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

– メモリ操作の汎用化と共通ライブラリの有効活用 –

2014 年 2 月 24 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 24 日	板垣 衛	（初稿）

■ 目次

■ 概略

ゲームプログラミングを安全かつ快適にするためのメモリ管理手法と操作手法を、段階的に説明する。

■ 目的

本書は、STL のような内部で暗黙的にメモリ確保を行う処理を、安全に利用するための手法を確立し、ゲームプログラミングの安全性と生産性を確保することを目的とする。

メモリ操作に対する一定の理解が必要なため、メモリ操作に関する基礎技術から始め、具体的なメモリ管理手法の説明、効果的なメモリ操作手法の説明を記載する。

なお、本書で扱うメモリ管理は、別紙の「ゲーム制御のためのメモリ管理方針」と「柔軟性を追求したメモリ管理システム」で扱っているような大掛かりなシステムではく、もっと基礎的なものである。

■ 本書の内容について

本書は、最終的にシンプルで強力なメモリ操作の手法を説明する。
その最終目的の理解を得るために、段階的な内容で構成している。

メモリ操作の基礎の説明から始まり、それによる問題点と解決策を示し、幾つかのシンプルなメモリ管理の仕組みと活用法を解説し、それらを統合的に活用するための共通メモリインターフェースを策定する。
これにより、グローバル new / delete 演算子に多態性を与えることが可能となり、シンプルなプログラミングで非常に強力なメモリ操作を行うことができるようになる。

▼ メリット

本書が最終的に提示する仕組みにより、様々なメモリ管理の仕組みを導入しつつ、ほぼすべてのメモリ操作をグローバル new / delete 演算子で行うことができるようになる。
これによるメリットは下記のとおり。

・ コードの再利用性が高まる

・ STL のように、内部でメモリ確保を行う標準ライブラリを便利に活用できる。
  ➢ カスタムアロケータを作る必要もない。
・ 配列の配置 new / 配置 delete を使用した時、および、多重継承クラスをアップキャストした時に起こる「ポインタのずれ」の問題を解消できる。

▼ デメリット

多態性を持たせることにより、実際のメモリ確保関数を呼び出すまでの処理が少し長くなる。TLS（スレッドローカルストレージ）からメモリ管理オブジェクトのポインタを参照し、vtable を経由して仮想メソッドが呼び出されてから、実際の処理が実行される。内部的にはこのようなまわりくどい処理が行われる。頻繁なメモリ操作では低速化が懸念される。

しかし、この問題は開発プロジェクトの「プログラミング方針」によって回避できる。

まず、この仕組みの導入によって、それまでプログラム構造上、（低速な）ヒープメモリで処理しなければならなかったような箇所を、高速なスタックアロケータを使った処理に置き換えることができる。STL のような標準ライブラリを使用している箇所に特に有効である。

このように、生産性を損ねずに高速なメモリアロケータの活用の幅を広げることで、デメリットは十分相殺できる。

▼ マルチスレッド対応

本書は、最後にマルチスレッドで有効なメモリ操作手法を説明する。

■ ゲームプログラミングのメモリ操作に対するアプローチ

メモリ操作には問題が多い。

「配置 new / 配置 delete を使ったらデストラクタがきちんと実行されない」、「配置 new / 配置 delete を使ったら配列 new の delete がうまくいかない」、「STL を使ったらメモリ状態が不安定になった」など様々である。

メモリ事情の厳しいゲームプログラミングでは、コーディング規約で厳しい制約を設けることでこうした問題に対処することも珍しくない。「デストラクタは使っちゃダメ」、「配列の new は使っちゃダメ」、「STL は使っちゃダメ」などである。

しかし、基本的な仕組みによってこれらの問題が解消でき、プログラミングに対する制

限を緩和できるなら、間違いなく生産性と安全性を高めることができる。本書は、そのための道筋を示すものである。

■ 配置 new / 配置 delete の基礎

まず、基本として、「配置 new」(placement new = プレースメント new)と「配置 delete」(placement delete = プレースメント delete) について説明する。

▼ 配置 new / 配置 delete とは？

「配置 new」「配置 delete」とは、(本来) 任意の場所 (メモリ) にデータを「配置」することを意味したものである。

任意の new / delete オペレータを作成し、第 2 引数以降に任意のパラメータを指定してオーバーロードすしたものを配置 new / 配置 delete と呼ぶ。

規定の new / delete :

```
void* operator new(const std::size_t size) throw(std::bad_alloc);
void operator delete(void * p) throw(std::bad_alloc);
```

配置 new / 配置 delete のサンプル :

```
void* operator new(const std::size_t size, void* where) throw();
void operator delete(void* p, void* where) throw();
```

より複雑な配置 new / 配置 delete のサンプル :

```
void* operator new(const std::size_t size, CAllocator& allocator, const std::size_t align) throw();
void operator delete(void* p, CAllocator& allocator, const std::size_t) throw();
```

● 一対の配置 new と配置 delete

「配置 new」と「配置 delete」は必ず一対で用意する必要がある。

それぞれ第 2 引数以降の内容が一致した関数を対で定義するということである。

その理由は、コンストラクタで例外が発生した場合、delete 関数が自動的に呼び出されるためである。

new 関数でメモリを確保して return すると、続いて自動的にコンストラクタが呼び出される。そのコンストラクタの処理中に例外がスローされた場合、new が失敗したものとして処理するために、自動的にメモリの解放が行われる。この時、デストラクタは呼び出されない。

● 配置 new / 配置 delete と非配置 new / 非配置 delete

「配置 new」「配置 delete」とは、あくまでも「第 2 引数以降が指定された new / delete オペレータ」のことである。言葉の意味どおり、任意の場所への「配置」を目的としたものでなくとも、「第 2 引数以降が指定」されれば、それは「配置 new」「配置 delete」である。

オーバーロードした new / delete オペレータが全て配置 new / 配置 delete というわけではない。

標準的な new / delete 演算子として振る舞う任意の処理を定義することができるが、それらは「非配置 new」「非配置 delete」である。

グローバル new / delete（標準の new / delete）オペレータの置き換えや、クラス内 new / delete によって定義できる。

非配置 new / 非配置 delete のサンプル：（余計なパラメータがない）

```
//グローバル new/delete
void* operator new(const std::size_t size) throw();
void operator delete(void* p) throw();

//クラス
class CTest
{
public:
	//クラス内 new/delete
	//※static である必要があるが、付けなくても勝手に static 扱いになる
	static void* operator new(const std::size_t size) throw();
	static void operator delete(void* p) throw();
	static void operator delete(void* p, const std::size_t size) throw();
                                                    //クラス内 delete は、この形も非配置 delete として使える
                                                    //両方定義したら operator delete(void*) の方が優先
};
```

▼ 規定の配置 new でコンストラクタ呼び出し

自分で確保したバッファにクラスのインスタンスを作るために、コンストラクタ呼び出しだけを行いたい場合がある。

そのような目的の配置 new は、#include <new> に定義されているので、そのまま使うことができる。

配置 delete の呼び出す時に operator delete を明示しなければならい点と、デストラクタを明示的に呼び出す必要がある点に注意。

規定の配置 new / 配置 delete：

```
//※#include <new> で定義されている規定の配置 new/配置 delete
void* operator new(size_t, void* where) throw();
void* operator new[](size_t, void* where) throw();
void operator delete(void*, void*) throw();
void operator delete[](void*, void*) throw();
```

```
void* operator new(size_t size, const nothrow_t&) throw();//例外をスローしない
void* operator new[](size_t size, const nothrow_t&) throw();//例外をスローしない
void operator delete(void*, const nothrow_t&) throw();//例外をスローしない
void operator delete[](void*, const nothrow_t&) throw();//例外をスローしない
```

規定の配置 new / 配置 delete を使ったサンプル：

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest0
{
public:
	//コンストラクタ
	CTest0(const char* name) :
		m_name(name)
	{
		printf("CTest0::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CTest0()
	{
		printf("CTest0::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	const char* m_name;//名前
};
```

【テスト】

```
//--------------------
//テスト
void test0()
{
	char buff[32];//インスタンス用のバッファ
	printf("buff=0x%p¥n", buff);
	CTest0* obj_p = new(buff)CTest0("テスト0");
	printf("obj_p=0x%p¥n", obj_p);
	obj_p->~CTest0();//明示的なデストラクタ呼び出し
	operator delete(obj_p, buff);//operator delete で配置 delete を呼び出す（デストラクタが呼び出されない点に注意）
}
```

↓（実行結果）

```
buff=0x0024F838		←バッファの先頭アドレス
CTest0::Constructor : name="テスト0"
obj_p=0x0024F838		←new で生成したオブジェクトのポインタが、new に受け渡したバッファと同じ
CTest0::Destructor : name="テスト0"
```

● new / delete の例外スローについて

規定の配置 new / 配置 delete を見ると、すべて「throw()」が付いている。

これは例外をスローしないことを意味しており、これが付いていないと、(コンパイラによっては) ヌルチェックせずにコンストラクタを呼び出す。つまり、「try～catch」が必須なのである。

「throw()」が付いているものは、例外をスローしない代わりに、コンパイラが new 演算子の戻り値をチェックし、ヌルならコンストラクタを呼び出さない。

参考までに、例外をスローする(普通の)new / delete は、「throw(std::bad_alloc)」

が付く。

この「new の戻り値をヌルチェックしない」というのは、GCC の標準仕様である。そもそも new 演算子に対する要件が「決してヌルポインタを返してはならない」と規定されている。GCC でこの標準の挙動を変えてヌルチェックするようにするには、コンパイル時に、コンパイラオプション「-fcheck-new」を付けるか、全ての配置 new / 配置 delete（および非配置 new / 非配置 delete）に「throw()」を付けることである。

本書では、全てに「throw()」を付けて記述する。実際には「CHECK_NEW」のようなマクロを用意したほうが良いかもしれない。Visual C++ では「_THROW0()」というマクロを用意して使用している。

▼ 配置 delete を隠蔽した delete 関数でデストラクタを自動呼び出し

上記のサンプルでは、配置 delete を呼び出すために operator delete を明示し、かつ、デストラクタを明示的に呼び出していたが、これをもう少し簡略化するために、専用の delete 関数を用意する。

テンプレート関数を使用することで、配置 delete を使用した場合でもデストラクタを呼び出せる扱い易い delete 関数を定義できる。

ついでにポインタ変数の初期化も行い、安全性を高めることもできる。

難点としては、専用関数として構成するため、構文が new と対をなさず、分かりにくい、もしく、気持ち悪いという点がある。

なお、下記のサンプルコードでは配列の delete を扱っていない。それは、汎用的な配列の delete 関数を作れないためである。この問題の意味と解消方法については後述する。

配置 delete を隠蔽した delete テンプレート関数のサンプル：

【配置 delete 用 delete テンプレート関数定義】

```
//--------------------
//配置 delete 用 delete 関数
template<class T>
void delete_ptr(*& p, char* buff)
{
	if (!p)
		return;
	p->~T();//デストラクタ呼び出し（デストラクタが定義されていない型やプリミティブ型でも問題なし）
	//operator delete(p, p);//配置 delete の呼び出しは、特に必要ではない
	p = nullptr;//安全のためにポインタを初期化
}
//※配列版の delete 関数はなし
//  型にデストラクタが定義されているかどうかを気にせず使えるるような汎用処理が作れないため
//  クラス内 delete を用いることでこの問題を解消できる
```

【テスト】（「既定の配置 mew / 配置 delete」のサンプルを一部改訂したもの）

```
//--------------------
//テスト
void test0()
```

```
{
	char buff[32];//インスタンス用のバッファ
	printf("buff=0x%p¥n", buff);
	CTest0* obj_p = new(buff)CTest0("テスト 0");
	printf("obj_p=0x%p¥n", obj_p);
	delete_ptr(obj_p, buff);//専用 delete 関数で削除（内部でデストラクタを呼び出す）
	printf("obj_p=0x%p¥n", obj_p);//削除後のポインタを確認
	//不要になった処理
	//obj_p->~CTest0();//明示的なデストラクタ呼び出し
	//operator delete(obj_p, buff);//operator delete で配置 delete を呼び出す（デストラクタが呼び出されない点に注意）
}
```

↓（実行結果）

```
buff=0x0024F838
CTest0::Constructor : name="テスト 0"
obj_p=0x0024F838
CTest0::Destructor : name="テスト 0"	←正しくデストラクタが呼び出されている
obj_p=0x00000000	←ポインタがクリアされている
```

● 配置 new を隠蔽した new 関数で delete 関数と一対の構文

オブジェクトを生成する処理の「= new(buff) TypeName()」と、削除する処理の「delete_ptr()」では、構文が対になっておらず、分かりにくく、気持ち悪い。

この対策としては、配置 new の方の構文を delete 関数に合わせる方法がある。これは、C++11 の可変長テンプレート引数を用いる事で実現できる。

配置 new を隠蔽した new テンプレート関数のサンプル：

【配置 new 用 new テンプレート関数定義】

```
//--------------------
//配置 new 用 new 関数
//※可変長テンプレート引数を使用して、コンストラクタの引数を取得
template<class T, typename... Tx>
T* new_ptr(char* buff, Tx... nx)
{
	if (!buff)
		return nullptr;
	return new(buff)T(nx...);//配置 new 呼び出し（コンストラクタは可変長比数渡し）
}
//※配列版の new 関数はなし
//  クラス内 delete と対のクラス内 new を用いたほうがよい
```

【テスト】（「既定の配置 mew / 配置 delete」のサンプルを一部改訂したもの）

```
//--------------------
//テスト
void test0()
{
	char buff[32];//インスタンス用のバッファ
	printf("buff=0x%p¥n", buff);
	CTest0* obj_p = new_ptr<CTest0>(buff, "テスト 0");
	//不要になった処理
	//CTest0* obj_p = new(buff)CTest0("テスト 0");
	printf("obj_p=0x%p¥n", obj_p);
	delete_ptr(obj_p, buff);//専用 delete 関数で削除（内部でデストラクタを呼び出す）
	printf("obj_p=0x%p¥n", obj_p);//削除後のポインタを確認
}
```

↓（実行結果）

```
buff=0x0024F838
CTest0::Constructor : name="テスト 0"	←正しくパラメータが渡されてコンストラクタが呼び出されている
obj_p=0x0024F838
```

```
CTest0::Destructor : name="テスト 0"
obj_p=0x00000000
```

▼ 配列の new / delete の注意点

配列の new / delete には、気をつけなければならない点がある。

new / delete の挙動を確認するために、配置 new / 配置 delete を利用し、関数に受け渡される値などを表示してみる。

配列の配置 new / 配置 delete の挙動確認用サンプル：

【挙動確認のための配置 new/配置 delete 定義】

```
//--------------------
//配置 new/配置 delete
//※それぞれ第 2 引数に char* を取り、既定の関数をオーバーロードする
//単体 new
void* operator new(const std::size_t size, char* where) throw()
{
	printf("placement new(size=%d, where=0x%p)¥n", size, where);
	return where;//渡されたバッファをそのまま返す
}
//配列 new
//※デストラクタを持った型の場合、バッファの先頭に要素数を格納するために、少し大きなサイズが渡される
//  この時、戻り値のポインタは、関数を抜けた時、要素数情報の分がシフトしたポインタに変換される
//  型のアラインメントによっては大きくシフトする可能性もある
//  デストラクタを持たない型もしくはプリミティブ型ではこの情報は扱われない（なのでややこしい）
void* operator new[](const std::size_t size, char* where) throw()
{
	printf("placement new[](size=%d, where=0x%p)¥n", size, where);
	return where;//渡されたバッファをそのまま返す
}
//単体 delete
//※配置 new と対になる配置 delete は必須
//  （コンストラクタで例外が発生すると自動的に呼び出されるため）
void operator delete(void* p, char* where) throw()
{
	printf("placement delete(p=0x%p, where=0x%p)¥n", p, where);
	//なにもしない
}
//配列 delete
//※デストラクタを持った型のポインタであっても、受け渡されたポインタがそのまま扱われるので注意
//  （メモリ確保時の正確なポインタが分からない）
//※クラス内 delete 関数を使用することでこの問題を解消できる
void operator delete[](void* p, char* where) throw()
{
	printf("placement delete[](p=0x%p, where=0x%p)¥n", p, where);
	//なにもしない
}
```

【挙動確認のための delete 関数定義】

```
//--------------------
//固定バッファ用 delete 関数
//※デストラクタを呼び出すテンプレート関数
template<class T>
void delete_ptr(T*& p)
{
	printf("delete_ptr(p=0x%p)¥n", p);
	if (!p)
		return;
	p->~T();//デストラクタ呼び出し
```

```
	operator delete(p, reinterpret_cast<char*>(p));//配置 delete 呼び出し
	p = nullptr;//安全のためにポインタを初期化
}
//挙動の確認のために配列版の delete 関数も用意
//（残念ながら）配列の要素数を受け渡す
template<class T>
void delete_ptr(T*& p, const std::size_t array_num)
{
	printf("delete_ptr(p=0x%p, array_num=%d)¥n", p, array_num);
	if (!p)
		return;
	for (std::size_t i = 0; i < array_num; ++i)
		p->~T();//デストラクタ呼び出し
	operator delete[](p, reinterpret_cast<char*>(p));//配置 delete 呼び出し
	p = nullptr;//安全のためにポインタを初期化
}
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest1
{
public:
	//デフォルトコンストラクタ
	CTest1() :
		CTest1("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
		//m_name("(default)")//C++11 が使えない場合は普通に初期化
	{
		printf("CTest1::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest1(const char* name) :
		m_name(name)
	{
		printf("CTest1::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CTest1()
	{
		printf("CTest1::Destructor : name=¥"%s¥"¥n", m_name);
	}
protected:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【テスト】

```
//--------------------
//テスト
void test1()
{
	//固定バッファでクラスのインスタンスを生成
	{
		printf("----------クラス¥n");
		char buff[32];//インスタンス用のバッファ
		printf("buff=0x%p¥n", buff);
		CTest1* obj_p = new(buff)CTest1("テスト 1");
		printf("obj_p=0x%p¥n", obj_p);
		delete_ptr(obj_p);
	}
	//固定バッファでクラス配列のインスタンスを生成
	{
		printf("----------クラスの配列¥n");
		char buff[64];//インスタンス用のバッファ
```

```
            printf("buff=0x%p¥n", buff);
            const int array_num = 3;
            CTest1* obj_p = new(buff)CTest1[array_num]();
            printf("obj_p=0x%p¥n", obj_p);
            printf("*reinterpret_cast<int*>(buff)=%d¥n", *reinterpret_cast<int*>(buff));
                                        //デストラクタを持った型の配列は、バッファの先頭に要素数が格納されている
            delete_ptr(obj_p, array_num);
      }
      //固定バッファでプリミティブ型のインスタンスを生成
      {
            printf("----------プリミティブ型¥n");
            char buff[32];//インスタンス用のバッファ
            printf("buff=0x%p¥n", buff);
            char* data_p = new(buff)char;
            printf("data_p=0x%p¥n", data_p);
            *data_p = 123;
            printf("*data_p(0x%p)=%d¥n", data_p, *data_p);
            delete_ptr(data_p);//デストラクタのない型で実行しても問題なし
      }
      //固定バッファでプリミティブ型の配列のインスタンスを生成
      {
            printf("----------プリミティブ型の配列¥n");
            char buff[32];//インスタンス用のバッファ
            printf("buff=0x%p¥n", buff);
            const int array_num = 3;
            char* data_p = new(buff)char[array_num];
            printf("data_p=0x%p¥n", data_p);
            printf("*reinterpret_cast<int*>(buff)=%d¥n", *reinterpret_cast<int*>(buff));
                                          //デストラクタを持たない型の配列は、バッファに要素数を格納していない
            data_p[0] = 123;
            printf("*data_p(0x%p)[0]=%d¥n", data_p, data_p[0]);
            delete_ptr(data_p, array_num);//デストラクタのない型で実行しても問題なし
      }
}
```

↓（実行結果）

```
----------クラス
buff=0x00C2FDD4
placement new(size=8, where=0x00C2FDD4)
CTest1::Constructor : name="テスト1"
obj_p=0x00C2FDD4
delete_ptr(p=0x00C2FDD4)
CTest1::Destructor : name="テスト1"
placement delete(p=0x00C2FDD4, where=0x00C2FDD4)
----------クラスの配列
buff=0x00C2FDA0                              ←バッファの先頭アドレス
placement new[](size=28, where=0x00C2FDA0)   ←new に渡されたサイズが 4 バイト多い（8 * 3 + 4）
CTest1::Constructor : name="(default)"
CTest1::DefaultConstructor : name="(default)"
CTest1::Constructor : name="(default)"
CTest1::DefaultConstructor : name="(default)"
CTest1::Constructor : name="(default)"
CTest1::DefaultConstructor : name="(default)"
obj_p=0x00C2FDA4                             ←new が返したポインタが 4 バイトシフトしている
*reinterpret_cast<int*>(buff)=3              ←バッファの先頭を確認すると要素数が格納されている
delete_ptr(p=0x00C2FDA4, array_num=3)
CTest1::Destructor : name="(default)"
CTest1::Destructor : name="(default)"
CTest1::Destructor : name="(default)"
placement delete[](p=0x00C2FDA4, where=0x00C2FDA4)    ←【問題】delete に渡してもポインタが 4 バイトシフトしたまま
----------プリミティブ型
buff=0x00C2FD60
placement new(size=1, where=0x00C2FD60)
data_p=0x00C2FD60
*data_p(0x00C2FD60)=123
```

```
delete_ptr(p=0x00C2FD60)
placement delete(p=0x00C2FD60, where=0x00C2FD60)
----------プリミティブ型の配列
buff=0x00C2FD2C                          ←バッファの先頭アドレス
placement new[](size=3, where=0x00C2FD2C)    ←new に渡されたサイズに増加はない
data_p=0x00C2FD2C                        ←new が返したポインタはそのまま
*reinterpret_cast<int*>(buff)=-858993460     ←バッファの先頭を確認しても要素数は格納されていない
*data_p(0x00C2FD2C)[0]=123
delete_ptr(p=0x00C2FD2C, array_num=3)
placement delete[](p=0x00C2FD2C, where=0x00C2FD2C)
```

上記の実行結果から確認できるとおり、「デストラクタを持った型の配列の new[]」の時だけ、純粋なデータサイズではなく、「配列の要素数」を格納するための領域が追加される。

プリミティブ型の場合はそのような領域の追加はなく、普通に要素サイズ×要素数で扱われる。これはデストラクタのない型（クラス／構造体）でも同様で、サンプルのクラスからデストラクタ「~CTest1()」を削除すると、領域の追記が行われないことが確認できる。

● 【問題点】ポインタのずれ

ここで問題なのは、new[]関数内で確保したポインタと異なるポインタを扱うことである。

配置 delete[]を使っているためか、delete[]関数呼び出し時に本来のポインタに戻されることを期待したが、実際にはそうはならなかった。(Visual C++ 2013, CygWin + GCC 4.8.2, Linux + GCC 4.4.7 でそれぞれ確認し、全て同じ結果)

これは、「operaotr delete」と明示して呼び出すことにより、「delete 演算子として」ではなく、単なる関数として呼び出されるためである。

独自のメモリアロケータと組み合わせて配置 delete[]を使用する場合、このポインタをそのまま使おうとすると、アロケート時のアドレスと異なるため、問題となる。

● 【解決策】クラス内 new / delete

この問題の解決策は、クラス内 new/delete を使うか、グローバル new / delete を置き換えることである。どちらの方法も後述する。

● 【類似の問題】ポインタのずれが生じる他のケース：多重継承

このような「ポインタのずれ」が生じるケースは、配列を使用した時だけではない。

それは「多重継承」を使用した時である。

クラスの多態性を利用する場合、サブクラスのインスタンスをスーパークラスのポインタで管理することはよくある。その場合、そのスーパークラスのポインタでそのまま delete することもある。この時、多重継承を使用していると、ポインタがずれるこ

とがあるので注意が必要である。

多重継承の問題確認用サンプル：

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest1
{
//…（略）…※上記と同じ
};
//--------------------
//テスト用クラス（多重継承テスト用）
class CTest1ex
{
public:
	//コンストラクタ
	CTest1ex()
	{
		printf("CTest1ex::Constructor¥n");
	}
	//デストラクタ
	~CTest1ex()
	{
		printf("CTest1ex::Destructor¥n");
	}
protected:
	//フィールド
	int m_dummy;//ダミー
};
//--------------------
//テスト用クラス（多重継承）
class CDerivedTest1 : public CTest1, public CTest1ex
{
public:
	//デフォルトコンストラクタ
	CDerivedTest1() :
		CTest1(),
		CTest1ex()
	{
		printf("CDerivedTest1::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CDerivedTest1(const char* name) :
		CTest1(name),
		CTest1ex()
	{
		printf("CDerivedTest1::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CDerivedTest1()
	{
		printf("CDerivedTest1::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	int m_dummy;//ダミー
};
```

【テスト】

```
//--------------------
//テスト
void test1()
{
	//固定バッファで多重継承したクラスのインスタンスを生成
```

