m_parentArrayBlockIndex(src.m_arrayBlockIndex),
		m_parentElemIndex(src.m_elemIndex),
		m_parentItemIndex(src.m_itemIndex)
	{}
	//デストラクタ
	~COArchiveStyleText()
	{}
private:
	//フィールド
	COArchiveStyleText* m_parent;//親アーカイブ形式オブジェクト
	int m_nestLevel;//データのネストレベル
	PROCESS m_process;//処理状態
	int m_blockIndex;//ブロックインデックス
	int m_arrayBlockIndex;//配列ブロックインデックス
	int m_elemIndex;//要素インデックス
	int m_itemIndex;//データ項目インデックス
	PROCESS m_parentProcess;//親の処理状態
	int m_parentBlockIndex;//親のブロックインデックス
	int m_parentArrayBlockIndex;//親の配列ブロックインデックス
	int m_parentElemIndex;//親の要素インデックス
	int m_parentItemIndex;//親のデータ項目インデックス
};
```

【テキスト形式アーカイブクラス（アーカイブ読み込み用）】

```
//--------------------
//テキスト形式アーカイブクラス（アーカイブ読み込み用）
```

```
class CIArchiveStyleText : public CArchiveStyleTextBase
{
public:
	//メソッド
	//パース
	bool parse(CIArchiveHelper arc)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//シグネチャ読み込み
	bool readSignature(CIArchiveHelper arc)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//ブロックヘッダー読み込み
	//※読み込んだオブジェクトの型情報とバージョンを返す
	bool readBlockHeader(CIArchiveHelper arc, const CItemBase& item, const CItemBase* delegate_item,
			const CVersion& ver, CItemBase& input_item, CVersion& input_ver, std::size_t& block_size)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//配列ブロックヘッダー読み込み
	bool readArrayHeader(CIArchiveHelper arc, const CItemBase& item, std::size_t& array_elem_num,
			std::size_t& array_block_size)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//要素ヘッダー読み込み
	bool readElemHeader(CIArchiveHelper arc, const CItemBase& item, const std::size_t index,
			short& items_num, std::size_t& elem_size)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//データ項目読み込み
	bool readDataItem(CIArchiveHelper arc, const CItemBase& item, const CItemBase* delegate_child_item_now,
			const bool item_is_valid, const bool is_required_retry)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//要素フッター読み込み
	//※読み込みテストの結果、要素フッターでなければデータ項目の読み込みを継続する
```

```
bool tryAndReadElemFooter(CIArchiveHelper arc, const CItemBase& item, const std::size_t index, bool& is_elem_end)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//要素読み込み終了
bool finishReadElem(CIArchiveHelper parent_arc, CIArchiveHelper child_arc)
{
	CResult& result = child_arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//配列ブロックフッター読み込み
//※読み込みテストの結果、配列ブロックフッターでなければデータ項目の読み込みを継続する
bool tryAndReadArrayFooter(CIArchiveHelper arc, const CItemBase& item, bool& is_array_block_end)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//配列ブロック読み込み終了
bool finishReadArray(CIArchiveHelper parent_arc, CIArchiveHelper child_arc)
{
	CResult& result = child_arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//ブロックの読み込みをスキップ
bool skipReadBlock(CIArchiveHelper arc)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//ブロックフッター読み込み
//※読み込みテストの結果、ブロックフッターでなければデータ項目（オブジェクト）の読み込みを継続する
bool tryAndReadBlockFooter(CIArchiveHelper arc, const CItemBase& item, bool& is_block_end)
{
	CResult& result = arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//ブロック読み込み終了
bool finishReadBlock(CIArchiveHelper parent_arc, CIArchiveHelper child_arc)
{
	CResult& result = child_arc.getResult();
	if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
		return false;
	//未実装
	return !result.hasFatalError();
}
//※読み込みテストの結果、ブロックフッターでなければデータ項目（オブジェクト）の読み込みを継続する
```

```
	//次のブロックヘッダー問い合わせ（先行読み込み）
	//※処理を進めず、次の情報を読み取るのみ
	//※（例えば、バイナリスタイルなら、読み込みバッファのポインタを進めない）
	bool requireNextBlockHeader(CIArchiveHelper arc, CItemBase& require_item, std::size_t& require_block_size,
	                                                                                  bool& is_found_next_block)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
	//ターミネータ読み込み
	bool readTerminator(CIArchiveHelper arc)
	{
		CResult& result = arc.getResult();
		if (result.hasFatalError())//致命的なエラーが出ている時は即時終了する
			return false;
		//未実装
		return !result.hasFatalError();
	}
public:
	//コンストラクタ
	CIArchiveStyleText() :
		CArchiveStyleTextBase()
	{}
	//親を受け取るコンストラクタ
	//※処理階層が深くなるごとにコピーが行われる
	CIArchiveStyleText(CIArchiveStyleText& parent) :
		CArchiveStyleTextBase(parent)
	{}
	//デストラクタ
	~CIArchiveStyleText()
	{}
};
```

▼ アーカイブクラス

アーカイブ読み書きクラスのテンプレート引数にアーカイブ形式クラスを与えることで、実際に使用可能なアーカイブクラスになる。

【バイナリ形式アーカイブ書き込みクラス】

```
//--------------------
//バイナリ形式アーカイブ書き込みクラス
using COBinaryArchive = COArchive<COArchiveStyleBinary>;
```

【バイナリ形式アーカイブ読み込みクラス】

```
//--------------------
//バイナリ形式アーカイブ読み込みクラス
using CIBinaryArchive = CIArchive<CIArchiveStyleBinary>;
```

【テキスト形式アーカイブ書き込みクラス】

```
//--------------------
//テキスト形式アーカイブ書き込みクラス
using COTextArchive = COArchive<COArchiveStyleText>;
```

【テキスト形式アーカイブ読み込みクラス】※未実装

```
//--------------------
//テキスト形式アーカイブ読み込みクラス
using CITextArchive = CIArchive<CIArchiveStyleText>;
```

▼ システムの依存関係

このサンプルプログラムは、別紙に掲載する他のサンプルプログラムを必要とする。
下記のプログラムである。

・ スタックアロケータクラス／スタックアロケータアダプタークラス
  ➢ 別紙の「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。

・ 多態アロケータクラス／グローバル多態アロケータ
  ➢ 別紙の「様々なメモリ管理手法と共通アロケータインターフェース」に掲載。

サンプルプログラム内では STL も使用しているが、この多態アロケータにより、ヒープからメモリを取られず、固定バッファで動作するようにしている。

■■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