```
    {
        printf("----------多重継承クラス¥n");
        char buff[32];//インスタンス用のバッファ
        printf("buff=0x%p¥n", buff);
        CDerivedTest1* obj_p = new(buff)CDerivedTest1("テスト１多重継承");
        CTest1* parent_p = obj_p;//親１にキャスト
        CTest1ex* parent_ex_p = obj_p;//親２にキャスト
        printf("obj_p=0x%p, parent_p=0x%p, parent_ex_p=0x%p¥n", obj_p, parent_p, parent_ex_p);
        //delete_ptr(obj_p);//子として削除
        //delete_ptr(parent_p);//親１として削除
        delete_ptr(parent_ex_p);//親２として削除
    }
}
```

↓（実行結果）

```
----------多重継承クラス
buff=0x0079F864
placement new(size=16, where=0x0079F864)
CTest1::Constructor : name="テスト１多重継承"
CTest1ex::Constructor
CDerivedTest1::Constructor : name="テスト１多重継承"
obj_p=0x0079F864, parent_p=0x0079F864, parent_ex_p=0x0079F86C     ←多重継承している場合、キャスト時にポインタの
delete_ptr(p=0x0079F86C)                                            位置を適切に変える事で、多数の親クラスに
CTest1ex::Destructor                                                振る舞いを変える事を実現している
placement delete(p=0x0079F86C, where=0x0079F86C)      ←【問題】配置 delete を呼んでも本来のポインタに戻らない
```

● 【多重継承問題の解決策】仮想化とクラス内 new / delete

この問題の解決策は、まず、親クラスを仮想化することである。

続いて、配列の問題の場合と同じく、クラス内 new / delete を定義するか、グローバル new / delete を置き換え、delete 時に「opertor delete」を明示して呼び出すのではなく「delete 演算子」として使用するようにする。

なお、仮想化の際は、virtual 関数を一つでも定義すればよい。ただし、「スーパークラスのポインタで delete」を行うのであれば、「デストラクタを virtual」にすることが必須となる。

● 【補足】多重継承のキャスト

上記のサンプルからも分かるとおり、多重継承したクラスは、キャストによってポインタが変化する。逆に言えば、このポインタ変化が正しく行われないと、キャストが成立しない。「reinterpret_cast」を使用してキャストすると、ポインタを変化させずに型だけ変えてしまうので、誤ったキャストになる。

間違いのないキャストを行うには、アップキャスト（親クラスへのキャスト）の際は、キャストを明示せずに親クラスの変数に代入するか、「static_cast」を用いる。ダウンキャスト（子クラスへのキャスト）の際は、「dynamic_cast」を用いる。

なお、「dynamic_cast」を用いる場合、コンパイラの設定で「RTTI」が有効になっている必要がある。

▼ クラス内 new / delete

クラス（もしくは構造体）は、そのクラスのインスタンス生成／破棄専用の new 演算子／delete 演算子を定義することができる。

これを用いると、前述の配列と多重継承の delete の問題を解消できる。

前述のサンプルは、delete を「delete 演算子」としてではなく、「配置 delete 関数」として呼び出していたために、受け渡したポインタもそのままで、デストラクタも呼び出されなかった。

クラス内 delete を用いることで、「配置 delete 関数」としてではなく、本来の「delete 演算子」として扱うことができる。

クラス内 new / delete の挙動確認用サンプル：

【テスト用クラス定義】

```
//--------------------
//インスタンス用バッファ
static char s_buffForTestClass2[64];
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest2
{
public:
	//オペレータ
	//クラス内 new/delete
	//new
	//※自クラスのインスタンス専用の new/delete
	//※普通の new/delete 演算子でインスタンスを生成／破棄すると呼び出される
	//※配置 new/配置 delete も使用可（配置 delete の際は CTest2::operator delete(p, xxx) のようにクラス名を
	//  明示する必要あり）
	//※静的 operator である必要があるが、static を付けなくても暗黙的に static 扱いになる
	//単体 new
	static void* operator new(const std::size_t size) throw()
	{
		printf("CTest2::new(size=%d)¥n", size);
		return s_buffForTestClass2;//インスタンス用バッファを返す
	}
	//配列 new
	static void* operator new[](const std::size_t size) throw()
	{
		printf("CTest2::new[](size=%d)¥n", size);
		return s_buffForTestClass2;// インスタンス用バッファを返す
	}
#if 0
	//delete
	//※クラス内 delete には 2 種類のスタイルがある
	//※これは標準的なスタイル
	//単体 delete
	static void operator delete(void* p) throw()
	{
		printf("CTest2::delete(p=0x%p)¥n", p);
		//なにもしない
	}
	//配列 delete
	static void operator delete[](void* p) throw()
	{
```

```
            printf("CTest2::delete[](p=0x%p)¥n", p);
            //なにもしない
      }
#endif
      //delete
      //※クラス内 delete では、第 2 引数に size_t を取る形も標準スタイルの一つ（非配置 delete）
      //※2 種類とも定義したら、operator delete(void*) の方が優先
      //単体 delete
      static void operator delete(void* p, const std::size_t size) throw()
      {
            printf("CTest2::delete(p=0x%p, size=%d)¥n", p, size);
            //なにもしない
      }
      //配列 delete
      static void operator delete[](void* p, const std::size_t size) throw()
      {
            printf("CTest2::delete[](p=0x%p, size=%d)¥n", p, size);
            //なにもしない
      }
public:
      //デフォルトコンストラクタ
      CTest2() :
            CTest2("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
            //m_name("(default)")//C++11 が使えない場合は普通に初期化
      {
            printf("CTest2::DefaultConstructor : name=¥"%s¥"¥n", m_name);
      }
      //コンストラクタ
      CTest2(const char* name) :
            m_name(name)
      {
            printf("CTest2::Constructor : name=¥"%s¥"¥n", m_name);
      }
      //デストラクタ
      //※多重継承のテストのために virtual 化
      virtual ~CTest2()
      {
            printf("CTest2::Destructor : name=¥"%s¥"¥n", m_name);
      }
protected:
      //フィールド
      const char* m_name;//名前
      int m_dummy;//ダミー
};
```

【多重継承テスト用クラス定義】

```
//--------------------
//テスト用クラス（多重継承テスト用）
class CTest2ex
{
public:
      //オペレータ
      //クラス内 new
      static void* operator new(const std::size_t size) throw()
      {
            printf("CTest2ex::new(size=%d)¥n", size);
            return s_buffForTestClass2;//インスタンス用バッファを返す
      }
      //クラス内 delete
      static void operator delete(void* p) throw()
      {
            printf("CTest2ex::delete(p=0x%p)¥n", p);
            //なにもしない
      }
public:
```

```
	//コンストラクタ
	CTest2ex()
	{
		printf("CTest2ex::Constructor¥n");
	}
	//デストラクタ
	virtual ~CTest2ex()
	{
		printf("CTest2ex::Destructor¥n");
	}
protected:
	//フィールド
	int m_dummy;//ダミー
};
//--------------------
//テスト用クラス（多重継承）
class CDerivedTest2 : public CTest2, public CTest2ex
{
public:
	//オペレータ
	//クラス内 new
	static void* operator new(const std::size_t size) throw()
	{
		printf("placement CDerivedTest2::new(size=%d)¥n", size);
		return s_buffForTestClass2;//インスタンス用バッファを返す
	}
	//クラス内 delete
	static void operator delete(void* p) throw()
	{
		printf("placement CDerivedTest2::delete(p=0x%p)¥n", p);
		//なにもしない
	}
public:
	//デフォルトコンストラクタ
	CDerivedTest2() :
		CTest2(),
		CTest2ex()
	{
		printf("CDerivedTest2::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CDerivedTest2(const char* name) :
		CTest2(name),
		CTest2ex()
	{
		printf("CDerivedTest2::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CDerivedTest2() override
	{
		printf("CDerivedTest2::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	int m_dummy;//ダミー
};
```

【テスト】

```
//--------------------
//テスト
void test2()
{
	{
		printf("----------クラス¥n");
		printf("s_buffForTestClass2=0x%p¥n", s_buffForTestClass2);
```

```
            CTest2* obj_p = new CTest2("テスト 2");//普通の new 演算子として使える
            printf("obj_p=0x%p¥n", obj_p);
            delete obj_p;//普通の delete 演算子として使える
        }
        {
            printf("----------クラスの配列¥n");
            printf("s_buffForTestClass2=0x%p¥n", s_buffForTestClass2);
            CTest2* obj_p = new CTest2[3]; //普通の new[]演算子として使える
            printf("obj_p=0x%p¥n", obj_p);
            printf("*reinterpret_cast<int*>(s_buffForTestClass2)=%d¥n", *reinterpret_cast<int*>(s_buffForTestClass2));
                                        //デストラクタを持った型の配列は、バッファの先頭に要素数が格納されている
            delete[] obj_p;//普通の delete[]演算子として使える
        }
        {
            printf("----------多重継承クラス¥n");
            printf("s_buffForTestClass2=0x%p¥n", s_buffForTestClass2);
            CDerivedTest2* obj_p = new CDerivedTest2("テスト 2 多重継承");
            CTest2* parent_p = obj_p;//親 1 にキャスト
            CTest2ex* parent_ex_p = obj_p;//親 2 にキャスト
            printf("obj_p=0x%p, parent_p=0x%p, parent_ex_p=0x%p¥n", obj_p, parent_p, parent_ex_p);
            //delete obj_p;//子として削除
            //delete parent_p;//親 1 として削除
            delete parent_ex_p;//親 2 として削除
        }
}
```

↓（実行結果）

```
----------クラス
s_buffForTestClass2=0x00E4F680                  ←バッファの先頭アドレス
CTest2::new(size=12)
CTest2::Constructor : name="テスト 2"
obj_p=0x00E4F680                                ←new により、固定バッファ割り当て
CTest2::Destructor : name="テスト 2"            ←【重要】delete により、デストラクタが自動的に呼び出されている
CTest2::delete(p=0x00E4F680, size=12)
----------クラスの配列
s_buffForTestClass2=0x00E4F680                  ←バッファの先頭アドレス
CTest2::new[](size=40)                          ←new に渡されたサイズが 4 バイト多い（12 * 3 + 4）
CTest2::Constructor : name="(default)"
CTest2::DefaultConstructor : name="(default)"
CTest2::Constructor : name="(default)"
CTest2::DefaultConstructor : name="(default)"
CTest2::Constructor : name="(default)"
CTest2::DefaultConstructor : name="(default)"
obj_p=0x00E4F684                                ←new が返したポインタが 4 バイトシフトしている
*reinterpret_cast<int*>(s_buffForTestClass2)=3  ←バッファの先頭を確認すると要素数が格納されている
CTest2::Destructor : name="(default)"
CTest2::Destructor : name="(default)"
CTest2::Destructor : name="(default)"
CTest2::delete[](p=0x00E4F680, size=12)         ←【重要】delete に本来のポインタが渡されている
----------多重継承クラス
s_buffForTestClass2=0x00E4F680                  ←バッファの先頭アドレス
CDerivedTest2::new(size=24)
CTest2::Constructor : name="テスト 2 多重継承"
CTest2ex::Constructor
CDerivedTest2::Constructor : name="テスト 2 多重継承"
obj_p=0x00E4F680, parent_p=0x00E4F680, parent_ex_p=0x00E4F68C     ←多重継承のキャストで変化するポインタ
CDerivedTest2::Destructor : name="テスト 2 多重継承"   ←【重要】delete により、デストラクタが自動的に呼び出されている
CTest2ex::Destructor                                   ←（同上）
CTest2::Destructor : name="テスト 2 多重継承"          ←（同上）
CDerivedTest2::delete(p=0x00E4F680)             ←【重要】親クラスのポインタで delete したが、子クラスの方の
                                                         delete 処理が本来のポインタで実行されている
```

● 【補足①】クラス内 new / delete の適用範囲

念のため、クラス内 new / delete の適用範囲について補足する。

クラス内 new / delete は、あくまでも「operator+()」などと同じく、そのクラス自身に対するオペレータ（new 演算子と delete 演算子）のオーバーロードである。

「operator new()」「operatror delete()」をクラス内のメンバー関数と思うと、クラス内の処理に記述された「new」「delete」に反応しそうにも思えてしまうが、そうではなく、（主にクラスの外部で）そのクラスのインスタンスを生成／破棄するときに呼び出されるものである。

クラス内スコープの new / delete（配置 new／配置 delete 含む）を定義したくなることがあるが、そういう使い方はできないので注意。

● 【補足②】局所 new / delete は事実上使用不可

ネームスペース内に局所的に定義された new / delete は使い物にならず、事実上使用できない。

ネームスペース内に new / delete を定義することは可能であり、「namespace::operator」を明示して、関数として呼び出すことはできる。

その場合、演算子扱いにならないので、配置 delete で説明したとおり、適切なポインタ変換もデストラクタ呼び出しも行われない。new はさらに使い勝手が悪く、関数として呼び出すとコンストラクタが実行されず、必要メモリのサイズも直接渡さなければならない。

- なお、同様に「classname::operator new / delete」と明示すれば、クラス内の new / delete も同様に呼び出せる。

ネームスペースに定義した new / delete の挙動確認用サンプル：

【ネームスペースに new/delete を定義】

```
//--------------------
//ネームスペース
namespace test2_ns
{
	//ネームスペース内バッファ
	char s_buff[32];

	//ネームスペースに定義した new
	void* operator new(const std::size_t size) throw()
	{
		printf("test2_ns::new size=%d¥n", size);
		return s_buff;
	}
	//ネームスペースに定義した delete
	void operator delete(void* p) throw()
	{
```

```
            printf("test2_ns::delete p=0x%p¥n", p);
        }
}
```

【テスト】(「クラス内 new/delete」のサンプルを流用)

```
//--------------------
//テスト
void test2()
{
    {
        printf("----------ネームスペースの new/delete¥n");
        CTest2* obj_p = (CTest2*)test2_ns::operator new(sizeof(CTest2));//ネームスペースに定義した new
        ::new(obj_p)CTest2("テスト 2");//配置 new で明示的なコンストラクタ呼び出し
        obj_p->~CTest2();//明示的なデストラクタ呼び出し
        test2_ns::operator delete(obj_p);//ネームスペースに定義した delete
    }
}
```

↓ (実行結果)

```
----------ネームスペースの new/delete
test2_ns::new size=12
CTest2::Constructor : name="テスト 2"
CTest2::Destructor : name="テスト 2"
test2_ns::delete p=0x00E4F660
```

なお、この処理は Visual C++ 2013 では動作するが、GCC ではコンパイルエラーとなった。(Cygwin + GCC 4.8.2 で確認)

ネームスペースに定義した new / delete によるコンパイルエラー :

```
main.cpp:529:43: エラー: 'void* test2_ns::operator new(std::size_t)' may not be declared within a namespace
  void* operator new(const std::size_t size)
                                           ^

main.cpp:535:30: エラー: 'void test2_ns::operator delete(void*)' may not be declared within a namespace
  void operator delete(void* p)
```

何にしても、ネームスペース内の new / delete は「使えない」と考えるべきである。

▼ グローバル new / delete の置き換え

「クラス内 new / delete」で説明したとおり、配列と多重継承の delete の問題は、delete 処理を「配置 delete 関数」としてではなく、本来の「delete 演算子」として扱うことで解消できる。これは、グローバル new / delete を置き換えることでも対応できる。

グローバル new / delete の置き換え確認用サンプル :

【インスタンス用バッファ定義】

```
//--------------------
//インスタンス用バッファ
static char s_buffForTestClass2[64];
```

【グローバル new / delete オペレータ定義】

```
//--------------------
//グローバル new/delete テスト
//単体 new
void* operator new(const std::size_t size) throw()
{
    printf("new(size=%d)¥n", size);
```

```
	return s_buffForTestClass2;//インスタンス用バッファを返す
}
//配列 new
void* operator new[](const std::size_t size) throw()
{
	printf("new[](size=%d)¥n", size);
	return s_buffForTestClass2;//インスタンス用バッファを返す
}
//単体 delete
void operator delete(void* p) throw()
{
	printf("delete(p=0x%p)¥n", p);
	//なにもしない
}
//配列 delete
void operator delete[](void* p) throw()
{
	printf("delete[](p=0x%p)¥n", p);
	//なにもしない
}
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest2
{
public:
	//デフォルトコンストラクタ
	CTest2() :
		CTest2("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
		//m_name("(default)")//C++11 が使えない場合は普通に初期化
	{
		printf("CTest2::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest2(const char* name) :
		m_name(name)
	{
		printf("CTest2::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	//※多重継承のテストのために virtual 化
	virtual ~CTest2()
	{
		printf("CTest2::Destructor : name=¥"%s¥"¥n", m_name);
	}
protected:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【多重継承テスト用クラス定義】

```
//--------------------
//テスト用クラス（多重継承テスト用）
class CTest2ex
{
public:
	//コンストラクタ
	CTest2ex()
	{
		printf("CTest2ex::Constructor¥n");
	}
	//デストラクタ
	virtual ~CTest2ex()
```

```
	{
		printf("CTest2ex::Destructor¥n");
	}
protected:
	//フィールド
	int m_dummy;//ダミー
};
//--------------------
//テスト用クラス（多重継承）
class CDerivedTest2 : public CTest2, public CTest2ex
{
public:
	//デフォルトコンストラクタ
	CDerivedTest2() :
		CTest2(),
		CTest2ex()
	{
		printf("CDerivedTest2::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CDerivedTest2(const char* name) :
		CTest2(name),
		CTest2ex()
	{
		printf("CDerivedTest2::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CDerivedTest2() override
	{
		printf("CDerivedTest2::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	int m_dummy;//ダミー
};
```

【テスト】

```
//--------------------
//テスト
void test2()
{
	{
		printf("----------クラス¥n");
		printf("s_buffForTestClass2=0x%p¥n", s_buffForTestClass2);
		CTest2* obj_p = new CTest2("テスト 2");//普通の new 演算子として使える
		printf("obj_p=0x%p¥n", obj_p);
		delete obj_p;//普通の delete 演算子として使える
	}
	{
		printf("----------クラスの配列¥n");
		printf("s_buffForTestClass2=0x%p¥n", s_buffForTestClass2);
		CTest2* obj_p = new CTest2[3];//普通の new[]演算子として使える
		printf("obj_p=0x%p¥n", obj_p);
		printf("*reinterpret_cast<int*>(s_buffForTestClass2)=%d¥n", *reinterpret_cast<int*>(s_buffForTestClass2));
                                    //デストラクタを持った型の配列は、バッファの先頭に要素数が格納されている
		delete[] obj_p;//普通の delete[]演算子として使える
	}
	{
		printf("----------多重継承クラス¥n");
		printf("s_buffForTestClass2=0x%p¥n", s_buffForTestClass2);
		CDerivedTest2* obj_p = new CDerivedTest2("テスト 2 多重継承");
		CTest2* parent_p = obj_p;//親 1 にキャスト
		CTest2ex* parent_ex_p = obj_p;//親 2 にキャスト
		printf("obj_p=0x%p, parent_p=0x%p, parent_ex_p=0x%p¥n", obj_p, parent_p, parent_ex_p);
		//delete obj_p;//子として削除
```

```
            //delete parent_p;//親１として削除
            delete parent_ex_p;//親２として削除
    }
}
```

↓（実行結果）※クラス内 new/delete と全く同じ結果

```
----------クラス
s_buffForTestClass2=0x00E4F680                  ←バッファの先頭アドレス
new(size=12)
CTest2::Constructor : name="テスト２"
obj_p=0x00E4F680                                ←new により、固定バッファ割り当て
CTest2::Destructor : name="テスト２"            ←【重要】delete により、デストラクタが自動的に呼び出されている
delete(p=0x00E4F680)
----------クラスの配列
s_buffForTestClass2=0x00E4F680                  ←バッファの先頭アドレス
new[](size=40)                                  ←new に渡されたサイズが４バイト多い（12 * 3 + 4）
CTest2::Constructor : name="(default)"
CTest2::DefaultConstructor : name="(default)"
CTest2::Constructor : name="(default)"
CTest2::DefaultConstructor : name="(default)"
CTest2::Constructor : name="(default)"
CTest2::DefaultConstructor : name="(default)"
obj_p=0x00E4F684                                ←new が返したポインタが４バイトシフトしている
*reinterpret_cast<int*>(s_buffForTestClass2)=3  ←バッファの先頭を確認すると要素数が格納されている
CTest2::Destructor : name="(default)"
CTest2::Destructor : name="(default)"
CTest2::Destructor : name="(default)"
delete[](p=0x00E4F680)                          ←【重要】delete に本来のポインタが渡されている
----------多重継承クラス
s_buffForTestClass2=0x00E4F680                  ←バッファの先頭アドレス
new(size=24)
CTest2::Constructor : name="テスト２多重継承"
CTest2ex::Constructor
CDerivedTest2::Constructor : name="テスト２多重継承"
obj_p=0x00E4F680, parent_p=0x00E4F680, parent_ex_p=0x00E4F68C     ←多重継承のキャストで変化するポインタ
CDerivedTest2::Destructor : name="テスト２多重継承"   ←【重要】delete により、デストラクタが自動的に呼び出されている
CTest2ex::Destructor                                  ←（同上）
CTest2::Destructor : name="テスト２多重継承"          ←（同上）
delete(p=0x00E4F680)                            ←【重要】親クラスのポインタで delete したが、
                                                          delete 処理が本来のポインタで実行されている
```

▼ 自作メモリアロケータ作成時の注意点

自作メモリアロケータを作成した場合、それを暗黙的に使用するための配置 new と配置 delete を作成することはよくあるやり方である。

ここまでに説明したように、配置 new を使用すると、配置 new 内で確保したメモリのポインタが正確に配置 delete に渡されることを保証できない問題がある。これは、メモリアロケータを「使う側の問題」であるため、防ぎようがない。

この対策として、最低限メモリリークを防ぐために、「delete（free）処理に渡されたポインタが、new（allocate）処理で確保したメモリの先頭ポインタでなくとも、その範囲内のポインタなら削除可能とする」という仕様の導入が考えられる。

別紙の「ゲーム制御のためのメモリ管理方針」および「柔軟性を追求したメモリ管理システム」で策定したメモリ管理システムでは、メモリノードの管理情報をデータ部と分離し

ており、ポインタから管理情報を検索することができるようにしているため、このような削除処理を導入することも不可能ではない。ただし、このような削除処理は、誤った削除要求と区別できなくなる点にも要注意。

問題解決の他の方法として、「配置 new」はともかくとして、「「配置 delete」を一切使わない」という方針にするのもよい。

配置 delete を使わずに自作アロケータを使用するためには、後述する「共通アロケータインターフェース」を便利に使うことができる。

■ 配置 new / 配置 delete の応用

配置 new / 配置 delete の応用例として、固定バッファシングルトンを説明する。

シングルトンは、別紙の「効率化と安全性のためのロック制御」でも説明しており、そこではスレッドセーフに重点を置いた実装を示している。本書では、シングルトンのメモリ操作に限定して説明する。

▼ 固定バッファシングルトン①：専用シングルトン：単純タイプ

まずは最も単純なシングルトンを説明する。

new / delete を使わず、static 変数で最初からインスタンスを持つ単純なタイプ。コンストラクタは main 関数よりも前に実行される。単純に作れるが融通が利かない。

専用シングルトン（単純タイプ）のサンプル：

【シングルトンクラス定義】

```
//--------------------
//専用シングルトン：単純タイプ
class CTest3Singleton1
{
public:
	//静的アクセッサ
	static CTest3Singleton1* getSigleton(){ return &m_singleton; }//シングルトンを取得
public:
	//アクセッサ
	int getData() const { return m_data; }//データを取得
	void setData(const int data){ m_data = data; }//データを更新
public:
	//コンストラクタ
	CTest3Singleton1() :
		m_data(0)
	{
		printf("CTest3Singleton1::Constructor : m_data=%d¥n", m_data);
	}
	//デストラクタ
	~CTest3Singleton1()
	{
```

```
            printf("CTest3Singleton1::Destructor : m_data=%d¥n", m_data);
      }
private:
      //フィールド
      int m_data;//データ
      static CTest3Singleton1 m_singleton;//シングルトンインスタンス
};
```

【シングルトンクラスの静的変数のインスタンス化】

```
//--------------------
//シングルトン静的変数のインスタンス化
CTest3Singleton1 CTest3Singleton1::m_singleton;
```

【テスト（関数に処理を分割）】

```
//--------------------
//テスト
//シングルトンのデータ表示
void printData_typeA1()
{
      CTest3Singleton1* singleton_p = CTest3Singleton1::getSigleton();
      printf("singleton_p->getData()=%d¥n", singleton_p->getData());
}
//シングルトンのデータ更新
void updateData_typeA1()
{
      CTest3Singleton1* singleton_p = CTest3Singleton1::getSigleton();
      singleton_p->setData(123);
}
```

【テスト（メイン）】

```
//--------------------
//テスト
void test3()
{
      {
            printf("----------専用シングルトン：単純タイプ¥n");
            printData_typeA1();//データ表示
            updateData_typeA1();//データ更新
            printData_typeA1();//データ表示
      }
}
```

↓（実行結果）

```
CTest3Singleton1::Constructor : m_data=0          ←メイン関数実行前に実行
----------専用シングルトン：単純タイプ
singleton_p->getData()=0
singleton_p->getData()=123                        ←別関数で行った更新が正しく反映されている

CTest3Singleton1::Destructor : m_data=123         ←メイン関数終了後に実行
```

▼ 固定バッファシングルトン②：専用シングルトン：インスタンス操作タイプ

単純タイプのシングルトンは、main 関数実行前にコンストラクタが呼び出されてしまう。こうした処理が多数あると、処理の順序性が保証しきれず、問題になる場合がある。そこで、インスタンス生成のタイミングを自分で制御できる形に修正する。

クラスの静的メンバーにはバッファとポインタを持ち、自クラスのインスタンスを操作するために、プライベートな new / delete オペレータを実装する。

専用シングルトン（インスタンス操作タイプ）のサンプル：

【シングルトンクラス定義】

```
//--------------------
//専用シングルトン：インスタンス操作タイプ
class CTest3Singleton2
{
public:
	//静的アクセッサ
	static CTest3Singleton2* getSigleton()//シングルトンを取得
	{
		create();//インスタンスがなければ生成
		return m_singleton;
	}
public:
	//アクセッサ
	int getData() const { return m_data; }//データを取得
	void setData(const int data){ m_data = data; }//データを更新
private:
	//オペレータ (private)
	static void* operator new(const std::size_t) throw() { return m_singletonBuff; }//new 演算子
	static void operator delete(void*) throw() {}//delete 演算子
private:
	//静的メソッド (private)
	//シングルトンインスタンスの生成
	static void create()
	{
		if (m_singleton)
			return;
		m_singleton = new CTest3Singleton2();
	}
public:
	//静的メソッド
	//シングルトンインスタンスの明示的な破棄
	static void destroy()
	{
		if (!m_singleton)
			return;
		delete m_singleton;
		m_singleton = nullptr;
	}
public:
	//コンストラクタ
	CTest3Singleton2() :
		m_data(0)
	{
		printf("CTest3Singleton2::Constructor : m_data=%d¥n", m_data);
	}
	//デストラクタ
	~CTest3Singleton2()
	{
		printf("CTest3Singleton2::Destructor : m_data=%d¥n", m_data);
	}
private:
	//フィールド
	int m_data;//データ
	static CTest3Singleton2* m_singleton;//シングルトンインスタンス参照
	static unsigned char m_singletonBuff[];//シングルトンインスタンス用バッファ
};
```

【シングルトンクラスの静的変数のインスタンス化】

```
//--------------------
//シングルトン静的変数のインスタンス化
CTest3Singleton2* CTest3Singleton2::m_singleton = nullptr;
unsigned char CTest3Singleton2::m_singletonBuff[sizeof(CTest3Singleton2)];//シングルトンインスタンス用バッファ
```

【テスト（関数に処理を分割）】

```
//--------------------
//テスト
//シングルトンのデータ表示
void printData_typeA2()
{
	CTest3Singleton2* singleton_p = CTest3Singleton2::getSigleton();
	printf("singleton_p->getData()=%d¥n", singleton_p->getData());
}
//シングルトンのデータ更新
void updateData_typeA2()
{
	CTest3Singleton2* singleton_p = CTest3Singleton2::getSigleton();
	singleton_p->setData(123);
}
//シングルトンのインスタンスを明示的に破棄
void deleteIntentionally_typeA2()
{
	CTest3Singleton2::destroy();
}
```

【テスト（メイン）】

```
//--------------------
//テスト
void test3()
{
	{
		printf("----------専用シングルトン：インスタンス操作タイプ¥n");
		printData_typeA2();//データ表示
		updateData_typeA2();//データ更新
		printData_typeA2();//データ表示
		deleteIntentionally_typeA2();//明示的なインスタンス破棄
		printData_typeA2();//データ表示
	}
}
```

↓（実行結果）

```
----------専用シングルトン：インスタンス操作タイプ
CTest3Singleton2::Constructor : m_data=0  ←初めてシングルトンにアクセスした時に自動的にインスタンス生成
singleton_p->getData()=0
singleton_p->getData()=123                ←別関数で行った更新が正しく反映されている
CTest3Singleton2::Destructor : m_data=123 ←明示的なインスタンス破棄
CTest3Singleton2::Constructor : m_data=0  ←再アクセス時にまた自動的にインスタンス生成
singleton_p->getData()=0
```

▼ 固定バッファシングルトン③：汎用シングルトン

インスタンス操作タイプのシングルトンを発展させて、汎用的なテンプレートクラスにする。

シングルトンの機能を構成する汎用的なテンプレートクラスと、シングルトンとしての対象クラスを分けて扱い、シングルトン対象クラスのインスタンス生成に配置 new を用いる。

シングルトンとしての機能を完全に分離するため、シングルトン対象クラスは、シングルトンとして利用されることを気にせずにシンプルに構成できる。

汎用シングルトンテンプレートクラスのサンプル：

【汎用シングルトンテンプレートクラス定義】

```
//--------------------
//汎用固定バッファシングルトンテンプレートクラス
template<class T>
class CSingleton
{
	//配置 new テンプレートをフレンド化
	template<class U>
	friend void* operator new(const std::size_t size, CSingleton<U>& singleton) throw();
public:
	//オペレータ（シングルトン本体のプロキシー）
	T* operator->(){ return m_singleton; }
public:
	//メソッド
	//シングルトンインスタンスの明示的な破棄
	void destroy();
public:
	//コンストラクタ
	CSingleton();
private:
	//フィールド
	static T* m_singleton;//シングルトン本体参照
	static unsigned char m_singletonBuff[];//シングルトン本体用バッファ
};

//--------------------
//テンプレート配置 new
template<class T>
void* operator new(const std::size_t size, CSingleton<T>& singleton) throw()
{
	return singleton.m_singletonBuff;//汎用シングルトンテンプレートクラス内のバッファを返す
}
//テンプレート配置 delete
template<class T>
void operator delete(void* p, CSingleton<T>& singleton) throw()
{
	//なにもしない
}

//--------------------
//汎用固定バッファシングルトンテンプレートクラス：実装部
//明示的な削除
template<class T>
void CSingleton<T>::destroy()
{
	if (!m_singleton)
		return;
	m_singleton->~T();//デストラクタ呼び出し
	operator delete(m_singleton, *this);//配置 delete 呼び出し（なくてもよい）
	m_singleton = nullptr;
}
//コンストラクタ
template<class T>
CSingleton<T>::CSingleton()
{
	//まだ初期化していなければ初期化
	if (!m_singleton)
		m_singleton = new(*this)T();
}
```

【シングルトンクラスの静的変数のインスタンス化】

```
//--------------------
//シングルトン静的変数のインスタンス化
```

```
template<class T>
T* CSingleton<T>::m_singleton = nullptr;//シングルトン参照
template<class T>
unsigned char CSingleton<T>::m_singletonBuff[sizeof(T)];//シングルトン用バッファ
```

【シングルトン対象クラス定義】

```
//--------------------
//テスト用クラス
class CTest3
{
public:
	//アクセッサ
	int getData() const { return m_data; }//データ取得
	void setData(const int data){ m_data = data; }//データ更新
public:
	//コンストラクタ
	CTest3() :
		m_data(0)
	{
		printf("CTest3::Constructor : m_data=%d¥n", m_data);
	}
	//デストラクタ
	~CTest3()
	{
		printf("CTest3::Destructor : m_data=%d¥n", m_data);
	}
private:
	//フィールド
	int m_data;//データ
};
```

【テスト（関数に処理を分割)】

```
//--------------------
//テスト
//シングルトンのデータ表示
void printData_typeB()
{
	CSingleton<CTest3> singleton;
	printf("singleton->getData()=%d¥n", singleton->getData());
}
//シングルトンのデータ更新
void updateData_typeB()
{
	CSingleton<CTest3> singleton;
	singleton->setData(123);
}
//シングルトンのインスタンスを明示的に破棄
void deleteIntentionally_typeB()
{
	CSingleton<CTest3> singleton;
	singleton.destroy();
}
```

【テスト（メイン)】

```
//--------------------
//テスト
void test3()
{
	{
		printf("----------汎用シングルトン¥n");
		printData_typeB();//データ表示
		updateData_typeB();//データ更新
		printData_typeB();//データ表示
		deleteIntentionally_typeB();//明示的なインスタンス破棄
		printData_typeB();//データ表示
```

```
    }
}
```

↓（実行結果）

```
----------汎用シングルトン
CTest3::Constructor : m_data=0      ←初めてシングルトンにアクセスした時に自動的にインスタンス生成
singleton->getData()=0
singleton->getData()=123            ←別関数で行った更新が正しく反映されている
CTest3::Destructor : m_data=123     ←明示的なインスタンス破棄
CTest3::Constructor : m_data=0      ←再アクセス時にまた自動的にインスタンス生成
singleton->getData()=0
```

■ スタックアロケータ

スタックアロケータは、非常に軽量で、無駄なく高速にメモリを操作することができるので、ゲーム制作にはとても重用される。

ただし、「メモリを個別に削除できない」という大きな問題もあるため、その導入は決して易しいものではない。

スタックアロケータを有効活用する手法は、後述する「フレームヒープ」で説明するが、本節ではスタックアロケータの仕組みを説明する。

▼ スタックアロケータの仕組み

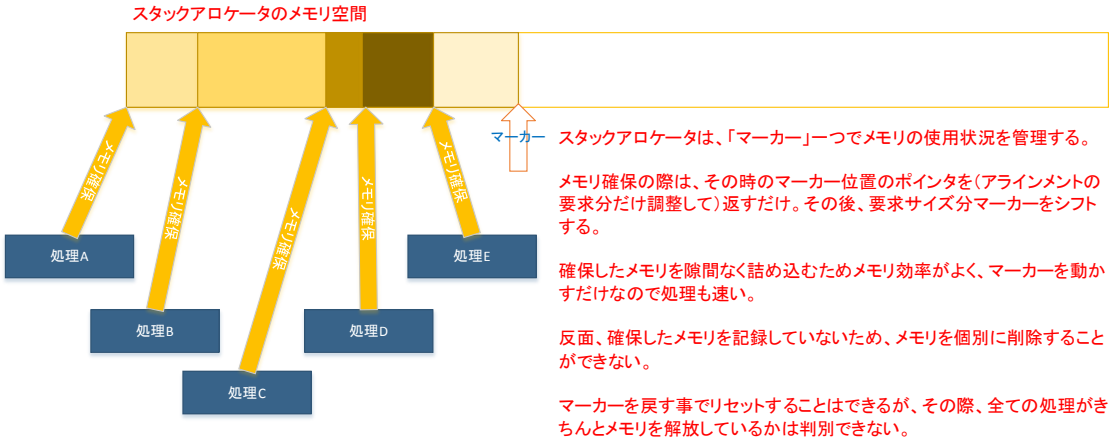
スタックアロケータは、非常に単純な仕組みである。

一塊の大きなバッファと、「どこまでバッファを使用しているか」を示すマーカーのみで構成する。

処理はメモリ確保の要求に応じてマーカーをずらしていくだけである。

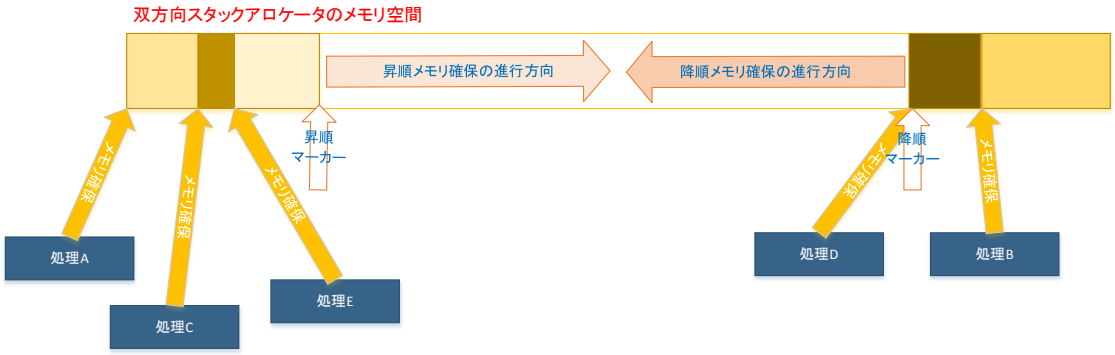
削除に対応しない代わりに、マーカーを任意の位置に強制的に戻すことができる。

スタックアロケータの仕組み：



スタックアロケータをもう少し融通の利いたものにするために、メモリの前後から二つのマーカーを用いて管理する手法もある。

双方向スタックアロケータの仕組み：



▼ スタックアロケータのサンプル

スタックアロケータのサンプルプログラムを示す。

サンプルでは、ローカル変数領域（スタック領域）のバッファを使用した処理としている。

メモリの位置とアラインメントの計算に「uintptr_t 型」を使用しているのがポイントの一つ（サンプルプログラムはアラインメント計算に対応している）。uint_ptr_t 型は、C++99 からの仕様で、ポインタを整数として計算する時に用いる型。「#include <stdint.h>」により使用できる。負の差分を得たい場合は符号付きの「intptr_t 型」も使える。この型を使う事により、32bit と 64bit で共用可能なコードを書くことができる。

なお、スタックアロケータは、後述する「フレームヒープ」でも活用する。

スタックアロケータのサンプル：

【スタックアロケータインターフェースクラス定義】※双方向版とインターフェースを共通化するためのもの

```
//------------------
//スタックアロケータインターフェースクラス
class IStackAllocator
{
public:
	//定数
	static const std::size_t DEFAULT_ALIGN = sizeof(int);//デフォルトのアラインメント
	//スタック順
	enum E_ORDERED
	{
		DEFAULT = 0,//デフォルト
		NORMAL = 1,//正順
		REVERSE = -1,//逆順
	};
public:
	//型
	typedef unsigned char byte;//バッファ用
	typedef uintptr_t marker_t;//スタックマーカー型
```

```
public:
	//アクセッサ
	virtual std::size_t getTotal() const = 0;//全体のメモリ量を取得
	virtual std::size_t getUsed() const = 0;//使用中のメモリ量を取得
	virtual std::size_t getRemain() const = 0;//残りのメモリ量を取得
	virtual const byte* getNowPtr() const = 0;//現在のバッファ位置を取得
	virtual marker_t getMarker() const = 0;//現在のマーカーを取得
public:
	//メソッド
	//メモリ確保
	virtual void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN) = 0;
	//メモリを以前のマーカーに戻す
	//※マーカー指定版
	virtual void back(const marker_t marker) = 0;
	//メモリを以前のマーカーに戻す
	//※ポインタ指定版
	virtual void back(const void* p) = 0;
	//メモリ破棄（正順）
	virtual void clear() = 0;
	//メモリ破棄（全て）
	virtual void clearAll() = 0;
public:
	//デストラクタ
	virtual ~IStackAllocator(){}
};
//定数のエイリアス
static const IStackAllocator::E_ORDERED DSA_DEFAULT = IStackAllocator::DEFAULT;//デフォルト
static const IStackAllocator::E_ORDERED DSA_NORMAL = IStackAllocator::NORMAL;//正順
static const IStackAllocator::E_ORDERED DSA_REVERSE = IStackAllocator::REVERSE;//逆順
```

【スタックアロケータクラス定義】

```
//--------------------
//スタックアロケータクラス
//※非スレッドセーフ（処理速度優先）
//※インターフェースを実装するが、高速化のために virtual メソッドに頼らず操作可能
class CStackAllocator : public IStackAllocator
{
public:
	//アクセッサ
	std::size_t getTotal() const override { return m_buffSize; }//全体のメモリ量を取得
	std::size_t getUsed() const override { return m_used; }//使用中のメモリ量を取得
	std::size_t getRemain() const override { return m_buffSize - m_used; }//残りのメモリ量を取得
	const byte* getBuff() const { return m_buffPtr; }//バッファ取得を取得
	const byte* getNowPtrN() const { return m_buffPtr + m_used; }//現在のバッファ位置（正順）を取得
	const byte* getNowPtr() const override { return getNowPtrN(); }//現在のバッファ位置を取得
	marker_t getMarkerN() const { return m_used; }//現在のマーカー（正順）を取得
	marker_t getMarker() const override { return getMarkerN(); }//現在のマーカーを取得
public:
	//メソッド
	//メモリ確保（正順）
	void* allocN(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
	{
		const uintptr_t now_ptr = reinterpret_cast<uintptr_t>(m_buffPtr) + m_used;//現在のポインタ位置算出
		const uintptr_t align_diff = align > 0 ? now_ptr % align == 0 ? 0 : align - now_ptr % align : 0;
                                                                                //アラインメント計算
		const marker_t next_used = m_used + align_diff + size;//次のマーカー算出
		if (next_used > m_buffSize)//メモリオーバーチェック（符号なしなので、範囲チェックは大判定のみで OK）
		{
			printf("stack overflow!(size=%d+align=%d, remain=%d)¥n", size, align_diff, m_buffSize - m_used);
			return nullptr;
		}
		const uintptr_t alloc_ptr = now_ptr + align_diff;//メモリ確保アドレス算出
		m_used = next_used;//マーカー更新
		return reinterpret_cast<void*>(alloc_ptr);
	}
```

```
	//メモリ確保
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN) override
	{
		return allocN(size, align);
	}
	//メモリを以前のマーカーに戻す（正順）
	//※マーカー指定版
	void backN(const marker_t marker)
	{
		if (marker >= m_used)//符号なしなので、範囲チェックは大判定のみで OK
			return;
		m_used = marker;
	}
	//メモリを以前のマーカーに戻す
	//※マーカー指定版
	void back(const marker_t marker) override
	{
		return backN(marker);
	}
	//メモリを以前のマーカーに戻す（正順）
	//※ポインタ指定版
	void backN(const void* p)
	{
		const marker_t marker = reinterpret_cast<uintptr_t>(p) - reinterpret_cast<uintptr_t>(m_buffPtr);
		back(marker);
	}
	//メモリを以前のマーカーに戻す
	//※ポインタ指定版
	void back(const void* p) override
	{
		backN(p);
	}
	//メモリ破棄（正順）
	void clearN()
	{
		m_used = 0;//マーカーをリセット
	}
	//メモリ破棄（正順）
	void clear() override
	{
		clearN();
	}
	//メモリ破棄（全て）
	void clearAll() override
	{
		clearN();
	}
public:
	//コンストラクタ
	CStackAllocator(void* buff_p, const std::size_t buff_size) :
		m_buffPtr(reinterpret_cast<byte*>(buff_p)),//バッファ先頭アドレス
		m_buffSize(buff_size),//バッファサイズ
		m_used(0)//マーカー
	{}
	//デストラクタ
	~CStackAllocator() override
	{}
private:
	//フィールド
	byte* m_buffPtr;//バッファ先頭アドレス
	const std::size_t m_buffSize;//バッファサイズ
	marker_t m_used;//マーカー（正順）
};
```

【スタックアロケータ用配置 new / 配置 delete 定義】

```
//--------------------
//配置 new/配置 delete
//※メモリ使用状況を確認するためにマーカーを表示
//配置 new
void* operator new(const std::size_t size, CStackAllocator& allocator,
                                                const std::size_t align = IStackAllocator::DEFAULT_ALIGN) throw()
{
     printf("placement new(size=%d, stack_allocator.marker=%d, align=%d)¥n", size, allocator.getMarker(), align);
     return allocator.alloc(size, align);
}
//配列版
void* operator new[](const std::size_t size, CStackAllocator& allocator,
                                                const std::size_t align = IStackAllocator::DEFAULT_ALIGN) throw()
{
     printf("placement new[](size=%d, stack_allocator.marker=%d, align=%d)¥n", size, allocator.getMarker(), align);
     return allocator.alloc(size, align);
}
//配置 delete
void operator delete(void* p, CStackAllocator& allocator, const std::size_t) throw()
{
     printf("placement delete(p=0x%p, stack_allocator.marker=%d)¥n", p, allocator.getMarker());
     //allocator.back(p) は実行しない
}
//配列版
void operator delete[](void* p, CStackAllocator& allocator, const std::size_t) throw()
{
     printf("placement delete[](p=0x%p, stack_allocator.marker=%d)¥n", p, allocator.getMarker());
     //allocator.back(p) は実行しない
}
```

【スタックアロケータ用 delete 関数定義】

```
//デストラクタ呼び出し付き delete
template<class T>
void delete_ptr(T*& p, CStackAllocator& allocator)
{
     printf("delete_ptr(p=0x%p, stack_allocator.marker=%d)¥n", p, allocator.getMarker());
     if (!p)
          return;
     p->~T();//デストラクタ呼び出し
     //operator delete(p, allocator, 0) は実行しない
     //allocator.back(p) は実行しない
     p = nullptr;//安全のためにポインタを初期化
}
//配列版
template<class T>
void delete_ptr(T*& p, const std::size_t array_num, CStackAllocator& allocator)
{
     printf("delete_ptr(p=0x%p, array_num=%d, stack_allocator.marker=%d)¥n", p, array_num, allocator.getMarker());
     if (!p)
          return;
     for (std::size_t i = 0; i < array_num; ++i)
          p->~T();//デストラクタ呼び出し
     //operator delete[](p, allocator, 0) は実行しない
     //allocator.back(p) は実行しない
     p = nullptr;//安全のためにポインタを初期化
}
```

【バッファ付きスタックアロケータテンプレートクラス定義】

```
//--------------------
//バッファ付きスタックアロケータテンプレートクラス
//※静的バッファを使用したければ、固定バッファシングルトン化する
template<std::size_t SIZE>
class CStackAllocatorWithBuff : public CStackAllocator
{
```

```
public:
	//定数
	static const std::size_t BUFF_SIZE = SIZE;//バッファサイズ
public:
	//コンストラクタ
	CStackAllocatorWithBuff() :
		CStackAllocator(m_buff, BUFF_SIZE)
	{}
	//デストラクタ
	~CStackAllocatorWithBuff() override
	{}
private:
	//フィールド
	byte m_buff[BUFF_SIZE];//バッファ
};
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest4a
{
public:
	//デフォルトコンストラクタ
	CTest4a() :
		CTest4a("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
		//m_name("(default)")//C++11 が使えない場合は普通に初期化
	{
		printf("CTest4a::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest4a(const char* name) :
		m_name(name)
	{
		printf("CTest4a::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CTest4a()
	{
		printf("CTest4a::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【テスト】

```
//--------------------
//テスト
void test4a()
{
	CStackAllocatorWithBuff<64> stack;//スタックアロケータ生成（64 バイトのバッファを確保）
	printf("buff=0x%p¥n", stack.getBuff());
	{
		printf("-----¥n");
		int* i1 = new(stack) int[2];
		printf("i1=0x%p¥n", i1);
		char* c1 = new(stack, 1) char[13];
		printf("c1=0x%p¥n", c1);
		CStackAllocator::marker_t marker = stack.getMarker();//マーカー取得
		printf("marker=%d¥n", marker);
		float* f1 = new(stack) float[4];
		printf("f1=0x%p¥n", f1);
		CTest4a* obj1_p = new(stack)CTest4a("テスト 4a");
		printf("obj1_p=0x%p¥n", obj1_p);
```

```
            delete_ptr(f1, 4, stack);
            delete_ptr(c1, 15, stack);
            delete_ptr(i1, 2, stack);
            delete_ptr(obj1_p, stack);
            stack.back(marker);//マーカーを戻す
            printf("back(%d)¥n", marker);
    }
    {
            printf("-----¥n");
            CStackAllocator::marker_t marker = stack.getMarker();//マーカー取得
            printf("marker=%d¥n", marker);
            CTest4a* obj2_p = new(stack)CTest4a[3];
            printf("obj2_p=0x%p¥n", obj2_p);
            int* i2 = new(stack) int;
            printf("i2=0x%p¥n", i2);
            float* f2 = new(stack) float;
            printf("f2=0x%p¥n", f2);
            char* c2 = new(stack, 1) char;
            printf("c2=0x%p¥n", c2);
            int* over = new(stack) int[10];//【NG】サイズオーバー
            printf("over=0x%p¥n", over);
            delete_ptr(f2, stack);
            delete_ptr(c2, stack);
            delete_ptr(i2, stack);
            delete_ptr(obj2_p, 3, stack);
    }
}
```

↓（実行結果）

```
buff=0x002DFA4C                                          ←バッファの先頭アドレス
-----
placement new[](size=8, stack_allocator.marker=0, align=4)   ←メモリ確保時点のマーカーと要求されたアライメントを表示
                                                              （以降、同様に赤字で示す）
i1=0x002DFA4C                                            ←確保したメモリのアドレス（以降、同様に赤字で示す）
placement new[](size=13, stack_allocator.marker=8, align=1)
c1=0x002DFA54
marker=21                                                ←現時点のマーカー（この時点のマーカーを記憶）
placement new[](size=16, stack_allocator.marker=21, align=4)
f1=0x002DFA64                                            ←直前のマーカーが奇数だったが、アラインメント調整された
placement new(size=8, stack_allocator.marker=40, align=4)    メモリが確保されている
CTest4a::Constructor : name="テスト4a"
obj1_p=0x002DFA74
delete_ptr(p=0x002DFA64, array_num=4, stack_allocator.marker=48)  ←delete してもマーカーに変化なし
delete_ptr(p=0x002DFA54, array_num=15, stack_allocator.marker=48) ←（同上）
delete_ptr(p=0x002DFA4C, array_num=2, stack_allocator.marker=48)  ←（同上）
delete_ptr(p=0x002DFA74, stack_allocator.marker=48)               ←（同上）
CTest4a::Destructor : name="テスト4a"
back(21)                                                 ←マーカーを戻す
-----
marker=21                                                ←マーカーが前の時点に正しく戻されている
placement new[](size=28, stack_allocator.marker=21, align=4)
CTest4a::Constructor : name="(default)"
CTest4a::DefaultConstructor : name="(default)"
CTest4a::Constructor : name="(default)"
CTest4a::DefaultConstructor : name="(default)"
CTest4a::Constructor : name="(default)"
CTest4a::DefaultConstructor : name="(default)"
obj2_p=0x002DFA68
placement new(size=4, stack_allocator.marker=52, align=4)
i2=0x002DFA80
placement new(size=4, stack_allocator.marker=56, align=4)
f2=0x002DFA84
placement new(size=1, stack_allocator.marker=60, align=1)
c2=0x002DFA88
placement new[](size=40, stack_allocator.marker=61, align=4)
```

```
stack overflow!(size=40+align=3, remain=3)                    ←メモリ不足発生
over=0x00000000
delete_ptr(p=0x002DFA84, stack_allocator.marker=61)
delete_ptr(p=0x002DFA88, stack_allocator.marker=61)
delete_ptr(p=0x002DFA80, stack_allocator.marker=61)
delete_ptr(p=0x002DFA68, array_num=3, stack_allocator.marker=61)
CTest4a::Destructor : name="(default)"
CTest4a::Destructor : name="(default)"
CTest4a::Destructor : name="(default)"
```

▼ 双方向スタックアロケータのサンプル

双方向スタックアロケータのサンプルプログラムを示す。
双方向であること以外は、前述の「スタックアロケータ」とほぼ同じ。

双方向スタックアロケータは、後述する「二重フレームヒープ」に応用すると、メモリ効率がよくなることが期待できる。

双方向スタックアロケータのサンプル：

【双方向スタックアロケータクラス定義】

```
//---------------------
//双方向スタックアロケータクラス
//※非スレッドセーフ（処理速度優先）
//※インターフェースを実装するが、高速化のために virtual メソッドに頼らず操作可能
class CDualStackAllocator : public IStackAllocator
{
public:
	//アクセッサ
	std::size_t getTotal() const override { return m_buffSize; }//全体のメモリ量を取得
	std::size_t getUsed() const override { return m_usedN + m_buffSize - m_usedR; }//使用中のメモリ量を取得
	std::size_t getRemain() const override { return m_usedR - m_usedN; }//残りのメモリ量を取得
	E_ORDERED getDefaultOrdered() const { return m_defaultOrdered; }//デフォルトのスタック順を取得
	void setDefaultOrdered(const E_ORDERED ordered)//デフォルトのスタック順を更新
	{
		m_defaultOrdered = ordered == REVERSE ? REVERSE : NORMAL;
	}
	const byte* getBuff() const { return m_buffPtr; }//バッファ取得を取得
	const byte* getNowPtrN() const { return m_buffPtr + m_usedN; }//現在のバッファ位置（正順）を取得
	const byte* getNowPtrR() const { return m_buffPtr + m_usedR; }//現在のバッファ位置（逆順）を取得
	const byte* getNowPtrD() const { return getNowPtr(m_defaultOrdered); }//現在のバッファ位置を取得
	const byte* getNowPtr(const E_ORDERED ordered) const {
		return ordered == DEFAULT ? getNowPtrD() : ordered == REVERSE ? getNowPtrR() : getNowPtrN(); }
		                                                                    //現在のバッファ位置を取得
	const byte* getNowPtr() const override { return getNowPtrD(); }//現在のバッファ位置を取得
	marker_t getMarkerN() const { return m_usedN; }//現在のマーカー（正順）を取得
	marker_t getMarkerR() const { return m_usedR; }//現在のマーカー（逆順）を取得
	marker_t getMarkerD() const { return getMarker(m_defaultOrdered); }//現在のマーカーを取得
	marker_t getMarker(const E_ORDERED ordered) const {
		return ordered == DEFAULT ? getMarkerD() : ordered == REVERSE ? getMarkerR() : getMarkerN(); }
		                                                                    //現在のマーカーを取得
	marker_t getMarker() const override { return getMarkerD(); }//現在のマーカーを取得
public:
	//メソッド
	//メモリ確保（正順）
	void* allocN(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
	{
		const uintptr_t now_ptr = reinterpret_cast<uintptr_t>(m_buffPtr) + m_usedN;//現在のポインタ位置算出
		const uintptr_t align_diff = align > 0 ? now_ptr % align == 0 ? 0 : align - now_ptr % align : 0;
		                                                                    //アラインメント計算
```

```
		const marker_t next_used = m_usedN + align_diff + size;//次のマーカー算出
		if (next_used > m_usedR)//メモリオーバーチェック（符号なしなので、範囲チェックは大判定のみで OK）
		{
			printf("normal-stack overflow!(size=%d+align=%d, remain=%d)¥n", size, align_diff, m_usedR - m_usedN);
			return nullptr;
		}
		const uintptr_t alloc_ptr = now_ptr + align_diff;//メモリ確保アドレス算出
		m_usedN = next_used;//マーカー更新
		return reinterpret_cast<void*>(alloc_ptr);
	}
	//メモリ確保（逆順）
	void* allocR(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
	{
		const uintptr_t now_ptr = reinterpret_cast<uintptr_t>(m_buffPtr) + m_usedR;//現在のポインタ位置算出
		const uintptr_t alloc_ptr_tmp = now_ptr - size;//メモリ確保アドレス算出（暫定）
		const uintptr_t align_diff = align > 0 ? alloc_ptr_tmp % align == 0 ? 0 : alloc_ptr_tmp % align : 0;
                                                                                    //アラインメント計算
		const marker_t next_used = m_usedR - size - align_diff;//次のマーカー算出
		if (next_used < m_usedN || next_used > m_buffSize)
                                    //メモリオーバーチェック（オーバーフローして値が大きくなる可能性もチェック）
		{
			printf("reversed-stack overflow!(size=%d+align=%d, remain=%d)¥n",
                                                                        size, align_diff, m_usedR - m_usedN);
			return nullptr;
		}
		const uintptr_t alloc_ptr = now_ptr - size - align_diff;//メモリ確保アドレス算出
		m_usedR = next_used;//マーカー更新
		return reinterpret_cast<void*>(alloc_ptr);
	}
	//メモリ確保
	void* allocD(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
	{
		return alloc(m_defaultOrdered, size, align);
	}
	//メモリ確保
	void* alloc(const E_ORDERED ordered, const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
	{
		return ordered == DEFAULT ? allocD(size, align) :
				ordered == REVERSE ? allocR(size, align) : allocN(size, align);
	}
	//メモリ確保
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN) override
	{
		return allocD(size, align);
	}
	//メモリを以前のマーカーに戻す（正順）
	//※マーカー指定版
	void backN(const marker_t marker_n)
	{
		if (marker_n > m_usedR)//符号なしなので、範囲チェックは大判定のみで OK
			return;
		m_usedN = marker_n;
	}
	//メモリを以前のマーカーに戻す（正順）
	//※ポインタ指定版
	void backN(const void* p)
	{
		const marker_t marker = reinterpret_cast<uintptr_t>(p) - reinterpret_cast<uintptr_t>(m_buffPtr);
		backN(marker);
	}
	//メモリを以前のマーカーに戻す（逆順）
	//※マーカー指定版
	void backR(const marker_t marker_r)
	{
		if (marker_r < m_usedN || marker_r > m_buffSize)//メモリオーバーチェック
```

```
                return;
        m_usedR = marker_r;
    }
    //メモリを以前のマーカーに戻す（逆順）
    //※ポインタ指定版
    void backR(const void* p)
    {
        const marker_t marker = reinterpret_cast<uintptr_t>(p) - reinterpret_cast<uintptr_t>(m_buffPtr);
        backR(marker);
    }
    //メモリを以前のマーカーに戻す
    //※マーカー指定版
    void backD(const marker_t marker)
    {
        back(m_defaultOrdered, marker);
    }
    //メモリを以前のマーカーに戻す
    //※ポインタ指定版
    void backD(const void* p)
    {
        back(m_defaultOrdered, p);
    }
    //メモリを以前のマーカーに戻す
    //※マーカー指定版
    void back(const E_ORDERED ordered, const marker_t marker)
    {
        ordered == DEFAULT ? backD(marker) : ordered == REVERSE ? backR(marker) : backN(marker);
    }
    //メモリを以前のマーカーに戻す
    //※ポインタ指定版
    void back(const E_ORDERED ordered, const void* p)
    {
        ordered == DEFAULT ? backD(p) : ordered == REVERSE ? backR(p) : backN(p);
    }
    //メモリを以前のマーカーに戻す
    //※マーカー指定版
    void back(const marker_t marker) override
    {
        backD(marker);
    }
    //メモリを以前のマーカーに戻す
    //※ポインタ指定版
    void back(const void* p) override
    {
        backD(p);
    }
    //メモリ破棄（正順）
    void clearN()
    {
        m_usedN = 0;
    }
    //メモリ破棄（逆順）
    void clearR()
    {
        m_usedR = m_buffSize;
    }
    //メモリ破棄
    void clearD()
    {
        clear(m_defaultOrdered);
    }
    //メモリ破棄（両方）
    void clearNR()
    {
        clearN();
```

```
            clearR();
      }
      //メモリ破棄
      void clear(const E_ORDERED ordered)
      {
            ordered == DEFAULT ? clearD() : ordered == REVERSE ? clearR() : clearN();
      }
      //メモリ破棄
      void clear() override
      {
            clearD();
      }
      //メモリ破棄（全て）
      void clearAll() override
      {
            clearNR();
      }
public:
      //コンストラクタ
      CDualStackAllocator(void* buff_p, const std::size_t buff_size, const E_ORDERED default_ordered = NORMAL) :
            m_buffPtr(reinterpret_cast<byte*>(buff_p)),//バッファ先頭アドレス
            m_buffSize(buff_size),//バッファサイズ
            m_usedN(0),//マーカー（正順）
            m_usedR(buff_size)//マーカー（逆順）
      {
            setDefaultOrdered(default_ordered);//デフォルトのスタック順を補正
      }
      //デストラクタ
      ~CDualStackAllocator() override
      {}
private:
      //フィールド
      byte* m_buffPtr;//バッファ先頭アドレス
      const std::size_t m_buffSize;//バッファサイズ
      marker_t m_usedN;//マーカー（正順）
      marker_t m_usedR;//マーカー（逆順）
      E_ORDERED m_defaultOrdered;//デフォルトのスタック順
};
```

【双方向スタックアロケータ用配置 new / 配置 delete 定義】

```
//--------------------
//配置 new/配置 delete
//※メモリ使用状況を確認するためにマーカーを表示
//配置 new
void* operator new(const std::size_t size, CDualStackAllocator& allocator, const std::size_t align,
                   const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
      printf("placement new(size=%d, dual_allocator.marker=%d,%d, align=%d, ordered=%d)¥n",
                                          size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
      return allocator.alloc(ordered, size, align);
}
void* operator new(const std::size_t size, CDualStackAllocator& allocator,
                   const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
      const std::size_t align = IStackAllocator::DEFAULT_ALIGN;
      printf("placement new(size=%d, dual_allocator.marker=%d,%d, (align=%d), ordered=%d)¥n",
                                          size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
      return allocator.alloc(ordered, size, align);
}
//配列版
void* operator new[](const std::size_t size, CDualStackAllocator& allocator, const std::size_t align,
                   const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
      printf("placement new(size=%d, dual_allocator.marker=%d,%d, align=%d, ordered=%d)¥n",
                                          size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
```

```
	return allocator.alloc(ordered, size, align);
}
void* operator new[](const std::size_t size, CDualStackAllocator& allocator,
                             const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
	const std::size_t align = IStackAllocator::DEFAULT_ALIGN;
	printf("placement new(size=%d, dual_allocator.marker=%d,%d, (align=%d), ordered=%d)¥n",
                                            size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
	return allocator.alloc(ordered, size, align);
}
//配置 delete
void operator delete(void* p, CDualStackAllocator& allocator, const std::size_t align,
                                                                    const IStackAllocator::E_ORDERED) throw()
{
	printf("placement delete(p=0x%p, dual_allocator.marker=%d,%d, align)¥n",
                                                                p, allocator.getMarkerN(), allocator.getMarkerR());
	//allocator.back(ordered, p) は実行しない
}
void operator delete(void* p, CDualStackAllocator& allocator, const IStackAllocator::E_ORDERED) throw()
{
	printf("placement delete(p=0x%p, dual_allocator.marker=%d,%d)¥n",
                                                                p, allocator.getMarkerN(), allocator.getMarkerR());
	//allocator.back(ordered, p) は実行しない
}
//配列版
void operator delete[](void* p, CDualStackAllocator& allocator,
                                    const std::size_t align, const IStackAllocator::E_ORDERED ordered) throw()
{
	printf("placement delete[](p=0x%p, dual_allocator.marker=%d,%d, align)¥n",
                                                                p, allocator.getMarkerN(), allocator.getMarkerR());
	//allocator.back(ordered, p) は実行しない
}
void operator delete[](void* p, CDualStackAllocator& allocator, const IStackAllocator::E_ORDERED ordered) throw()
{
	const std::size_t align = IStackAllocator::DEFAULT_ALIGN;
	printf("placement delete[](p=0x%p, dual_allocator.marker=%d,%d)¥n",
                                                                p, allocator.getMarkerN(), allocator.getMarkerR());
	//allocator.back(ordered, p) は実行しない
}
```

【双方向スタックアロケータ用 delete 関数定義】

```
//デストラクタ呼び出し付き delete
template<class T>
void delete_ptr(T*& p, CDualStackAllocator& allocator)
{
	printf("delete_ptr(p=0x%p, dual_allocator.marker=%d,%d)¥n", p, allocator.getMarkerN(), allocator.getMarkerR());
	if (!p)
		return;
	p->~T();//デストラクタ呼び出し
	//operator delete(p, allocator, 0) は実行しない
	//allocator.back(p) は実行しない
	p = nullptr;//安全のためにポインタを初期化
}
//配列版
template<class T>
void delete_ptr(T*& p, const std::size_t array_num, CDualStackAllocator& allocator)
{
	printf("delete_ptr(p=0x%p, array_num=%d, dual_allocator.marker=%d,%d)¥n",
                                                    p, array_num, allocator.getMarkerN(), allocator.getMarkerR());
	if (!p)
		return;
	for (std::size_t i = 0; i < array_num; ++i)
		p->~T();//デストラクタ呼び出し
	//operator delete[](p, allocator, 0) は実行しない
	//allocator.back(p) は実行しない
```

```
	p = nullptr;//安全のためにポインタを初期化
}
```

【バッファ付き双方向スタックアロケータテンプレートクラス定義】

```
//--------------------
//バッファ付き双方向スタックアロケータテンプレートクラス
//※静的バッファを使用したければ、固定バッファシングルトン化する
template<std::size_t SIZE>
class CDualStackAllocatorWithBuff : public CDualStackAllocator
{
public:
	//定数
	static const std::size_t BUFF_SIZE = SIZE;//バッファサイズ
public:
	//コンストラクタ
	CDualStackAllocatorWithBuff(const E_ORDERED default_ordered = NORMAL) :
		CDualStackAllocator(m_buff, BUFF_SIZE, default_ordered)
	{}
	//デストラクタ
	~CDualStackAllocatorWithBuff() override
	{}
private:
	//フィールド
	byte m_buff[BUFF_SIZE];//バッファ
};
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest4b
{
public:
	//デフォルトコンストラクタ
	CTest4b() :
		CTest4b("(default)")//他のコンストラクタ呼び出し（C++11で追加された仕様）
		//m_name("(default)")//C++11が使えない場合は普通に初期化
	{
		printf("CTest4b::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest4b(const char* name) :
		m_name(name)
	{
		printf("CTest4b::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CTest4b()
	{
		printf("CTest4b::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【テスト】

```
//--------------------
//テスト
void test4b()
{
	CDualStackAllocatorWithBuff<64> stack(DSA_NORMAL);//双方向スタックアロケータ生成（64バイトのバッファを確保）
	printf("buff=0x%p¥n", stack.getBuff());
	{
		printf("-----¥n");
```

```
        printf("getDefaultOrdered()=%d¥n", stack.getDefaultOrdered());
        int* i1 = new(stack) int[2];
        printf("i1=0x%p¥n", i1);
        char* c1 = new(stack, 1, DSA_REVERSE) char[13];
        printf("c1=0x%p¥n", c1);
        CStackAllocator::marker_t marker = stack.getMarker();//マーカー取得
        CStackAllocator::marker_t marker_n = stack.getMarkerN();//マーカー取得
        CStackAllocator::marker_t marker_r = stack.getMarkerR();//マーカー取得
        printf("marker: default=%d, normal=%d, reverse=%d¥n", marker, marker_n, marker_r);
        float* f1 = new(stack) float[4];
        printf("f1=0x%p¥n", f1);
        CTest4b* obj1_p = new(stack, DSA_REVERSE)CTest4b("テスト4b");
        printf("obj1_p=0x%p¥n", obj1_p);
        delete_ptr(f1, 4, stack);
        delete_ptr(c1, 15, stack);
        delete_ptr(i1, 2, stack);
        delete_ptr(obj1_p, stack);
        stack.backN(marker_n);//マーカーを戻す
        printf("backN(%d)¥n", marker_n);
        stack.backR(marker_r);//マーカーを戻す
        printf("backR(%d)¥n", marker_n);
    }
    {
        printf("-----¥n");
        stack.setDefaultOrdered(DSA_REVERSE);//デフォルト伸すタック順を変更
        printf("getDefaultOrdered()=%d¥n", stack.getDefaultOrdered());
        CStackAllocator::marker_t marker = stack.getMarker();//マーカー取得
        CStackAllocator::marker_t marker_n = stack.getMarkerN();//マーカー取得
        CStackAllocator::marker_t marker_r = stack.getMarkerR();//マーカー取得
        printf("marker: default=%d, normal=%d, reverse=%d¥n", marker, marker_n, marker_r);
        CTest4b* obj2_p = new(stack, DSA_NORMAL)CTest4b[3];
        printf("obj2_p=0x%p¥n", obj2_p);
        int* i2 = new(stack) int;
        printf("i2=0x%p¥n", i2);
        float* f2 = new(stack, DSA_NORMAL) float;
        printf("f2=0x%p¥n", f2);
        char* c2 = new(stack, 1) char;
        printf("c2=0x%p¥n", c2);
        int* over1 = new(stack, DSA_NORMAL) int[10];//【NG】サイズオーバー
        printf("over1=0x%p¥n", over1);
        int* over2 = new(stack) int[10];//【NG】サイズオーバー
        printf("over2=0x%p¥n", over2);
        delete_ptr(f2, stack);
        delete_ptr(c2, stack);
        delete_ptr(i2, stack);
        delete_ptr(obj2_p, 3, stack);
    }
}
```

↓（実行結果）

```
buff=0x0076FA84                                                    ←バッファの先頭アドレス
-----
getDefaultOrdered()=1                                              ←現在のデフォルトのスタック順
placement new(size=8, dual_allocator.marker=0,64, (align=4), ordered=0) ←メモリ確保時点のマーカーと要求された向き、
                                                                     アライメントを表示（以降、同様に赤字で示す）
                                                                     ordered は 0 が正順、1 が逆順
i1=0x0076FA84                                                      ←確保したメモリのアドレス
                                                                     （以降、同様に赤字で示す）
placement new(size=13, dual_allocator.marker=8,64, align=1, ordered=-1)
c1=0x0076FAB7                                                      ←指定のアラインメントが 1 だったので
                                                                     アドレスが奇数になっている（問題なし）
marker: default=8, normal=8, reverse=51                            ←現時点のマーカー（この時点のマーカーを記憶）
placement new(size=16, dual_allocator.marker=8,51, (align=4), ordered=0)
f1=0x0076FA8C
placement new(size=8, dual_allocator.marker=24,51, (align=4), ordered=-1)
```

```
CTest4b::Constructor : name="テスト4b"
obj1_p=0x0076FAAC                                                    ←直前のマーカーが奇数だったが、
                                                                       アラインメント調整されたメモリが確保
delete_ptr(p=0x0076FA8C, array_num=4, dual_allocator.marker=24,40)   ←delete してもマーカーに変化なし
delete_ptr(p=0x0076FAB7, array_num=15, dual_allocator.marker=24,40)  ←（同上）
delete_ptr(p=0x0076FA84, array_num=2, dual_allocator.marker=24,40)   ←（同上）
delete_ptr(p=0x0076FAAC, dual_allocator.marker=24,40)                ←（同上）
CTest4b::Destructor : name="テスト4b"
backN(8)                                                             ←マーカーを戻す
backR(51)                                                            ←マーカーを戻す
-----
getDefaultOrdered()=-1                                               ←現在のデフォルトのスタック順
marker: default=51, normal=8, reverse=51                             ←マーカーが前の時点に正しく戻されている
placement new(size=28, dual_allocator.marker=8,51, (align=4), ordered=1)
CTest4b::Constructor : name="(default)"
CTest4b::DefaultConstructor : name="(default)"
CTest4b::Constructor : name="(default)"
CTest4b::DefaultConstructor : name="(default)"
CTest4b::Constructor : name="(default)"
CTest4b::DefaultConstructor : name="(default)"
obj2_p=0x0076FA90
placement new(size=4, dual_allocator.marker=36,51, (align=4), ordered=0)
i2=0x0076FAB0
placement new(size=4, dual_allocator.marker=36,44, (align=4), ordered=1)
f2=0x0076FAA8
placement new(size=1, dual_allocator.marker=40,44, align=1, ordered=0)
c2=0x0076FAAF
placement new(size=40, dual_allocator.marker=40,43, (align=4), ordered=0)
normal-stack overflow!(size=40+align=0, remain=3)                    ←メモリ不足発生
over1=0x00000000
placement new(size=40, dual_allocator.marker=40,43, ordered=1, align=4)
reversed-stack overflow!(size=40+align=3, remain=3)                  ←メモリ不足発生
over2=0x00000000
delete_ptr(p=0x0076FAA8, dual_allocator.marker=40,43)
delete_ptr(p=0x0076FAAF, dual_allocator.marker=40,43)
delete_ptr(p=0x0076FAB0, dual_allocator.marker=40,43)
delete_ptr(p=0x0076FA90, array_num=3, dual_allocator.marker=40,43)
CTest4b::Destructor : name="(default)"
CTest4b::Destructor : name="(default)"
CTest4b::Destructor : name="(default)"
```

■ プールアロケータ

ゲームプログラミングでは（ゲームプログラミングに限らずだが）、同じサイズのオブジェクトを大量に必要とすることがある。そのような要件に、「プールアロケータ」が便利に使える。

多数のメモリブロック（オブジェクト）を単純な配列で管理するため、メモリ効率が良い。また、構造が単純であるため、高速に処理できる。フラグメンテーションを起こすこともなく、限界まで確実にメモリ確保が可能であることを保証し、かつ、限界以上の余計なメモリを消費することがないことを保証する。

様々な処理やリソースで意識的にメモリを分け合って協調動作するゲームシステムでは、とても扱い易いメモリ管理手法である。

別紙の「ゲーム制御のためのメモリ管理方針」および「柔軟性を追求したメモリ管理システム」では、メモリ管理システムの一部として、プールアロケータおよびスラブアロケータの導入を設計しており、必要に応じて自動拡張する仕様にしている。ここで扱うのは、そのようなフレキシブルなシステムではなく、もっと単純な、固定メモリのシステムである。

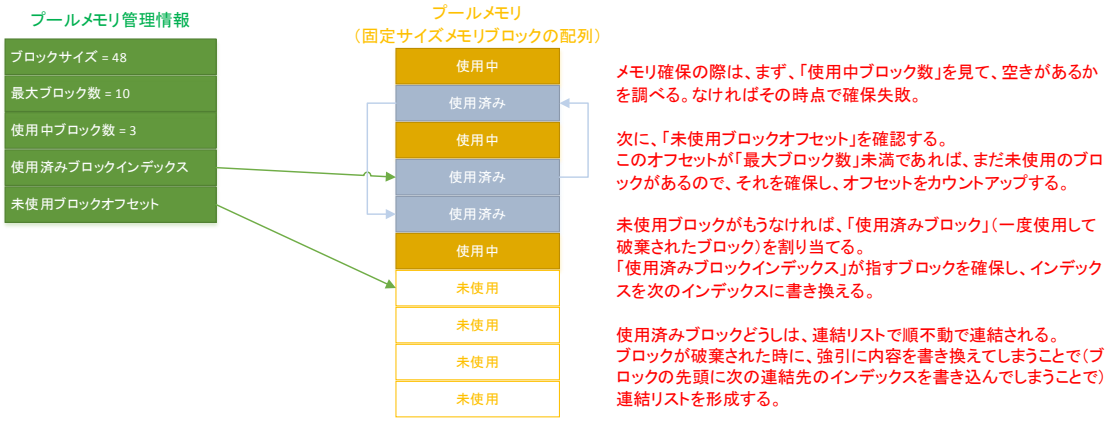
▼ プールアロケータの仕組み

プールアロケータは、とても単純な仕組みである。

一塊の大きな配列と、使用可能な配列要素を管理する。

メモリ確保の要求に応じて配列要素を割り当て、解放された要素は再利用可能な状態にする。

プールアロケータの仕組み：



▼ プールアロケータのサンプル

プールアロケータのサンプルプログラムを示す。

サンプルでは、ローカル変数領域（スタック領域）のバッファを使用した処理としている。

サンプルプログラム中に定数「INT_MAX」を使用しているのが、これは「#include <limits.h>」により使用できる。

別紙の「効率化と安全性のためのロック制御」にもほぼ同じサンプルを掲載しているが、本書のサンプルではロック制御を行っておらず、非スレッドセーフである。

プールアロケータのサンプル：

【プロトタイプ宣言】

```
//--------------------
//クラス宣言
class CPoolAllocator;
```

```
//--------------------
//配置 new
void* operator new(const std::size_t size, CPoolAllocator& allocator) throw();
//配置 delete
void operator delete(void* p, CPoolAllocator& allocator) throw();
```

【プールアロケータクラス定義】

```
//---------------------------------------
//プールアロケータクラス
//※非スレッドセーフ
class CPoolAllocator
{
public:
	//型宣言
	typedef unsigned char byte;//バッファ用
	typedef int index_t;//インデックス用
public:
	//定数
	static const index_t INVALID_INDEX = INT_MAX;//ブロックインデックスの無効値
public:
	//アクセッサ
	const byte* getBuff() const { return m_pool; }//バッファ取得
	std::size_t getBlockSize() const { return m_blockSize; }//ブロックサイズ
	index_t getBlocksNum() const { return m_poolBlocksNum; }//プールブロック数
	index_t getUsed() const { return m_used;/*.load();*/ }//使用中数取得
	index_t getRemain() const { return m_poolBlocksNum - m_used;/*.load();*/ }//残数取得
	const byte* getBlockConst(const index_t index) const { return m_pool + index * m_blockSize; }//ブロック取得
private:
	byte* getBlock(const index_t index){ return m_pool + index * m_blockSize; }//ブロック取得
private:
	//メソッド
	//メモリ確保状態リセット
	void reset()
	{
		//ロック取得
		//m_lock.lock();

		m_used = 0;/*.store(0);*///使用中数
		m_next = 0;//未使用先頭インデックス
		m_recycle = INVALID_INDEX;//リサイクルインデックス

		//ロック解放
		//m_lock.unlock();
	}
	//メモリブロック確保
	//※使用中フラグの空きを検索してフラグを更新し、
	//  確保したインデックスを返す
	int assign()
	{
		//使用中ブロック数チェックは、高速化のために、一度ロックの範囲外で行う
		if (m_used/*.load()*/ >= m_poolBlocksNum)
			return INVALID_INDEX;//空きなし

		//ロック取得
		//m_lock.lock();

		//使用中ブロック数を再チェック
		//if (m_used.load() >= m_poolBlocksNum)
		//{
		//	m_lock.unlock();//ロック解放
		//	return INVALID_INDEX;//空きなし
		//}

		//インデックス確保
```

```
            index_t index = INVALID_INDEX;
            if (m_next < m_poolBlocksNum)
            {
                  //未使用インデックスがある場合
                  index = m_next++;//未使用先頭インデックスカウントアップ
                  ++m_used;/*.fetch_add(1);*///使用中数カウントアップ
            }
            else
            {
                  if (m_recycle != INVALID_INDEX)
                  {
                        //リサイクル可能なインデックスがある場合
                        index = m_recycle;//リサイクルインデックス
                        m_recycle = *reinterpret_cast<unsigned int*>(getBlock(index));
                                                //リサイクルインデックス更新（空きノードの先頭に書き込まれている）
                        ++m_used;/*.fetch_add(1);*///使用中数カウントアップ
                  }
            }

            //ロック解放
            //m_lock.unlock();

            //終了
            return index;
      }
      //メモリブロック解放
      //※指定のインデックスの使用中フラグをリセット
      void release(const index_t index)
      {
            //インデックスの範囲チェック（ロックの範囲外で行う）
            if (index < 0 || index >= m_poolBlocksNum)
                  return;

            //ロック取得
            //m_lock.lock();

            //リサイクル
            *reinterpret_cast<unsigned int*>(getBlock(index)) = m_recycle;
                                          //リサイクルインデックス書き込み（空きノードの先頭に強引に書き込む）
            m_recycle = index;//リサイクルインデックス組み換え

            //使用中数カウントダウン
            --m_used;/*.fetch_sub(1);*/

            //ロック解放
            //m_lock.unlock();
      }
public:
      //メモリ確保
      void* alloc(const std::size_t size)
      {
            //【アサーション】要求サイズがブロックサイズを超える場合は即時確保失敗
            //ASSERT(size <= m_blockSize, "CPoolAllocator::alloc(%d) cannot allocate. Size must has been under %d.",
            //      size, m_blockSize);
            if (size > m_blockSize)
            {
                  return nullptr;
            }
            //空きブロックを確保して返す
            const index_t index = assign();
            //【アサーション】全ブロック使用中につき、確保失敗
            //ASSERT(index >= 0, "CPoolAllocator::alloc(%d) cannot allocate. Buffer is full. (num of blocks is %d)",
            //      size, m_poolBlocksNum);
            //確保したメモリを返す
            return index == INVALID_INDEX ? nullptr : getBlock(index);
```

```
	}
	//メモリ解放
	void free(void * p)
	{
		//nullptr 時は即時解放失敗
		//ASSERT(p != nullptr, "CPoolAllocator::free() cannot free. Pointer is null.");
		if (!p)
			return;
		//ポインタからインデックスを算出
		const intptr_t diff = reinterpret_cast<intptr_t>(p) - reinterpret_cast<intptr_t>(m_pool);//ポインタの差分
		const index_t index = diff / m_blockSize;//ブロックサイズで割ってインデックス算出
		//【アサーション】メモリバッファの範囲外なら処理失敗（release 関数内で失敗するのでそのまま実行）
		//ASSERT(index >= 0 && index < m_poolBlocksNum, "CPoolAllocator::free() cannot free. Pointer is different.");
		//【アサーション】ポインタが各ブロックの先頭を指しているかチェック
		//　　⇒多重継承とキャストしているとずれることがるのでこの問題は無視して解放してしまう
		//ASSERT(diff % m_blockSize == 0, "CPoolAllocator::free() cannot free. Pointer is illegal.");
		//算出したインデックスでメモリ解放
		release(index);
	}
	//コンストラクタ呼び出し機能付きメモリ確保
	//※C++11 の可変長テンプレートパラメータを活用
	template<class T, typename... Tx>
	T* create(Tx... nx)
	{
		return new(*this) T(nx...);
	}
	//デストラクタ呼び出し機能付きメモリ解放
	//※解放後、ポインタに nullptr をセットする
	template<class T>
	void destroy(T*& p)
	{
		if (!p)
			return;
		p->~T();//明示的なデストラクタ呼び出し（デストラクタ未定義のクラスでも問題なし）
		operator delete(p, *this);//配置 delete 呼び出し
		p = nullptr;//ポインタには nullptr をセット
	}
public:
	//コンストラクタ
	CPoolAllocator(void* pool_buff, const std::size_t block_size, const index_t pool_blocks_num) :
		m_pool(reinterpret_cast<byte*>(pool_buff)),//プールバッファ
		m_blockSize(block_size),//ブロックサイズ　※4 倍数であること
		m_poolBlocksNum(pool_blocks_num)//プールブロック数
	{
		//【アサーション】パラメータチェック
		//ASSERT((m_blockSize & 0x3) == 0, "CPoolAllocator::CPoolAllocator() block size is invalid.");

		//使用中フラグリセット
		reset();
	}
	//デストラクタ
	~CPoolAllocator()
	{}
private:
	//フィールド
	byte* m_pool;//プールバッファ
	const std::size_t m_blockSize;//ブロックサイズ
	const index_t m_poolBlocksNum;//プールブロック数
	index_t/*std::atomic<index_t>;*/ m_used;//使用中数
	index_t m_next;//未使用先頭インデックス
	index_t m_recycle;//リサイクルインデックス
	//CSpinLock m_lock;//ロック
};
```

【プールアロケータ用配置 new / 配置 delete 定義】

```
//--------------------
//配置 new
void* operator new(const std::size_t size, CPoolAllocator& allocator) throw()
{
	printf("placement new(size=%d, pool_allocator.used=%d/%d/%d)¥n",
					size, allocator.getTotal(), allocator.getUsed(), allocator.getRemain());
	return allocator.alloc(size);
}
//配置 delete
void operator delete(void* p, CPoolAllocator& allocator) throw()
{
	printf("placement delete(p=0x%p, pool_allocator.used=%d/%d/%d)¥n",
					p, allocator.getTotal(), allocator.getUsed(), allocator.getRemain());
	allocator.free(p);
}
```

【バッファ付きプールアロケータテンプレートクラス定義：ブロックサイズとブロック数指定】

```
//---------------------------------------
//バッファ付きプールアロケータクラス：ブロックサイズとブロック数指定
template<std::size_t S, int N>
class CPoolAllocatorWithBuff : public CPoolAllocator
{
public:
	//定数
	static const std::size_t BLOCK_SIZE = (S + 3) & ~3;//ブロックサイズ（4 の倍数に補正）
	static const index_t POOL_BLOCKS_NUM = N;//プールブロック数
public:
	//コンストラクタ
	CPoolAllocatorWithBuff() :
		CPoolAllocator(&m_poolBuff, BLOCK_SIZE, POOL_BLOCKS_NUM)
	{}
	//デストラクタ
	~CPoolAllocatorWithBuff()
	{}
private:
	//フィールド
	byte m_poolBuff[POOL_BLOCKS_NUM][BLOCK_SIZE];//プールバッファ
};
```

【バッファ付きプールアロケータテンプレートクラス定義：データ型とブロック数指定】

```
//---------------------------------------
//バッファ付きプールアロケータクラス：データ型とブロック数指定
template<typename T, int N>
class CPoolAllocatorWithType : public CPoolAllocatorWithBuff<sizeof(T), N>
{
public:
	//型
	typedef T data_t;//データ型
public:
	//定数
	static const std::size_t TYPE_SIZE = sizeof(data_t);//型のサイズ
public:
	//コンストラクタ呼び出し機能付きメモリ確保
	//※C++11 の可変長テンプレートパラメータを活用
	template<typename... Tx>
	data_t* createData(Tx... nx)
	{
		return CPoolAllocator::create<data_t>(nx...);
	}
	//デストラクタ呼び出し機能付きメモリ解放
	//※解放後、ポインタに nullptr をセットする
	void destroyData(data_t*& p)
	{
		CPoolAllocator::destroy(p);
```

```
	}
public:
	//コンストラクタ
	CPoolAllocatorWithType() :
		CPoolAllocatorWithBuff<TYPE_SIZE, N>()
	{}
	//デストラクタ
	~CPoolAllocatorWithType()
	{}
};
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest5
{
public:
	//デフォルトコンストラクタ
	CTest5() :
		CTest5("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
		//m_name("(default)")//C++11 が使えない場合は普通に初期化
	{
		printf("CTest5::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest5(const char* name) :
		m_name(name)
	{
		printf("CTest5::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CTest5()
	{
		printf("CTest5::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
//大きいクラス
class CTest5L
{
	char m_dummy[9];
};
```

【テスト】

```
//--------------------
//テスト
void test5()
{
	CPoolAllocatorWithType<CTest5, 5> pool;//プールアロケータ生成（sizeof(CTest5) * 5 のバッファを確保）
	printf("buff=0x%p¥n", pool.getBuff());
	{
		printf("-----¥n");
		//createData() メソッドでインスタンスを生成
		//※任意のコンストラクタ引数も受け渡し可
		CTest5* obj1_p = pool.createData("テスト 5-1");  //createData()は、プールアロケータに定義したデータ型で
		printf("obj1_p=0x%p¥n", obj1_p);                 //インスタンスを生成する
		CTest5* obj2_p = pool.createData("テスト 5-2");
		printf("obj2_p=0x%p¥n", obj2_p);
		CTest5* obj3_p = pool.createData("テスト 5-3");
		printf("obj3_p=0x%p¥n", obj3_p);
		CTest5* obj4_p = pool.createData("テスト 5-4");
		printf("obj4_p=0x%p¥n", obj4_p);
```

```
            CTest5* obj5_p = pool.createData("テスト5-5");
            printf("obj5_p=0x%p¥n", obj5_p);
            CTest5* obj6_p = pool.createData("テスト5-6");//【NG】個数オーバー
            printf("obj6_p=0x%p¥n", obj6_p);
            //destroyData() メソッドでインスタンスを破棄
            //※デストラクタ呼び出しに対応
            pool.destroyData(obj1_p);
            pool.destroyData(obj2_p);
            pool.destroyData(obj3_p);
            pool.destroyData(obj4_p);
            pool.destroyData(obj5_p);
            pool.destroyData(obj6_p);
        }
        {
            printf("-----¥n");
            //sizeof(CTest5)以下のサイズの型なら何でも扱える
            CTest5L* obj_l_p = pool.create<CTest5L>();//【NG】定義時のクラスよりサイズが大きいクラス
            printf("obj_l_p=0x%p¥n", obj_l_p);
            char* c_p = pool.create<char>();
            printf("c_p=0x%p¥n", c_p);
            short* s_p = pool.create<short>();
            printf("s_p=0x%p¥n", s_p);
            int* i_p = pool.create<int>();
            printf("i_p=0x%p¥n", i_p);
            float* f_p = pool.create<float>();
            printf("f_p=0x%p¥n", f_p);
            double* d_p = pool.create<double>();
            printf("d_p=0x%p¥n", d_p);
            long long* ll_p = pool.create<long long>();//【NG】個数オーバー
            printf("ll_p=0x%p¥n", ll_p);
            //destroy() メソッドはデストラクタがない型でも問題なし
            pool.destroy(obj_l_p);
            pool.destroy(c_p);
            pool.destroy(s_p);
            pool.destroy(i_p);
            pool.destroy(f_p);
            pool.destroy(d_p);
            pool.destroy(ll_p);
        }
}
```

↓（実行結果）

```
buff=0x0052FB64                                 ←バッファの先頭アドレス
-----
placement new(size=8, pool_allocator.used=5/0/5)←メモリ確保時点の使用中ブロック数と残数を表示（以降、同様に赤字で示す）
CTest5::Constructor : name="テスト5-1"
obj1_p=0x0052FB64                               ←確保したメモリのアドレス（以降、同様に赤字で示す）
placement new(size=8, pool_allocator.used=5/1/4)
CTest5::Constructor : name="テスト5-2"
obj2_p=0x0052FB6C
placement new(size=8, pool_allocator.used=5/2/3)
CTest5::Constructor : name="テスト5-3"
obj3_p=0x0052FB74
placement new(size=8, pool_allocator.used=5/3/2)
CTest5::Constructor : name="テスト5-4"
obj4_p=0x0052FB7C
placement new(size=8, pool_allocator.used=5/4/1)
CTest5::Constructor : name="テスト5-5"
obj5_p=0x0052FB84
placement new(size=8, pool_allocator.used=5/5/0)←ブロック数不足
obj6_p=0x00000000                               ←メモリ確保失敗
CTest5::Destructor : name="テスト5-1"
placement delete(p=0x0052FB64, pool_allocator.used=5/5/0)
CTest5::Destructor : name="テスト5-2"
placement delete(p=0x0052FB6C, pool_allocator.used=5/4/1)
```

```
CTest5::Destructor : name="テスト5-3"
placement delete(p=0x0052FB74, pool_allocator.used=5/3/2)
CTest5::Destructor : name="テスト5-4"
placement delete(p=0x0052FB7C, pool_allocator.used=5/2/3)
CTest5::Destructor : name="テスト5-5"
placement delete(p=0x0052FB84, pool_allocator.used=5/1/4)
-----
placement new(size=9, pool_allocator.used=5/0/5)←ブロックサイズオーバー
obj_l_p=0x00000000                              ←メモリ確保失敗
placement new(size=1, pool_allocator.used=5/0/5)
c_p=0x0052FB84                                  ←仕組み上、最も新しく破棄されたブロックから順に再利用される
placement new(size=2, pool_allocator.used=5/1/4)
s_p=0x0052FB7C
placement new(size=4, pool_allocator.used=5/2/3)
i_p=0x0052FB74
placement new(size=4, pool_allocator.used=5/3/2)
f_p=0x0052FB6C
placement new(size=8, pool_allocator.used=5/4/1)
d_p=0x0052FB64
placement new(size=8, pool_allocator.used=5/5/0)←ブロック数不足
ll_p=0x00000000                                 ←メモリ確保失敗
placement delete(p=0x0052FB84, pool_allocator.used=5/5/0)
placement delete(p=0x0052FB7C, pool_allocator.used=5/4/1)
placement delete(p=0x0052FB74, pool_allocator.used=5/3/2)
placement delete(p=0x0052FB6C, pool_allocator.used=5/2/3)
placement delete(p=0x0052FB64, pool_allocator.used=5/1/4)
```

▼ プールアロケータの応用

プールアロケータで大量のオブジェクトを扱うにあたって、個々のオブジェクトの初期化を効率的にする手法がある。デザインパターンの「プロトタイプパターン」と「フライウェイトパターン」を活用する手法である。

詳しい解説については、別紙の「ゲーム制御のためのメモリ管理方針」にて、「スラブアロケータの活用」として説明している。なお、本書におけるプールアロケータは、「ゲーム制御のためのメモリ管理方針」における「スラブアロケータ」と、ほぼ同じである。

■ その他のメモリアロケータ

自由なサイズのメモリ確保を行い、自由に解放できるメモリアロケータに「ヒープアロケータ」がある。

一般的に、静的領域、スタック領域以外のメモリはヒープメモリであり、malloc() 関数と free()関数で操作する。

ゲームプログラミングでは、この標準関数任せのメモリ管理を避け、ヒープメモリと同様のメモリ管理システムを作成することが多い。それは、ゲーム要件に合わせたメモリ制限やデバッグ情報の記録、更に効率的なメモリ操作などを目的とする。

ヒープメモリについては、別紙の「ゲーム制御のためのメモリ管理方針」と「柔軟性を追求したメモリ管理システム」にて設計を行っている。

そこでは、ヒープメモリのほか、メモリをページ管理するための「バディシステム」、小さいメモリを効率的に扱うための「フレームアロケータ」、多数のオブジェクトを動的に管理するための「スラブアロケータ」についても説明している。

■ フレームアロケータ

ここまでは基本的なアロケータの仕組みの説明であったが、ここからはアロケータの応用を説明していく。

まずは「フレームアロケータ」について説明する。

▼ フレームアロケータの考え方

ゲーム中のメモリ確保では、短期的にその役目を終えることも多い。

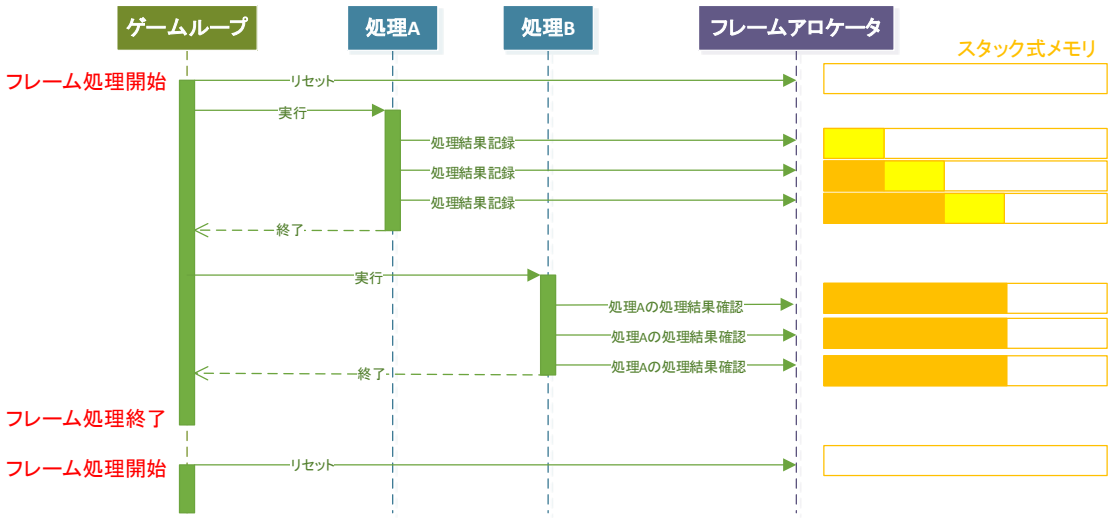
例えば、「処理 A」の結果を「処理 B」に受け渡し、「処理 B」が処理を終えたら破棄して良いようなデータがある。単純に「処理 A」から「処理 B」の関数呼び出しでデータを受け渡すならメモリ確保の必要すらないが、ゲームの場合、処理効率のために、同系統の処理を集中的に行い、多数の処理結果を一旦メモリにプールし、それをまとめて受け渡すといった流れも多い。

このような要件に対するメモリ確保には、「フレームアロケータ」を用いることで、メモリ効率と処理効率を向上させることができる。

フレームアロケータの実装には「スタックアロケータ」を用いる。

スタックアロケータはシンプルで高速。フラグメンテーションも起こらない。ただし、個別のメモリ削除ができない。しかし、上記のような、同一フレームで使い切るデータに用いることを前提にすれば、毎フレームスタックを全リセットしても問題がない。

フレームアロケータの使用イメージ：



▼ 単一フレームアロケータ

基本的に、単純なフレームアロケータ（単一フレームアロケータ）に記録した情報は、毎フレームリセットされる。

次のフレームに持ち越したいようなデータを記録することができない点に注意が必要である。

▼ 二重フレームアロケータ

フレームアロケータ用のバッファをダブルバッファにして、１フレームごとに交互に切り替える手法がある。

これにより、フレームをまたいで、次のフレームにデータを持ち越すことができる。データの寿命を１フレーム延ばすことができるだけだが、十分使いどころはある。

例えば、メインスレッドから描画スレッドにリレーするデータは、フレーム境界をまたいで継続的に必要なデータだが、その次のフレームまでには処理が完了して不要になることを保証できる。

もし、フレームごとに使用するメモリ量にかたよりがあるなら、二重スタックアロケータを活用することで、バッファサイズの総計を抑えることができる。

▼ 一時スタックアロケータ

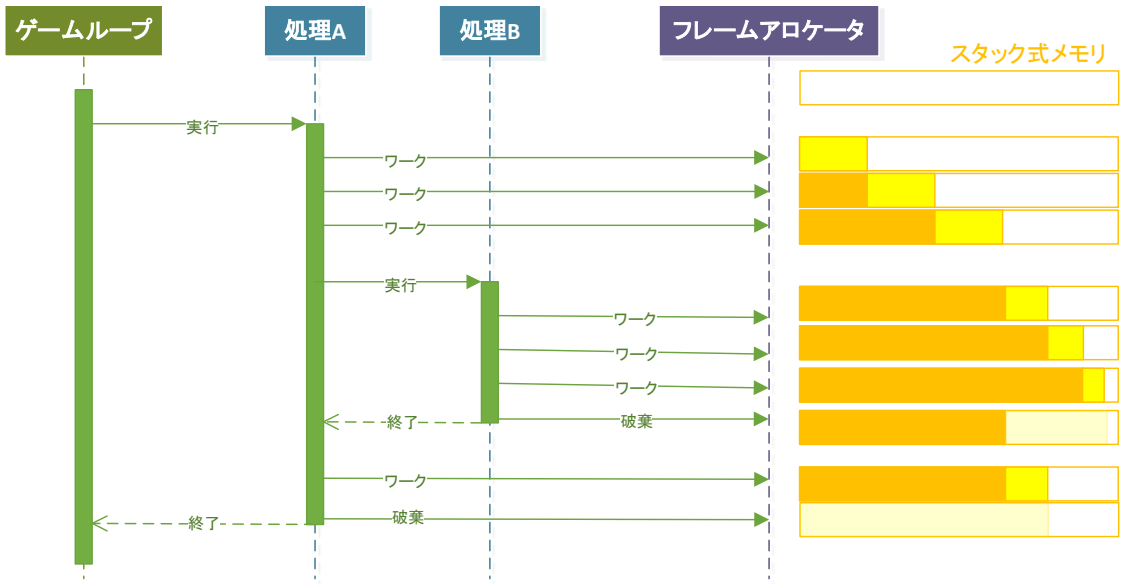
スタックアロケータは仕組みが単純で高速であるため、関数内で一時的に使用して、処理を終える時にまとめて破棄してしまうのも有効である。

処理としては、処理を開始した時のスタックのマーカーを記憶し、処理を終えるときに元の位置に戻すだけである。完全にワーク用の一時領域であり、通常のスタックフレーム（プログラム用のスタック領域）と用途はほぼ同じ。

ローカル変数で大きなバッファを使用するのをやめて、一時スタックアロケータの活用の幅を広げると、メモリの使用状況を確認し易くなるというメリットもある。例えば、「大きなメモリ確保が行われた時にトラップして、問題のある処理を検出する」といった処理を追加するようなことも手軽にできる。

また、STL などのライブラリー時スタックアロケータのメモリをつかって動作するようにすれば、安全性、生産性、処理速度を一挙に獲得できる。面倒なカスタムアロケータを作らずにそれを実践する方法を後述する。

一時スタックアロケータの使用イメージ：



■ 共通アロケータインターフェース

様々なアロケータが存在する環境では、メモリ確保を伴う処理の共通化がし難くなる。

この問題の解決策として、アロケータの実装を隠蔽した共通アロケータインターフェースを用意する。

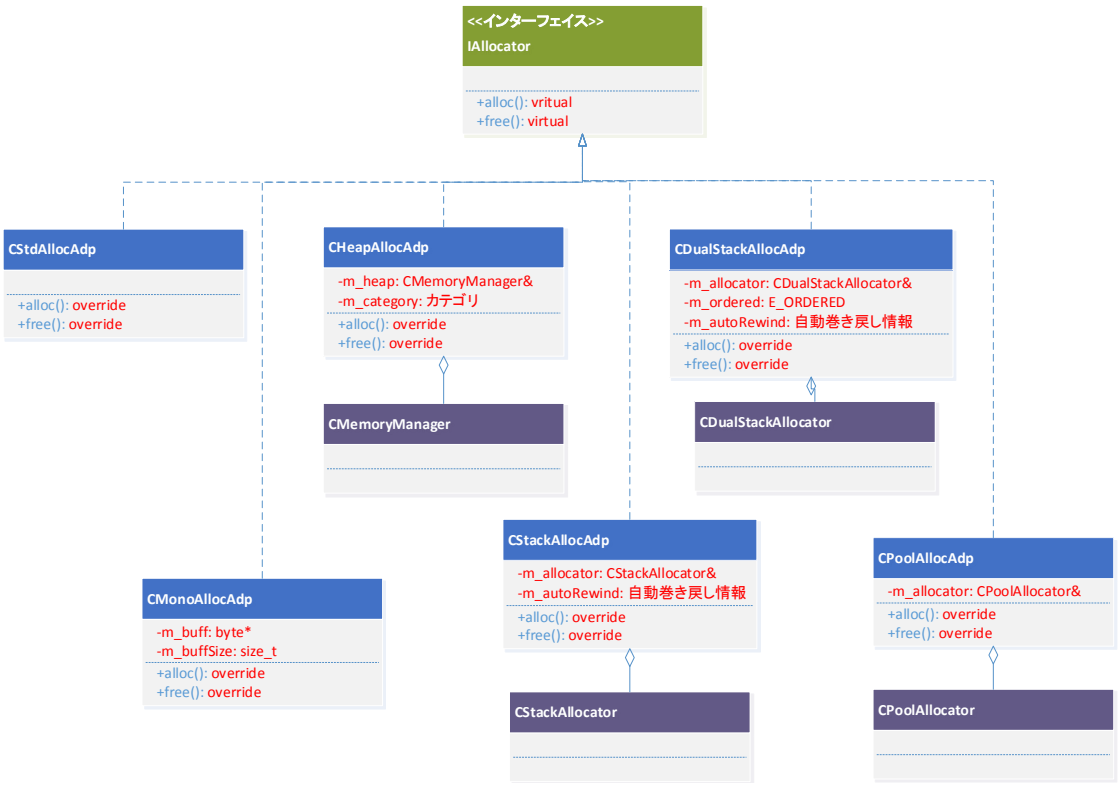
▼ 共通アロケータインターフェースの処理構造

共通アロケータインターフェースによる「アダプターパターン」を実践する。

デザインパターンの一つである「アダプターパターン」は、実装の異なる幾つかのクラスを、共通インターフェースを通して一様に扱えるようにする。（一般的なアダプターパターンの構造については、別紙の「デザインパターンの活用」で説明）

ここでは、実際のアロケータ（の参照）を内部に所持する「アロケータアダプタークラス」を、アロケータの種類数分用意する。それぞれが共通インターフェースとしてのメソッドを実装し、内部で実際のアロケータに処理を委譲する。

共通アロケータインターフェースのクラス図：



▼ 共通アロケータインターフェースのサンプル

共通アロケータインターフェースのサンプルプログラムを示す。

処理の状態を確認するために、配置 new / 配置 delete の処理内にはプリント文を仕込む。

なお、このサンプルプログラムは本書の以後の説明でも使用する。そのための準備として、本節の範囲を超える内容も含んだプログラムとなっている。「メモリ確保情報（デバッ

グ情報)」、「標準アロケータアダプターの new / delete」、「スタックアロケータアダプターの自動巻き戻し」については後述する。

共通アロケータインターフェースのサンプル：

【メモリ確保情報（デバッグ情報）】

```
//--------------------
//メモリ確保情報
//※デバッグ情報（型宣言のみ）
struct ALLOC_INFO;
```

【共通アロケータインターフェースクラス】

```
//--------------------
//共通アロケータインターフェースクラス
class IAllocator
{
public:
	//定数
	static const std::size_t DEFAULT_ALIGN = sizeof(int);//デフォルトのアラインメント
public:
	//メソッド
	virtual const char* getName() const = 0;//アロケータ名取得
	virtual std::size_t getTotal() const = 0;//全体メモリ量　※参考情報（正しい値とは限らない）
	virtual std::size_t getUsed() const = 0;//使用中メモリ量　※参考情報（正しい値とは限らない）
	virtual std::size_t getRemain() const = 0;//残りメモリ量　※参考情報（正しい値とは限らない）
	//メモリ確保
	virtual void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN,
	                                                                    const ALLOC_INFO* info = nullptr) = 0;
	//メモリ解放
	virtual void free(void* p) = 0;
public:
	//デストラクタ
	virtual ~IAllocator(){}
};
```

【標準アロケータアダプタークラス】※malloc() / free() による標準ヒープメモリ用

```
//--------------------
//標準アロケータアダプタークラス
//※内部で malloc, free を使用
//#define USE_MEMALIGN//GCC 用：memalign を使用する時はこのマクロを有効にする
#include <malloc.h>//malloc,_aligned_malloc 用
#include <stdlib.h>//memalign 用
class CStdAllocAdp : public IAllocator
{
public:
	//型
	typedef CStdAllocAdp ALLOCATOR_TYPE;//アロケータ型（便宜上自身を定義）
	typedef unsigned char byte;//バッファ用
public:
	//アクセッサ
	const char* getName() const override { return "CStdAllocAdp"; }//アロケータ名取得
	std::size_t getTotal() const override { return 0; }//全体メモリ量取得　※集計不可
	std::size_t getUsed() const override { return 0; }//使用中メモリ量取得　※集計不可
	std::size_t getRemain() const override { return 0; }//残りメモリ量取得　※集計不可
public:
	//オペレータ
	//※main 関数より前に何かしらのメモリ確保処理が実行された場合、
	//　CPolyAllocator が明示的にこのクラスを初期化する必要があるため、
	//　new / delete 演算子とインスタンス用の静的バッファを実装する
	static void* operator new(const std::size_t) throw(){ return m_buff; }//new 演算子
	static void operator delete(void*) throw() {}//delete 演算子
public:
	//メソッド
	//メモリ確保
```

```
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                                override
	{
		//return malloc(size);//通常 malloc()
	#ifdef USE_MEMALIGN
		return memalign(align, size);//GCC 用：アラインメント指定版 malloc
	#else//USE_MEMALIGN
		return _aligned_malloc(size, align);//MS 仕様：アラインメント指定版 malloc
	#endif//USE_MEMALIGN
	}
	//メモリ解放
	void free(void* p) override
	{
		if (!p)
			return;
	#ifdef USE_MEMALIGN
		::free(p);//通常 free() ※memalign で確保したメモリも free で解放
	#else//USE_MEMALIGN
		_aligned_free(p);//MS 仕様：アラインメント指定版 free
	#endif//USE_MEMALIGN
	}
public:
	//コンストラクタ
	CStdAllocAdp()
	{}
	//デストラクタ
	~CStdAllocAdp() override
	{}
private:
	//フィールド
	static byte m_buff[];
};
//標準アロケータアダプタークラスの静的変数インスタンス化
CStdAllocAdp::byte CStdAllocAdp::m_buff[sizeof(CStdAllocAdp)];
```

【単一バッファアロケータアダプタークラス】※単一のバッファを使用し、一回だけアロケートが可能

```
//--------------------
//単一バッファアロケータアダプタークラス
//※単なるバッファに１回だけアロケートするためのクラス
//※所定のバッファにインスタンスを生成したい場合に用いる
class CMonoAllocAdp : public IAllocator
{
public:
	//型
	//typedef CMonoAllocAdp ALLOCATOR_TYPE;//アロケータ型（アロケータなし）
	typedef unsigned char* byte;//バッファ用
public:
	//アクセッサ
	const char* getName() const override { return "CMonoAllocAdp"; }//アロケータ名取得
	std::size_t getTotal() const override { return m_buffSize; }//全体メモリ量取得
	std::size_t getUsed() const override { return m_isAllocated ? m_buffSize : 0; }//使用中メモリ量取得
	std::size_t getRemain() const override { return m_isAllocated ? 0 : m_buffSize; }//残りメモリ量取得
	const byte* getBuff() const { return m_buffPtr; }//バッファ取得
	std::size_t getBuffSize() const { return m_buffSize; }//バッファサイズ取得
	bool isAllocated() const { return m_isAllocated; }//メモリ確保済み
public:
	//メソッド
	//メモリ確保
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                                override
	{
		if (m_isAllocated)
			return nullptr;
		m_isAllocated = true;//メモリ確保済み
		return m_buffPtr;
```

```
	}
	//メモリ解放
	void free(void* p) override
	{
		if (!p)
			return;
		if (!m_isAllocated)
			return;
		m_isAllocated = false;//メモリ確保済み解除
	}
public:
	//デフォルトコンストラクタ
	CMonoAllocAdp() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CMonoAllocAdp(void* buff_p, const std::size_t buff_size) :
		m_buffPtr(reinterpret_cast<byte*>(buff_p)),//バッファアドレス
		m_buffSize(buff_size),//バッファサイズ
		m_isAllocated(false)//メモリ確保済み
	{}
	//デストラクタ
	~CMonoAllocAdp() override
	{}
private:
	//フィールド
	byte* m_buffPtr;//バッファアドレス
	const std::size_t m_buffSize;//バッファサイズ
	bool m_isAllocated;//メモリ確保済み
};
```

【バッファ付き単一バッファアロケータアダプターテンプレートクラス】

```
//--------------------
//バッファ付き単一バッファアロケータアダプターテンプレートクラス
template<std::size_t SIZE>
class CMonoAllocAdpWithBuff : public CMonoAllocAdp
{
public:
	//定数
	static const std::size_t BUFF_SIZE = SIZE;//バッファサイズ
public:
	//コンストラクタ
	CMonoAllocAdpWithBuff() :
		CMonoAllocAdp(m_buff, BUFF_SIZE)
	{}
	//デストラクタ
	~CMonoAllocAdpWithBuff() override
	{}
private:
	//フィールド
	byte m_buff[BUFF_SIZE];//バッファ
};
```

【スタックアロケータインターフェースアダプタークラス】※スタックアロケータ／双方向スタックアロケータ両用のアダプター

```
//--------------------
//スタックアロケータインターフェースアダプタークラス
//※スタックアロケータ／双方向スタックアロケータ両用のアダプター
class CIStackAllocAdp : public IAllocator
{
public:
	//型
	typedef IStackAllocator ALLOCATOR_TYPE;//アロケータ型
public:
	//定数
	//自動巻き戻し
	enum E_AUTO_REWIND
	{
```

```
            NOREWIND = 0,//巻き戻しなし
            AUTO_REWIND = 1,//指定のスタック順に巻き戻し
            BOTH_AUTO_REWIND = 2,//両方向に巻き戻し（双方向スタックアロケータのみ）
      };
public:
      //アクセッサ
      const char* getName() const override { return "CIStackAllocAdp"; }//アロケータ名取得
      std::size_t getTotal() const override { return m_allocator.getTotal(); }//全体メモリ量取得
      std::size_t getUsed() const override { return m_allocator.getUsed(); }//使用中メモリ量取得
      std::size_t getRemain() const override { return m_allocator.getRemain(); }//残りメモリ量取得
      IStackAllocator& getAllocator(){ return m_allocator; }//アロケータ取得
      const IStackAllocator& getAllocator() const { return m_allocator; }//アロケータ取得
      E_AUTO_REWIND getAutoRewind() const { m_autoRewind; }//自動巻き戻し指定取得
      void setAutoRewind(const E_AUTO_REWIND auto_rewind){ m_autoRewind = auto_rewind; }//自動巻き戻し指定更新
      IStackAllocator::marker_t getRewindMarker() const { return m_rewindMarker; }//巻き戻しマーカー取得
public:
      //メソッド
      //メモリ確保
      void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                                        override
      {
            return m_allocator.alloc(size, align);//virtual メンバーを使う
      }
      //メモリ解放
      void free(void* p) override
      {
            //なにもしない
      }
public:
      //デフォルトコンストラクタ
      CIStackAllocAdp() = delete;//コンストラクタ引数必須
      //コンストラクタ
      CIStackAllocAdp(IStackAllocator& allocator, const E_AUTO_REWIND auto_rewind = NOREWIND) :
            m_allocator(allocator),//スタックアロケータ
            m_autoRewind(auto_rewind)//自動巻き戻し指定
      {
            //自動巻き戻し位置記憶
            m_rewindMarker = m_allocator.getMarker();
      }
      //デストラクタ
      ~CIStackAllocAdp() override
      {
            //自動巻き戻し
            if (m_autoRewind == AUTO_REWIND)
            {
                  m_allocator.back(m_rewindMarker);
            }
      }
protected:
      //フィールド
      IStackAllocator& m_allocator;//スタックアロケータ
      E_AUTO_REWIND m_autoRewind;//自動巻き戻し指定
      IStackAllocator::marker_t m_rewindMarker;//巻き戻しマーカー
};
```

【スタックアロケータアダプタークラス】※スタックアロケータ用のアダプター

```
//--------------------
//スタックアロケータアダプター
//※スタックアロケータ用のアダプター
class CStackAllocAdp : public CIStackAllocAdp
{
public:
      //型
      typedef CStackAllocator ALLOCATOR_TYPE;//アロケータ型
public:
```

```
	//アクセッサ
	const char* getName() const override { return "CStackAllocAdp"; }//アロケータ名取得
	CStackAllocator& getAllocator(){ return *static_cast<CStackAllocator*>(&m_allocator); }//アロケータ取得
	const CStackAllocator& getAllocator() const { return *static_cast<CStackAllocator*>(&m_allocator); }
                                                                                    //アロケータ取得
public:
	//メソッド
	//メモリ確保
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                        override
	{
		CStackAllocator& allocator = getAllocator();
		return allocator.allocN(size, align);//非 virtual メンバーを使う
	}
	//メモリ解放
	void free(void* p) override
	{
		//なにもしない
	}
public:
	//デフォルトコンストラクタ
	CStackAllocAdp() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CStackAllocAdp(CStackAllocator& allocator, const E_AUTO_REWIND auto_rewind = NOREWIND) :
		CIStackAllocAdp(allocator, auto_rewind)//スタックアロケータ
	{}
	//デストラクタ
	~CStackAllocAdp() override
	{}
};
```

【双方向スタックアロケータアダプタークラス】※双方向スタックアロケータ用のアダプター

```
//--------------------
//双方向スタックアロケータアダプタークラス
//※双方向スタックアロケータ用のアダプター
class CDualStackAllocAdp : public CIStackAllocAdp
{
public:
	//型
	typedef CDualStackAllocator ALLOCATOR_TYPE;//アロケータ型
public:
	//アクセッサ
	const char* getName() const override { return "CDualStackAllocAdp"; }//アロケータ名取得
	CDualStackAllocator& getAllocator(){ return *static_cast<CDualStackAllocator*>(&m_allocator); }//アロケータ取得
	const CDualStackAllocator& getAllocator() const { return *static_cast<CDualStackAllocator*>(&m_allocator); }
                                                                                    //アロケータ取得
	IStackAllocator::E_ORDERED getOrdered() const { return m_ordered; }//スタック順取得
	void setOrdered(const IStackAllocator::E_ORDERED ordered){ m_ordered = ordered; }//スタック順更新
	IStackAllocator::marker_t getRewindMarkerN() const { return m_rewindMarker; }//巻き戻しマーカー（正順）取得
	IStackAllocator::marker_t getRewindMarkerR() const { return m_rewindMarkerR; }//巻き戻しマーカー（逆順）取得
public:
	//メソッド
	//メモリ確保
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                        override
	{
		CDualStackAllocator& allocator = getAllocator();
		return allocator.alloc(m_ordered, size, align);//非 virtual メンバーを使う
	}
	//メモリ解放
	void free(void* p) override
	{
		//なにもしない
	}
public:
```

```
	//デフォルトコンストラクタ
	CDualStackAllocAdp() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CDualStackAllocAdp(CDualStackAllocator& allocator,
	                                    const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT,
	                                                                const E_AUTO_REWIND auto_rewind = NOREWIND) :
		CIStackAllocAdp(allocator, auto_rewind),//双方向スタックアロケータ
		m_ordered(ordered)//スタック順
	{
		//自動巻き戻し位置記憶
		m_rewindMarker = allocator.getMarkerN();//正順
		m_rewindMarkerR = allocator.getMarkerR();//逆順
	}
	//デストラクタ
	~CDualStackAllocAdp() override
	{
		//自動巻き戻し
		if (m_autoRewind == BOTH_AUTO_REWIND)
		{
			//両方向の巻き戻し
			CDualStackAllocator& allocator = getAllocator();
			allocator.backN(m_rewindMarker);//正順
			allocator.backR(m_rewindMarkerR);//逆順
		}
		else if (m_autoRewind == AUTO_REWIND)
		{
			//単方向の巻き戻し
			CDualStackAllocator& allocator = getAllocator();
			if (m_ordered == IStackAllocator::NORMAL)//正順スタックのみ
				allocator.backN(m_rewindMarker);
			else if (m_ordered == IStackAllocator::REVERSE)//逆順スタックのみ
				allocator.backR(m_rewindMarkerR);
			else//if (m_ordered == IStackAllocator::DEFAULT)//アロケータのデフォルトスタック順に従う
			{
				//allocator.backD(m_rewindMarker);//このメソッドは使わない
				if (allocator.getDefaultOrdered() == IStackAllocator::REVERSE)//逆順スタックのみ
					allocator.backR(m_rewindMarkerR);
				else//if (allocator.getDefaultOrdered() == IStackAllocator::NORMAL)//正順スタックのみ
					allocator.backN(m_rewindMarker);
			}
		}
		m_autoRewind = NOREWIND;//親クラスのデストラクタ処理で問題が起こらないように設定を無効化する
	}
private:
	//フィールド
	IStackAllocator::E_ORDERED m_ordered;//スタック順
	//IStackAllocator::marker_t m_rewindMarker;//巻き戻しマーカー（正順）※親クラスの変数を使用
	IStackAllocator::marker_t m_rewindMarkerR;//巻き戻しマーカー（逆順）
};
```

【プールアロケータアダプタークラス】※プールアロケータ用のアダプター

```
//--------------------
//プールアロケータアダプタークラス
//※プールアロケータ用のアダプター
class CPoolAllocAdp : public IAllocator
{
public:
	//型
	typedef CPoolAllocator ALLOCATOR_TYPE;//アロケータ型
public:
	//アクセッサ
	const char* getName() const override { return "CPoolAllocAdp"; }//アロケータ名取得
	std::size_t getTotal() const override { return m_allocator.getBlocksNum() * m_allocator.getBlockSize(); }
	                                                                                    //全体メモリ量取得
	std::size_t getUsed() const override { return m_allocator.getUsed() * m_allocator.getBlockSize(); }
```

```
                                                                        //使用中メモリ量取得
    std::size_t getRemain() const override { return m_allocator.getRemain() * m_allocator.getBlockSize(); }
                                                                        //残りメモリ量取得
    CPoolAllocator& getAllocator() const { return m_allocator; }//アロケータ取得
public:
    //メソッド
    //メモリ確保
    void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                    override
    {
        return m_allocator.alloc(size);
    }
    //メモリ解放
    void free(void* p) override
    {
        return m_allocator.free(p);
    }
public:
    //デフォルトコンストラクタ
    CPoolAllocAdp() = delete;//コンストラクタ引数必須
    //コンストラクタ
    CPoolAllocAdp(CPoolAllocator& allocator) :
        m_allocator(allocator)//プールアロケータ
    {}
    //デストラクタ
    ~CPoolAllocAdp() override
    {}
private:
    //フィールド
    CPoolAllocator& m_allocator;//プールアロケータ
};
```

【共通アロケータインターフェース用配置 new / 配置 delete 定義】※上記の全アダプターに適合する

```
//--------------------
//配置 new
void* operator new(const std::size_t size, IAllocator& allocator) throw()
{
    printf("placement new(size=%d, i_allocator=¥"%s¥":%d/%d/%d)¥n", size, allocator.getName(), allocator.getTotal(),
                                                                    allocator.getUsed(), allocator.getRemain());
    void* p = allocator.alloc(size);
    printf("  p=0x%p¥n", p);
    return p;
}
//配列版
void* operator new[](const std::size_t size, IAllocator& allocator) throw()
{
    printf("placement new[](size=%d, i_allocator=¥"%s¥":%d/%d/%d)¥n", size, allocator.getName(), allocator.getTotal(),
                                                                    allocator.getUsed(), allocator.getRemain());
    void* p = allocator.alloc(size);
    printf("  p=0x%p¥n", p);
    return p;
}
//配置 delete
void operator delete(void* p, IAllocator& allocator) throw()
{
    printf("placement delete(p=0x%p, i_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator.getName(), allocator.getTotal(),
                                                                    allocator.getUsed(), allocator.getRemain());
    allocator.free(p);
}
//配列版
void operator delete[](void* p, IAllocator& allocator) throw()
{
    printf("placement delete[](p=0x%p, i_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator.getName(), allocator.getTotal(),
                                                                    allocator.getUsed(), allocator.getRemain());
    allocator.free(p);
```

```
}
```

【共通アロケータインターフェース用 delete 関数】※上記の全アダプターに適合する

```
//デストラクタ呼び出し付き delete
template<class T>
void delete_ptr(T*& p, IAllocator& allocator)
{
	printf("delete_ptr(p=0x%p, i_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator.getName(), allocator.getTotal(),
		allocator.getUsed(), allocator.getRemain());
	if (!p)
		return;
	p->~T();//デストラクタ呼び出し
	operator delete(p, allocator);
	p = nullptr;//安全のためにポインタを初期化
}
//配列版
template<class T>
void delete_ptr(T*& p, const std::size_t array_num, IAllocator& allocator)
{
	printf("delete_ptr(p=0x%p, array_num=%d, i_allocator=¥"%s¥":%d/%d/%d)¥n", p, array_num, allocator.getName(),
		allocator.getTotal(), allocator.getUsed(), allocator.getRemain());
	if (!p)
		return;
	for (std::size_t i = 0; i < array_num; ++i)
		p->~T();//デストラクタ呼び出し
	operator delete[](p, allocator);
	p = nullptr;//安全のためにポインタを初期化
}
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest6
{
public:
	//デフォルトコンストラクタ
	CTest6() :
		CTest6("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
		//m_name("(default)")//C++11 が使えない場合は普通に初期化
	{
		printf("CTest6::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest6(const char* name) :
		m_name(name)
	{
		printf("CTest6::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	virtual ~CTest6()
	{
		printf("CTest6::Destructor : name=¥"%s¥"¥n", m_name);
	}
protected:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【多重継承テスト用クラス定義】

```
//--------------------
//テスト用クラス（多重継承テスト用）
class CTest6ex
{
public:
```

```
	//コンストラクタ
	CTest6ex()
	{
		printf("CTest6ex::Constructor¥n");
	}
	//デストラクタ
	virtual ~CTest6ex()
	{
		printf("CTest6ex::Destructor¥n");
	}
protected:
	//フィールド
	int m_dummy;//ダミー
};
//--------------------
//テスト用クラス（多重継承）
class CDerivedTest6 : public CTest6, public CTest6ex
{
public:
	//デフォルトコンストラクタ
	CDerivedTest6() :
		CTest6(),
		CTest6ex()
	{
		printf("CDerivedTest6::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CDerivedTest6(const char* name) :
		CTest6(name),
		CTest6ex()
	{
		printf("CDerivedTest6::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CDerivedTest6() override
	{
		printf("CDerivedTest6::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	int m_dummy;//ダミー
};
```

【テスト（共通関数）】

```
//--------------------
//テスト（共通関数）
//※実際のアロケータを切り替えて共通処理を実行するテスト
//クラス単体のテスト
void test6_sub1(const char* name, IAllocator& allocator)
{
	CTest6* obj_p = new(allocator)CTest6(name);
	printf("obj_p=0x%p¥n", obj_p);
	delete_ptr(obj_p, allocator);
}
//クラスの配列のテスト
void test6_sub2(IAllocator& allocator)
{
	CTest6* obj_p = new(allocator)CTest6[3];
	printf("obj_p=0x%p¥n", obj_p);
	delete_ptr(obj_p, 3, allocator);
}
//多重継承クラスのテスト
void test6_sub3(const char* name, IAllocator& allocator)
{
	CTest6ex* obj_p = new(allocator)CDerivedTest6(name);//アップキャスト
```

```
	printf("obj_p=0x%p¥n", obj_p);
	delete_ptr(obj_p, allocator);
}
```

【テスト】

```
//--------------------
//テスト
void test6()
{
	//標準アロケータ (malloc / free) 使用
	{
		printf("----------CStdAllocAdp¥n");
		CStdAllocAdp allocator_adp;
		test6_sub1("テスト 6-1a", allocator_adp);
		//test6_sub2(allocator_adp);//クラスの配列でずれたポインタをそのまま free するのでハングする
		//test6_sub3("テスト 6-1b", allocator_adp);//多重継承でずれたポインタをそのまま free するのでハングする
	}
	//単一バッファアロケータ使用
	{
		printf("----------CMonoAllocAdp¥n");
		CMonoAllocAdpWithBuff<128> allocator_adp;
		test6_sub1("テスト 6-2a", allocator_adp);
		test6_sub2(allocator_adp);
		test6_sub3("テスト 6-2b", allocator_adp);
	}
	//スタックアロケータ使用
	{
		printf("----------CStackAllocAdp¥n");
		CStackAllocatorWithBuff<128> allocator;
		CStackAllocAdp allocator_adp(allocator);
		test6_sub1("テスト 6-3a", allocator_adp);
		{
			//自動巻き戻しのテスト
			CStackAllocAdp allocator_adp_tmp(allocator, CIStackAllocAdp::AUTO_REWIND);
			test6_sub2(allocator_adp_tmp);
		}
		test6_sub3("テスト 6-3b", allocator_adp);
	}
	//双方向スタックアロケータ使用
	{
		printf("----------CDualStackAllocAdp¥n");
		CDualStackAllocatorWithBuff<128> allocator;
		CDualStackAllocAdp allocator_adp(allocator, DSA_REVERSE);
		test6_sub1("テスト 6-4a", allocator_adp);
		{
			//自動巻き戻しのテスト
			CDualStackAllocAdp allocator_adp_tmp(allocator, DSA_REVERSE, CIStackAllocAdp::AUTO_REWIND);
			test6_sub2(allocator_adp_tmp);
		}
		test6_sub3("テスト 6-4b", allocator_adp);
	}
	//スタックアロケータをスタックアロケータインターフェースとして使用
	{
		printf("----------CIStackAllocAdp on CStackAllocAdp¥n");
		CStackAllocatorWithBuff<128> allocator;
		CIStackAllocAdp allocator_adp(allocator);
		test6_sub1("テスト 6-5a", allocator_adp);
		{
			//自動巻き戻しのテスト
			CIStackAllocAdp allocator_adp_tmp(allocator, CIStackAllocAdp::AUTO_REWIND);
			test6_sub2(allocator_adp_tmp);
		}
		test6_sub3("テスト 6-5b", allocator_adp);
	}
	//双方向スタックアロケータをスタックアロケータインターフェースとして使用
```

```
    {
        printf("----------CIStackAllocAdp on CDualStackAllocAdp¥n");
        CDualStackAllocatorWithBuff<128> allocator;
        CIStackAllocAdp allocator_adp(allocator);
        test6_sub1("テスト 6-6a", allocator_adp);
        {
            //自動巻き戻しのテスト
            CIStackAllocAdp allocator_adp_tmp(allocator, CIStackAllocAdp::AUTO_REWIND);
            test6_sub2(allocator_adp_tmp);
        }
        test6_sub3("テスト 6-6b", allocator_adp);
    }
    //プールアロケータ使用
    {
        printf("----------CPoolAllocAdp¥n");
        CPoolAllocatorWithBuff<24, 5> allocator;
        CPoolAllocAdp allocator_adp(allocator);
        test6_sub1("テスト 6-7a", allocator_adp);
        test6_sub2(allocator_adp);
        test6_sub3("テスト 6-7b", allocator_adp);
    }
}
```

↓（実行結果）

```
----------CStdAllocAdp
placement new(size=12, i_allocator="CStdAllocAdp":0/0/0)     ←アロケータ名:メモリ総量/使用量/残量を表示
  p=0x0054B7A0                                                 （以降、赤字で示す）
CTest6::Constructor : name="テスト 6-1a"                       ※標準アロケータ(malloc)は総量/使用量/残量不明
obj_p=0x0054B7A0
delete_ptr(p=0x0054B7A0, i_allocator="CStdAllocAdp":0/0/0)
CTest6::Destructor : name="テスト 6-1a"
placement delete(p=0x0054B7A0, i_allocator="CStdAllocAdp":0/0/0)
                                                             ←配列確保と多重継承はテストしない
                                                               delete 時のポインタのずれの問題を解消できていないため
----------CMonoAllocAdp
placement new(size=12, i_allocator="CMonoAllocAdp":128/0/128)          ←単一バッファアロケータで共通処理が
  p=0x0033FBA8                                                           実行できている
CTest6::Constructor : name="テスト 6-2a"
obj_p=0x0033FBA8
delete_ptr(p=0x0033FBA8, i_allocator="CMonoAllocAdp":128/128/0)        ←メモリ確保の要求サイズによらず、
CTest6::Destructor : name="テスト 6-2a"                                  バッファ全域を消費している
placement delete(p=0x0033FBA8, i_allocator="CMonoAllocAdp":128/128/0)
placement new[](size=40, i_allocator="CMonoAllocAdp":128/0/128)        ←メモリが解放されて残量が増えている
  p=0x0033FBA8
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
obj_p=0x0033FBAC                                                       ←配列確保によりポインタがずれている
delete_ptr(p=0x0033FBAC, array_num=3, i_allocator="CMonoAllocAdp":128/128/0)
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
placement delete[](p=0x0033FBAC, i_allocator="CMonoAllocAdp":128/128/0) ←ずれたポインタのまま delete
placement new(size=24, i_allocator="CMonoAllocAdp":128/0/128)              （以後の処理も同様）
  p=0x0033FBA8
CTest6::Constructor : name="テスト 6-2b"
CTest6ex::Constructor
CDerivedTest6::Constructor : name="テスト 6-2b"
obj_p=0x0033FBB4                                                       ←多重継承クラスのアップキャストにより
delete_ptr(p=0x0033FBB4, i_allocator="CMonoAllocAdp":128/128/0)          ポインタがずれている
CDerivedTest6::Destructor : name="テスト 6-2b"
CTest6ex::Destructor
```

```
CTest6::Destructor : name="テスト6-2b"
placement delete(p=0x0033FBB4, i_allocator="CMonoAllocAdp":128/128/0)    ←ずれたポインタのまま delete
----------CStackAllocAdp                                                    （以後の処理も同様）
placement new(size=12, i_allocator="CStackAllocAdp":128/0/128)           ←スタックアロケータで共通処理が
  p=0x0033FB10                                                             実行できている
CTest6::Constructor : name="テスト6-3a"
obj_p=0x0033FB10
delete_ptr(p=0x0033FB10, i_allocator="CStackAllocAdp":128/12/116)
CTest6::Destructor : name="テスト6-3a"
placement delete(p=0x0033FB10, i_allocator="CStackAllocAdp":128/12/116)
placement new[](size=40, i_allocator="CStackAllocAdp":128/12/116)        ←スタックアロケータのメモリは解放されない
  p=0x0033FB1C                                                             （以後のスタック系処理も同様）
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
obj_p=0x0033FB20
delete_ptr(p=0x0033FB20, array_num=3, i_allocator="CStackAllocAdp":128/52/76)
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
placement delete[](p=0x0033FB20, i_allocator="CStackAllocAdp":128/52/76)
placement new(size=24, i_allocator="CStackAllocAdp":128/12/116)          ←自動巻き戻し機能により、スタックの残量が
  p=0x0033FB1C                                                             配列のメモリ確保テスト前の状態に戻っている
CTest6::Constructor : name="テスト6-3b"                                    （以後のスタック系処理も同様）
CTest6ex::Constructor
CDerivedTest6::Constructor : name="テスト6-3b"
obj_p=0x0033FB28
delete_ptr(p=0x0033FB28, i_allocator="CStackAllocAdp":128/36/92)
CDerivedTest6::Destructor : name="テスト6-3b"
CTest6ex::Destructor
CTest6::Destructor : name="テスト6-3b"
placement delete(p=0x0033FB28, i_allocator="CStackAllocAdp":128/36/92)
----------CDualStackAllocAdp
placement new(size=12, i_allocator="CDualStackAllocAdp":128/0/128)       ←双方向スタックアロケータで共通処理が
  p=0x0033FABC                                                             実行できている
CTest6::Constructor : name="テスト6-4a"
obj_p=0x0033FABC
delete_ptr(p=0x0033FABC, i_allocator="CDualStackAllocAdp":128/12/116)
CTest6::Destructor : name="テスト6-4a"
placement delete(p=0x0033FABC, i_allocator="CDualStackAllocAdp":128/12/116)
placement new[](size=40, i_allocator="CDualStackAllocAdp":128/12/116)
  p=0x0033FA94
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
obj_p=0x0033FA98
delete_ptr(p=0x0033FA98, array_num=3, i_allocator="CDualStackAllocAdp":128/52/76)
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
placement delete[](p=0x0033FA98, i_allocator="CDualStackAllocAdp":128/52/76)
placement new(size=24, i_allocator="CDualStackAllocAdp":128/12/116)
  p=0x0033FAA4
CTest6::Constructor : name="テスト6-4b"
CTest6ex::Constructor
CDerivedTest6::Constructor : name="テスト6-4b"
obj_p=0x0033FAB0
delete_ptr(p=0x0033FAB0, i_allocator="CDualStackAllocAdp":128/36/92)
CDerivedTest6::Destructor : name="テスト6-4b"
```

```
CTest6ex::Destructor
CTest6::Destructor : name="テスト6-4b"
placement delete(p=0x0033FAB0, i_allocator="CDualStackAllocAdp":128/36/92)
----------CIStackAllocAdp on CStackAllocAdp
placement new(size=12, i_allocator="CIStackAllocAdp":128/0/128)          ←スタックアロケータインターフェース（実体は
  p=0x0033F968                                                             スタックアロケータ）で共通処理が実行
CTest6::Constructor : name="テスト6-5a"                                    できている
obj_p=0x0033F968
delete_ptr(p=0x0033F968, i_allocator="CIStackAllocAdp":128/12/116)
CTest6::Destructor : name="テスト6-5a"
placement delete(p=0x0033F968, i_allocator="CIStackAllocAdp":128/12/116)
placement new[](size=40, i_allocator="CIStackAllocAdp":128/12/116)
  p=0x0033F974
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
obj_p=0x0033F978
delete_ptr(p=0x0033F978, array_num=3, i_allocator="CIStackAllocAdp":128/52/76)
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
placement delete[](p=0x0033F978, i_allocator="CIStackAllocAdp":128/52/76)
placement new(size=24, i_allocator="CIStackAllocAdp":128/12/116)
  p=0x0033F974
CTest6::Constructor : name="テスト6-5b"
CTest6ex::Constructor
CDerivedTest6::Constructor : name="テスト6-5b"
obj_p=0x0033F980
delete_ptr(p=0x0033F980, i_allocator="CIStackAllocAdp":128/36/92)
CDerivedTest6::Destructor : name="テスト6-5b"
CTest6ex::Destructor
CTest6::Destructor : name="テスト6-5b"
placement delete(p=0x0033F980, i_allocator="CIStackAllocAdp":128/36/92)
----------CIStackAllocAdp on CDualStackAllocAdp
placement new(size=12, i_allocator="CIStackAllocAdp":128/0/128)          ←スタックアロケータインターフェース（実体は
  p=0x0033F8A0                                                             双方向スタックアロケータ）で共通処理が実行
CTest6::Constructor : name="テスト6-6a"                                    できている
obj_p=0x0033F8A0
delete_ptr(p=0x0033F8A0, i_allocator="CIStackAllocAdp":128/12/116)
CTest6::Destructor : name="テスト6-6a"
placement delete(p=0x0033F8A0, i_allocator="CIStackAllocAdp":128/12/116)
placement new[](size=40, i_allocator="CIStackAllocAdp":128/12/116)
  p=0x0033F8AC
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
CTest6::Constructor : name="(default)"
CTest6::DefaultConstructor : name="(default)"
obj_p=0x0033F8B0
delete_ptr(p=0x0033F8B0, array_num=3, i_allocator="CIStackAllocAdp":128/52/76)
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
CTest6::Destructor : name="(default)"
placement delete[](p=0x0033F8B0, i_allocator="CIStackAllocAdp":128/52/76)
placement new(size=24, i_allocator="CIStackAllocAdp":128/12/116)
  p=0x0033F8AC
CTest6::Constructor : name="テスト6-6b"
CTest6ex::Constructor
CDerivedTest6::Constructor : name="テスト6-6b"
obj_p=0x0033F8B8
delete_ptr(p=0x0033F8B8, i_allocator="CIStackAllocAdp":128/36/92)
```

```
CDerivedTest6::Destructor : name="テスト6-6b"
CTest6ex::Destructor
CTest6::Destructor : name="テスト6-6b"
placement delete(p=0x0033F8B8, i_allocator="CIStackAllocAdp":128/36/92)
----------CPoolAllocAdp
placement new(size=12, i_allocator="CPoolAllocAdp":120/0/120)          ←プールアロケータで共通処理が実行できている
  p=0x0033F7D8
CTest6::Constructor : name="テスト6-7a"
obj_p=0x0033F7D8
delete_ptr(p=0x0033F7D8, i_allocator="CPoolAllocAdp":120/24/96)        ←メモリ確保の要求サイズによらず、
CTest6::Destructor : name="テスト6-7a"                                   ブロックサイズ分のメモリを消費している
placement delete(p=0x0033F7D8, i_allocator="CPoolAllocAdp":120/24/96)
placement new[](size=40, i_allocator="CPoolAllocAdp":120/0/120)        ←メモリが解放されて残量が増えている
  p=0x00000000                                                         ←ブロックサイズを超えるメモリ要求につき、
obj_p=0x00000000                                                         残量が十分あるのにメモリ確保失敗
delete_ptr(p=0x00000000, array_num=3, i_allocator="CPoolAllocAdp":120/0/120)      ※配列のメモリ確保は一塊のメモリで
placement new(size=24, i_allocator="CPoolAllocAdp":120/0/120)                       あることに注意
  p=0x0033F7F0
CTest6::Constructor : name="テスト6-7b"
CTest6ex::Constructor
CDerivedTest6::Constructor : name="テスト6-7b"
obj_p=0x0033F7FC
delete_ptr(p=0x0033F7FC, i_allocator="CPoolAllocAdp":120/24/96)        ←ブロックサイズぴったりのメモリ確保はOK
CDerivedTest6::Destructor : name="テスト6-7b"
CTest6ex::Destructor
CTest6::Destructor : name="テスト6-7b"
placement delete(p=0x0033F7FC, i_allocator="CPoolAllocAdp":120/24/96)
```

▼ 一時スタックアロケータの実装

サンプルを先に示す形となったが、「アロケータアダプタークラス」を利用し、「スタックアロケータ」を「一時スタックアロケータ」として活用する機能を追加する。

一時スタックアロケータは、先に説明している通り、スタックアロケータの活用の一つである。処理の最初にマーカーを記憶し、処理を抜ける時に元の位置に戻すだけ。

「スタックアロケータアダプター」をローカル変数として使用することを前提に、この動作を自動的に行う仕組みを実装する。

スタックアロケータアダプターのコンストラクタで、「自動巻き戻し」を「有効」に設定すると、その時点のマーカーを内部に保持し、デストラクタで元に戻す。デフォルトは「無効」であり、意図的に使用しない限り自動巻き戻しは行われない。

一時スタックアロケータのサンプル：

【一時スタックアロケータ定義】

```
//一時スタックアロケータ（グローバル変数）
static CStackAllocatorWithBuff<1 * 1024 * 1024> g_tempStack;
```

【一時スタックアロケータを使用】

```
//処理ブロック
{
    //スタックアロケータアダプター（自動巻き戻し指定）
    CStackAllocAdp alloc_adp(g_tempStack, CIStackAllocAdp::AUTO_REWIND);
    CTest6* obj_p = new(alloc_adp)CTest6("テスト");
    char* buff = new(alloc_adp)char[1024];
    //…（処理）…  ※メモリ解放不要
}
```

```
//処理ブロックを抜ける時に、alloc_adp のデストラクタにより、自動的にマーカーが元の位置に戻る
```

■ STL などの標準ライブラリを便利に活用するテクニック

本書の最大の目的である標準ライブラリの活用テクニックについて説明する。

▼ グローバル new /delete と共通アロケータインターフェース

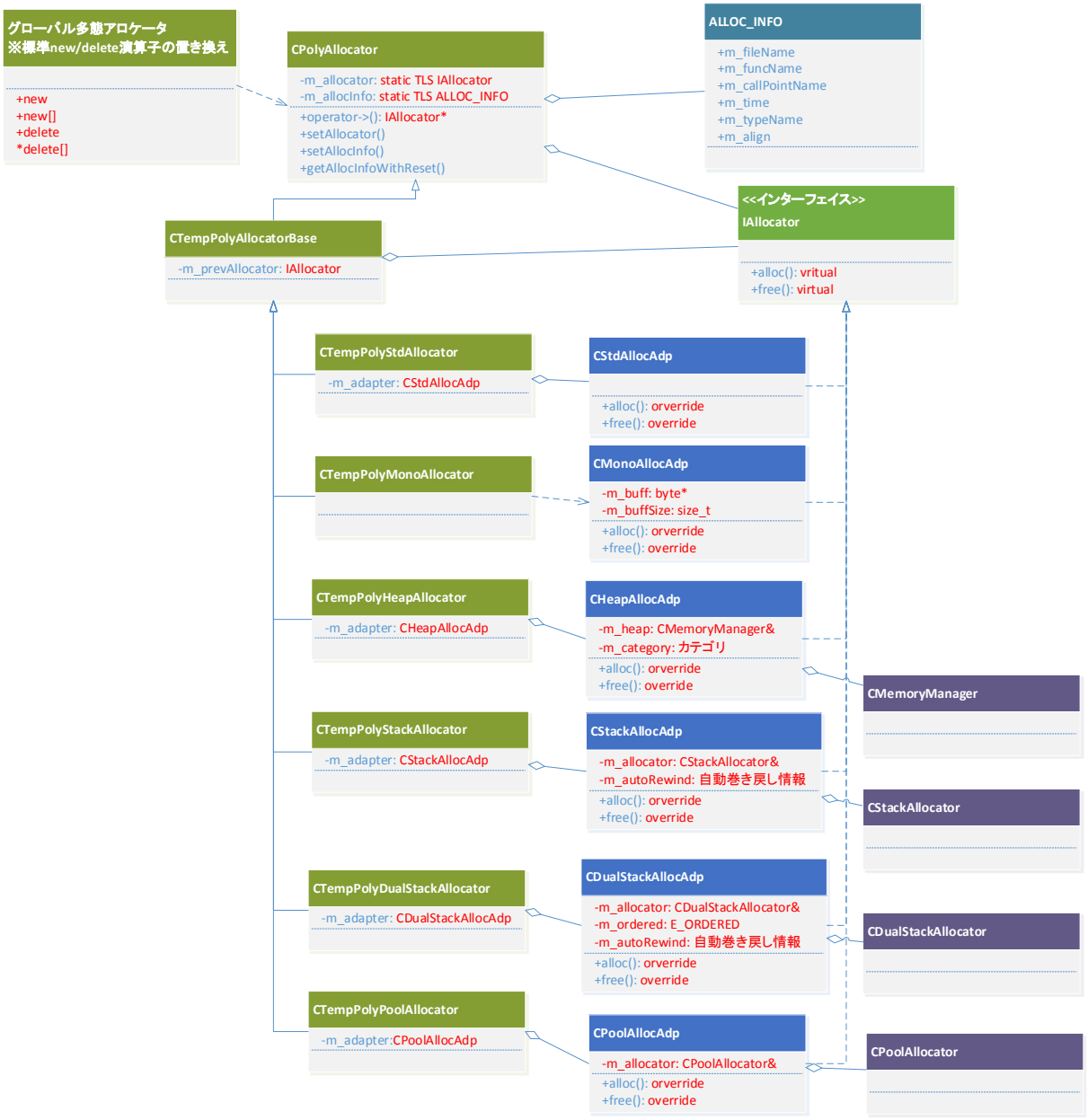
まず、ここまでに、配置 new / 配置 delete の問題点と、その解決策として、グローバル new / delete の置き換えが有効であることを説明した。しかし、様々なアロケータを利用するためには配置 new / 配置 delete が不可欠であったため、簡単に解決できる問題ではなかった。

しかし、グローバル new / delete の中で「共通アロケータインターフェース」を用いれば、グローバル new /delete の恩恵を受けつつ、様々なアロケータを混在して使用することができる。

▼ グローバル new /delete の多態性（ポリモーフィズム）の実現

グローバル new / delete の処理を独自に置き換え、その処理の中で「共通アロケータインターフェース」を用いたメモリの確保／破棄を行う。これにより、グローバル new / delete の「多態性」（ポリモーフィズム）を実現する。

多態アロケータのクラス図：



「グローバル多態アロケータ」として、標準の new / delete 演算子を置き換えた処理を用意する。処理内部では「多態アロケータクラス」を使用してメモリの確保と破棄を行う。

「多態アロケータ」は内部に静的変数で現在のアロケータ（アロケータインターフェースを実装したアダプター）を管理する。その静的変数は、TLS（スレッドローカルストレージ）によりスレッドごとに管理し、スレッド間で影響を及ぼし合わないようにする。ロックの必要もない。

さらに、多態アロケータにアロケータをセットし、処理を終えたら元に戻すための「一時多態アロケータ」を扱う。

▼ グローバル多態アロケータのサンプル

グローバル多態アロケータのサンプルプログラムを示す。

重要なポイントは二つ。

まず、配置 delete とデストラクタの明示的な呼び出しをせずに、delete 演算子で処理している点。デストラクタもきちんと実行され、かつ、配列 new および 多重継承のアップキャストによるポインタのずれも解消される。

もう一点は、前述のとおり、アダプター（共通アロケータインターフェース）の受け渡しに TLS を使用している点である。これだけで、この仕組みをスレッドセーフ、かつ、ロックフリーにしている。(TLS についての詳細は、別紙の「マルチスレッドプログラミングの基礎」を参照)

なお、このサンプルプログラムでは、前述の「共通アロケータインターフェース」を使用する。以下のサンプルコードには再掲載していないので注意。また、共通アロケータインターフェースのサンプルと同様に、処理の状態を確認するために、グローバル new / delete 演算子の処理内にはプリント文を仕込んでいる。

サンプルプログラムのコンパイル上の注意点がある。

多態アロケータクラス内でグローバル new / delete 演算子をフレンド宣言しているが、コンパイラによってはこのコンパイルが通らないので、切り替えスイッチ用のマクロを用意している。(Visual C++ 2013 では大丈夫で、GCC 4.8.2 ではダメだった)

グローバル多態アロケータのサンプル：

【コンパイラスイッチ】

```
//--------------------
//コンパイラスイッチ
//#define IS_NOT_FRIEND_WITH_NEW_OPERATOR//オペレータ new/delete をフレンド宣言しない時はこのマクロを有効にする
```

【多態アロケータクラス定義】

```
//--------------------
//クラス宣言
class CTempPolyAllocatorBase;//一時多態アロケータ基底クラス

//--------------------
//多態アロケータ
class CPolyAllocator
{
#ifndef IS_NOT_FRIEND_WITH_NEW_OPERATOR
     //標準 new / deleete 演算子をフレンド化
     //※この CPolyAllocator クラスを直接インスタンス化するのは new / delete 演算子のみ
     friend void* operator new(const std::size_t size) throw();
     friend void* operator new[](const std::size_t size) throw();
     friend void operator delete(void* p) throw();
     friend void operator delete[](void* p) throw();
#endif//IS_NOT_FRIEND_WITH_NEW_OPERATOR
     friend class CTempPolyAllocatorBase;//一時多態アロケータ基底クラス
public:
     //オペレータ
     IAllocator* operator->(){ return m_allocator; }//アロー演算子（プロキシー）
public:
```

```
	//アクセッサ
	//アロケータ取得
	static IAllocator* getAllocator()
	{
		return m_allocator;
	}
	//アロケータ変更
	//※戻りとして変更前のアロケータを返す
	//※なるべく直接使用禁止⇒CTempPoly***Allocator クラスを使い、コンストラクタで自動的に元に戻す
	static IAllocator* setAllocator(IAllocator& new_allocator)
	{
		IAllocator* prev_allocator = m_allocator;//変更前のアロケータ取得
		m_allocator = &new_allocator;//アロケータ変更
		return prev_allocator;//変更前のアロケータを返す
	}
	//メモリ確保情報取得
	static const ALLOC_INFO* getAllocInfo()
	{
		return m_allocInfo;
	}
	//メモリ確保情報取得と同時に情報を破棄
	static const ALLOC_INFO* getAllocInfoWithReset()
	{
		const ALLOC_INFO* info = m_allocInfo;
		m_allocInfo = nullptr;
		return info;
	}
	//メモリ確保情報をセット
	static void setAllocInfo(const ALLOC_INFO* info)
	{
		m_allocInfo = info;
	}
#ifdef IS_NOT_FRIEND_WITH_NEW_OPERATOR
public:
#else//IS_NOT_FRIEND_WITH_NEW_OPERATOR
private://直接インスタンス生成不可（フレンド専用）
#endif//IS_NOT_FRIEND_WITH_NEW_OPERATOR
	//デフォルトコンストラクタ
	CPolyAllocator()
	{
		if (!m_allocator)//まだ何のアロケータも保持していない場合標準アロケータを自動的に保持する
			m_allocator = new CStdAllocAdp();//標準アロケータを明示的に初期化（クラス内 new を使用）
	}
public:
	//デストラクタ
	~CPolyAllocator()
	{}
protected:
	//フィールド
	static thread_local IAllocator* m_allocator;//現在のアロケータ
	static thread_local const ALLOC_INFO* m_allocInfo;//現在のメモリ確保情報
	//TLS を利用し、アロケータの変更が他のスレッドに影響しないようにする
};
```

【多態アロケータの静的変数をインスタンス化】

```
//--------------------
//多態アロケータの静的変数インスタンス化
thread_local IAllocator* CPolyAllocator::m_allocator = nullptr;//現在のアロケータ
thread_local const ALLOC_INFO* CPolyAllocator::m_allocInfo = nullptr;//現在のメモリ確保情報
```

【グローバル多態アロケータ（標準 new / delete 演算子の置き換え）】

```
//--------------------
//グローバル多態アロケータ
//※標準 new / delete 演算子の置き換え
//new
```

```
void* operator new(const std::size_t size) throw()
{
	CPolyAllocator allocator;
	printf("new(size=%d, poly_allocator=¥"%s¥":%d/%d/%d)¥n", size, allocator->getName(), allocator->getTotal(),
		allocator->getUsed(), allocator->getRemain());
	void* p = allocator->alloc(size);
	printf("  p=0x%p¥n", p);
	return p;
}
//配列版
void* operator new[](const std::size_t size) throw()
{
	CPolyAllocator allocator;
	printf("new[](size=%d, poly_allocator=¥"%s¥":%d/%d/%d)¥n", size, allocator->getName(), allocator->getTotal(),
		allocator->getUsed(), allocator->getRemain());
	void* p = allocator->alloc(size);
	printf("  p=0x%p¥n", p);
	return p;
}
//delete
void operator delete(void* p) throw()
{
	CPolyAllocator allocator;
	printf("delete(p=0x%p, poly_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator->getName(), allocator->getTotal(),
		allocator->getUsed(), allocator->getRemain());
	allocator->free(p);
}
//配列版
void operator delete[](void* p) throw()
{
	CPolyAllocator allocator;
	printf("delete[](p=0x%p, poly_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator->getName(), allocator->getTotal(),
		allocator->getUsed(), allocator->getRemain());
	allocator->free(p);
}
```

【一時多態アロケータクラス説明】

```
//--------------------
//一時多態アロケータ
//※多態アロケータの「現在のアロケータ」を一時的に変更するためのクラス
//※処理ブロックを抜ける時に、デストラクタで自動的に元の状態に戻す
```

【一時多態アロケータ基底クラス定義】

```
//--------------------
//一時多態アロケータ基底クラス
//※継承専用クラス
class CTempPolyAllocatorBase : public CPolyAllocator
{
protected://直接インスタンス生成不可（継承専用）
	//デフォルトコンストラクタ
	CTempPolyAllocatorBase() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CTempPolyAllocatorBase(IAllocator& allocator) :
		CPolyAllocator()
	{
		m_prevAllocator = setAllocator(allocator);//アロケータを変更して、変更前のアロケータを記憶
	}
public:
	//デストラクタ
	~CTempPolyAllocatorBase()
	{
		if (m_prevAllocator)
		{
			m_allocator = m_prevAllocator;//変更前のアロケータに戻す
			m_prevAllocator = nullptr;
```

```
            }
        }
private:
        //フィールド
        IAllocator* m_prevAllocator;//変更前のアロケータ
};
```

【一時多態アロケータテンプレートクラス（アダプター保持タイプ）定義】※主に継承専用だが、任意に使ってもいい

```
//--------------------
//一時多態アロケータテンプレートクラス：アダプター保持タイプ
template<class ADAPTER>
class CTempPolyAllocatorWithAdp : public CTempPolyAllocatorBase
{
public:
        //型
        typedef ADAPTER ADAPTER_TYPE;//アダプター型
        typedef typename ADAPTER::ALLOCATOR_TYPE ALLOCATOR_TYPE;//アロケータ型
public:
        //デフォルトコンストラクタ
        CTempPolyAllocatorWithAdp() :
                m_adapter(),
                CTempPolyAllocatorBase(m_adapter)
        {}
        //コンストラクタ
        CTempPolyAllocatorWithAdp(ALLOCATOR_TYPE& allocator) :
                m_adapter(allocator),
                CTempPolyAllocatorBase(m_adapter)
        {}
        //デストラクタ
        ~CTempPolyAllocatorWithAdp()
        {}
protected:
        //フィールド
        ADAPTER_TYPE m_adapter;//アロケータアダプター
};
```

【一時多態アロケータテンプレートクラス（アダプター直接利用タイプ）定義】※主に継承専用だが、任意に使ってもいい

```
//--------------------
//一時多態アロケータテンプレートクラス：アダプター直接利用タイプ
//※アダプターを保持せず外部から受け渡したものをそのまま利用する
template<class ADAPTER>
class CTempPolyAllocatorDirect : public CTempPolyAllocatorBase
{
public:
        //型
        typedef ADAPTER ADAPTER_TYPE;//アダプター型
public:
        //デフォルトコンストラクタ
        CTempPolyAllocatorDirect() = delete;//コンストラクタ引数必須
        //コンストラクタ
        CTempPolyAllocatorDirect(ADAPTER_TYPE& adapter) :
                CTempPolyAllocatorBase(adapter)
        {}
        //デストラクタ
        ~CTempPolyAllocatorDirect()
        {}
};
```

【一時多態アロケータテンプレートクラス（スタックアロケータアダプター保持タイプ）定義】※主に継承専用だが、任意に使ってもいい

```
//--------------------
//一時多態アロケータテンプレートクラス：スタックアロケータアダプター保持タイプ
template<class ADAPTER>
class CTempPolyAllocatorWithStackAdp : public CTempPolyAllocatorBase
{
public:
```

```
	//型
	typedef ADAPTER ADAPTER_TYPE;//アダプター型
	typedef typename ADAPTER::ALLOCATOR_TYPE ALLOCATOR_TYPE;//アロケータ型
public:
	//デフォルトコンストラクタ
	CTempPolyAllocatorWithStackAdp() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CTempPolyAllocatorWithStackAdp(ALLOCATOR_TYPE& allocator, const CIStackAllocAdp::E_AUTO_REWIND auto_rewind) :
		m_adapter(allocator, auto_rewind),
		CTempPolyAllocatorBase(m_adapter)
	{}
	CTempPolyAllocatorWithStackAdp(ALLOCATOR_TYPE& allocator, const IStackAllocator::E_ORDERED ordered,
		const CIStackAllocAdp::E_AUTO_REWIND auto_rewind) :
		m_adapter(allocator, ordered, auto_rewind),
		CTempPolyAllocatorBase(m_adapter)
	{}
	CTempPolyAllocatorWithStackAdp(ALLOCATOR_TYPE& allocator, const IStackAllocator::E_ORDERED ordered) :
		m_adapter(allocator, ordered),
		CTempPolyAllocatorBase(m_adapter)
	{}
	//デストラクタ
	~CTempPolyAllocatorWithStackAdp()
	{}
protected:
	//フィールド
	ADAPTER_TYPE m_adapter;//アロケータアダプター
};
```

【一時多態アロケータクラス（標準アロケータ用）定義】

```
//--------------------
//一時多態アロケータクラス：標準アロケータ用
using CTempPolyStdAllocator = CTempPolyAllocatorWithAdp<CStdAllocAdp>;//C++11 形式
```

【一時多態アロケータクラス（単一バッファアロケータ用）定義】

```
//--------------------
//一時多態アロケータクラス：単一バッファアロケータ用
using CTempPolyMonoAllocator = CTempPolyAllocatorDirect<CMonoAllocAdp>;//C++11 形式
```

【一時多態アロケータクラス（スタックアロケータインターフェース用）定義】

```
//--------------------
//一時多態アロケータクラス：スタックアロケータインターフェース用
class CTempPolyIStackAllocator : public CTempPolyAllocatorWithStackAdp<CIStackAllocAdp>
{
public:
	//デフォルトコンストラクタ
	CTempPolyIStackAllocator() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CTempPolyIStackAllocator(IStackAllocator& allocator,
		const CIStackAllocAdp::E_AUTO_REWIND auto_rewind = CIStackAllocAdp::NOREWIND) :
		CTempPolyAllocatorWithStackAdp<CIStackAllocAdp>(allocator, auto_rewind)
	{}
	//デストラクタ
	~CTempPolyIStackAllocator()
	{}
};
```

【一時多態アロケータクラス（スタックアロケータ用）定義】

```
//--------------------
//一時多態アロケータクラス：スタックアロケータ用
class CTempPolyStackAllocator : public CTempPolyAllocatorWithStackAdp<CStackAllocAdp>
{
public:
	//デフォルトコンストラクタ
	CTempPolyStackAllocator() = delete;//コンストラクタ引数必須
	//コンストラクタ
```

```
	CTempPolyStackAllocator(CStackAllocator& allocator,
				const CIStackAllocAdp::E_AUTO_REWIND auto_rewind = CIStackAllocAdp::NOREWIND) :
		CTempPolyAllocatorWithStackAdp<CStackAllocAdp>(allocator, auto_rewind)
	{}
	//デストラクタ
	~CTempPolyStackAllocator()
	{}
};
```

【一時多態アロケータクラス（双方向スタックアロケータ用）定義】

```
//--------------------
//一時多態アロケータクラス：双方向スタックアロケータ用
class CTempPolyDualStackAllocator : public CTempPolyAllocatorWithStackAdp<CDualStackAllocAdp>
{
public:
	//デフォルトコンストラクタ
	CTempPolyDualStackAllocator() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CTempPolyDualStackAllocator(CDualStackAllocator& allocator,
				const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT,
				const CIStackAllocAdp::E_AUTO_REWIND auto_rewind = CIStackAllocAdp::NOREWIND) :
		CTempPolyAllocatorWithStackAdp<CDualStackAllocAdp>(allocator, ordered, auto_rewind)
	{}
	//デストラクタ
	~CTempPolyDualStackAllocator()
	{}
};
```

【一時多態アロケータクラス（プールアロケータ用）定義】

```
//--------------------
//一時多態アロケータクラス：プールアロケータ用
using CTempPolyPoolAllocator = CTempPolyAllocatorWithAdp<CPoolAllocAdp>;//C++11 形式
```

【テスト用クラス定義】

```
//--------------------
//テスト用クラス
class CTest7
{
public:
	//デフォルトコンストラクタ
	CTest7() :
		CTest7("(default)")//他のコンストラクタ呼び出し（C++11 で追加された仕様）
		//m_name("(default)")//C++11 が使えない場合は普通に初期化
	{
		printf("CTest7::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest7(const char* name) :
		m_name(name)
	{
		printf("CTest7::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	virtual ~CTest7()
	{
		printf("CTest7::Destructor : name=¥"%s¥"¥n", m_name);
	}
protected:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【多重継承テスト用クラス定義】

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

```
//テスト用クラス（多重継承テスト用）
class CTest7ex
{
public:
	//コンストラクタ
	CTest7ex()
	{
		printf("CTest7ex::Constructor¥n");
	}
	//デストラクタ
	virtual ~CTest7ex()
	{
		printf("CTest7ex::Destructor¥n");
	}
protected:
	//フィールド
	int m_dummy;//ダミー
};
//--------------------
//テスト用クラス（多重継承）
class CDerivedTest7 : public CTest7, public CTest7ex
{
public:
	//デフォルトコンストラクタ
	CDerivedTest7() :
		CTest7(),
		CTest7ex()
	{
		printf("CDerivedTest7::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CDerivedTest7(const char* name) :
		CTest7(name),
		CTest7ex()
	{
		printf("CDerivedTest7::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	~CDerivedTest7() override
	{
		printf("CDerivedTest7::Destructor : name=¥"%s¥"¥n", m_name);
	}
private:
	//フィールド
	int m_dummy;//ダミー
};
```

【テスト（共通関数）】

```
//--------------------
//テスト（共通関数）
//※先の共通アロケータインターフェースのテストと異なり、アダプターを受け取らず、そのまま new / delete する
//※呼び出し元で、new / delete の実装を切り替えている
//クラス単体のテスト
void test7_sub1(const char* name)
{
	CTest7* obj_p = new CTest7(name);//配置 new ではなく、普通の new 演算子
	printf("obj_p=0x%p¥n", obj_p);
	delete obj_p;//配置 delete ではなく、普通の delete 演算子
}
//クラスの配列のテスト
void test7_sub2()
{
	CTest7* obj_p = new CTest7[3];//配置 new ではなく、普通の new[]演算子
	printf("obj_p=0x%p¥n", obj_p);
	delete[] obj_p;//配置 delete ではなく、普通の delete[]演算子
```

```
}
//多重継承クラスのテスト
void test7_sub3(const char* name)
{
	CTest7ex* obj_p = new CDerivedTest7(name);//配置 new ではなく、普通の new 演算子
	printf("obj_p=0x%p¥n", obj_p);
	delete obj_p;//配置 delete ではなく、普通の delete 演算子
}
```

【テスト】

```
//--------------------
//テスト
void test7()
{
	//標準アロケータ (malloc / free) 使用
	{
		printf("----------CStdAllocAdp¥n");
		CTempPolyStdAllocator poly_allocator;
		test7_sub1("テスト 7-1a");
		test7_sub2();//クラスの配列でずれたポインタは、delet[]演算子によって正しく処理されるため、問題が起こらない
		test7_sub3("テスト 7-1b");//多重継承でずれたポインタは、delet 演算子によって正しく処理されるため、
					//問題が起こらない
	}
	//単一バッファアロケータ使用
	{
		printf("----------CMonoAllocAdp¥n");
		CMonoAllocAdpWithBuff<128> allocator_adp;
		CTempPolyMonoAllocator poly_allocator(allocator_adp);
		test7_sub1("テスト 7-2a");
		test7_sub2();
		test7_sub3("テスト 7-2b");
	}
	//スタックアロケータ使用
	{
		printf("----------CStackAllocAdp¥n");
		CStackAllocatorWithBuff<128> allocator;
		CTempPolyStackAllocator poly_allocator(allocator);
		test7_sub1("テスト 7-3a");
		{
			//自動巻き戻しのテスト
			CTempPolyStackAllocator poly_allocator_tmp(allocator, CIStackAllocAdp::AUTO_REWIND);
			test7_sub2();
		}
		test7_sub3("テスト 7-3b");
	}
	//双方向スタックアロケータ使用
	{
		printf("----------CDualStackAllocAdp¥n");
		CDualStackAllocatorWithBuff<128> allocator;
		CTempPolyDualStackAllocator poly_allocator(allocator, DSA_REVERSE);
		test7_sub1("テスト 7-4a");
		{
			//自動巻き戻しのテスト
			CTempPolyDualStackAllocator poly_allocator_tmp(allocator, DSA_REVERSE,
									CIStackAllocAdp::AUTO_REWIND);
			test7_sub2();
		}
		test7_sub3("テスト 7-4b");
	}
	//スタックアロケータをスタックアロケータインターフェースとして使用
	{
		printf("----------CIStackAllocAdp on CStackAllocAdp¥n");
		CStackAllocatorWithBuff<128> allocator;
		CTempPolyIStackAllocator poly_allocator(allocator);
		test7_sub1("テスト 7-5a");
```

```
            {
                  //自動巻き戻しのテスト
                  CTempPolyIStackAllocator poly_allocator_tmp(allocator, CIStackAllocAdp::AUTO_REWIND);
                  test7_sub2();
            }
            test7_sub3("テスト 7-5b");
      }
      //双方向スタックアロケータをスタックアロケータインターフェースとして使用
      {
            printf("----------CIStackAllocAdp on CDualStackAllocAdp¥n");
            CDualStackAllocatorWithBuff<128> allocator;
            CTempPolyIStackAllocator poly_allocator(allocator);
            test7_sub1("テスト 7-6a");
            {
                  //自動巻き戻しのテスト
                  CTempPolyIStackAllocator poly_allocator_tmp(allocator, CIStackAllocAdp::AUTO_REWIND);
                  test7_sub2();
            }
            test7_sub3("テスト 7-6b");
      }
      //プールアロケータ使用
      {
            printf("----------CPoolAllocAdp¥n");
            CPoolAllocatorWithBuff<24, 5> allocator;
            CTempPolyPoolAllocator poly_allocator(allocator);
            test7_sub1("テスト 7-7a");
            test7_sub2();
            test7_sub3("テスト 7-7b");
      }
}
```

↓（実行結果）

```
----------CStdAllocAdp
new(size=12, poly_allocator="CStdAllocAdp":0/0/0)          ←アロケータ名：メモリ総量/使用量/残量を表示
  p=0x0089B8B8                                                （以降、赤字で示す）
CTest7::Constructor : name="テスト 7-1a"                     ※標準アロケータ（malloc）は総量/使用量/残量不明
obj_p=0x0089B8B8
CTest7::Destructor : name="テスト 7-1a"                    ←delete 演算子により、デストラクタが自動的に
delete(p=0x0089B8B8, poly_allocator="CStdAllocAdp":0/0/0)    実行されている
new[](size=40, poly_allocator="CStdAllocAdp":0/0/0)        ←new / delete 演算子により、配列も問題なく処理可能
  p=0x0089B8B8
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x0089B8BC                                           ←配列確保によりポインタがずれている
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x0089B8B8, poly_allocator="CStdAllocAdp":0/0/0) ←delete 時には正しいポインタに戻っている
new(size=24, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x0089B8B8
CTest7::Constructor : name="テスト 7-1b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-1b"
obj_p=0x0089B8C4                                           ←多重継承のアップキャストによりポインタがずれている
CDerivedTest7::Destructor : name="テスト 7-1b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-1b"
delete(p=0x0089B8B8, poly_allocator="CStdAllocAdp":0/0/0)  ←delete 時には正しいポインタに戻っている
----------CMonoAllocAdp
new(size=12, poly_allocator="CMonoAllocAdp":128/0/128)     ←単一バッファアロケータが機能している
  p=0x005DF778
```

```
CTest7::Constructor : name="テスト 7-2a"
obj_p=0x005DF778
CTest7::Destructor : name="テスト 7-2a"
delete(p=0x005DF778, poly_allocator="CMonoAllocAdp":128/128/0)     ←メモリ確保の要求サイズによらず、
                                                                    バッファ全域を消費している
new[](size=40, poly_allocator="CMonoAllocAdp":128/0/128)     ←メモリが解放されて残量が増えている
  p=0x005DF778
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x005DF77C
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x005DF778, poly_allocator="CMonoAllocAdp":128/128/0)
new(size=24, poly_allocator="CMonoAllocAdp":128/0/128)
  p=0x005DF778
CTest7::Constructor : name="テスト 7-2b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-2b"
obj_p=0x005DF784
CDerivedTest7::Destructor : name="テスト 7-2b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-2b"
delete(p=0x005DF778, poly_allocator="CMonoAllocAdp":128/128/0)
----------CStackAllocAdp
new(size=12, poly_allocator="CStackAllocAdp":128/0/128)      ←スタックアロケータが機能している
  p=0x005DF6D4
CTest7::Constructor : name="テスト 7-3a"
obj_p=0x005DF6D4
CTest7::Destructor : name="テスト 7-3a"
delete(p=0x005DF6D4, poly_allocator="CStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CStackAllocAdp":128/12/116)   ←スタックアロケータのメモリは解放されない
  p=0x005DF6E0                                                 （以後のスタック系処理も同様）
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x005DF6E4
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x005DF6E0, poly_allocator="CStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CStackAllocAdp":128/12/116)     ←自動巻き戻し機能により、スタックの残量が
  p=0x005DF6E0                                                 配列のメモリ確保テスト前の状態に戻っている
CTest7::Constructor : name="テスト 7-3b"                       （以後のスタック系処理も同様）
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-3b"
obj_p=0x005DF6EC
CDerivedTest7::Destructor : name="テスト 7-3b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-3b"
delete(p=0x005DF6E0, poly_allocator="CStackAllocAdp":128/36/92)
----------CDualStackAllocAdp
new(size=12, poly_allocator="CDualStackAllocAdp":128/0/128) ←双方向スタックアロケータが機能している
  p=0x005DF678
CTest7::Constructor : name="テスト 7-4a"
obj_p=0x005DF678
CTest7::Destructor : name="テスト 7-4a"
delete(p=0x005DF678, poly_allocator="CDualStackAllocAdp":128/12/116)
```

```
new[](size=40, poly_allocator="CDualStackAllocAdp":128/12/116)
  p=0x005DF650
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x005DF654
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x005DF650, poly_allocator="CDualStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CDualStackAllocAdp":128/12/116)
  p=0x005DF660
CTest7::Constructor : name="テスト 7-4b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-4b"
obj_p=0x005DF66C
CDerivedTest7::Destructor : name="テスト 7-4b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-4b"
delete(p=0x005DF660, poly_allocator="CDualStackAllocAdp":128/36/92)
----------CIStackAllocAdp on CStackAllocAdp
new(size=12, poly_allocator="CIStackAllocAdp":128/0/128)     ←スタックアロケータインターフェース（実体は
  p=0x005DF51C                                                 スタックアロケータ）が機能している
CTest7::Constructor : name="テスト 7-5a"
obj_p=0x005DF51C
CTest7::Destructor : name="テスト 7-5a"
delete(p=0x005DF51C, poly_allocator="CIStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CIStackAllocAdp":128/12/116)
  p=0x005DF528
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x005DF52C
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x005DF528, poly_allocator="CIStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CIStackAllocAdp":128/12/116)
  p=0x005DF528
CTest7::Constructor : name="テスト 7-5b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-5b"
obj_p=0x005DF534
CDerivedTest7::Destructor : name="テスト 7-5b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-5b"
delete(p=0x005DF528, poly_allocator="CIStackAllocAdp":128/36/92)
----------CIStackAllocAdp on CDualStackAllocAdp
new(size=12, poly_allocator="CIStackAllocAdp":128/0/128)     ←スタックアロケータインターフェース（実体は
  p=0x005DF44C                                                 双方向スタックアロケータ）が機能している
CTest7::Constructor : name="テスト 7-6a"
obj_p=0x005DF44C
CTest7::Destructor : name="テスト 7-6a"
delete(p=0x005DF44C, poly_allocator="CIStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CIStackAllocAdp":128/12/116)
  p=0x005DF458
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
```

```
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x005DF45C
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x005DF458, poly_allocator="CIStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CIStackAllocAdp":128/12/116)
  p=0x005DF458
CTest7::Constructor : name="テスト 7-6b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-6b"
obj_p=0x005DF464
CDerivedTest7::Destructor : name="テスト 7-6b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-6b"
delete(p=0x005DF458, poly_allocator="CIStackAllocAdp":128/36/92)
----------CPoolAllocAdp
new(size=12, poly_allocator="CPoolAllocAdp":120/0/120)       ←プールアロケータが機能している
  p=0x005DF37C
CTest7::Constructor : name="テスト 7-7a"
obj_p=0x005DF37C
CTest7::Destructor : name="テスト 7-7a"
delete(p=0x005DF37C, poly_allocator="CPoolAllocAdp":120/24/96)    ←メモリ確保の要求サイズによらず、
                                                                    ブロックサイズ分のメモリを消費している
new[](size=40, poly_allocator="CPoolAllocAdp":120/0/120)     ←メモリが解放されて残量が増えている
  p=0x00000000                                               ←ブロックサイズを超えるメモリ確保要求につき、
obj_p=0x00000000                                               残量が十分あるのにメモリ確保失敗
new(size=24, poly_allocator="CPoolAllocAdp":120/0/120)         ※配列のメモリ確保は一塊のメモリであることに注意
  p=0x005DF394
CTest7::Constructor : name="テスト 7-7b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-7b"
obj_p=0x005DF3A0
CDerivedTest7::Destructor : name="テスト 7-7b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-7b"
delete(p=0x005DF394, poly_allocator="CPoolAllocAdp":120/24/96)    ←ブロックサイズぴったりのメモリ確保は OK
```

▼ メモリ確保情報（デバッグ情報）

メモリ確保時に、アラインメントの指定や関数名などの情報が受け渡せると便利である。

メモリ管理システム（メモリマネージャ）を作成した場合、メモリリークなどの問題追及のために、メモリ管理情報の一部としてこうしたデバッグ情報を記録することもある。

▼ メモリ確保情報受け渡しのサンプル

ここまでのアプローチを踏まえ、配置 new / 配置 delete ではなく、グローバル new / delete を使った上で、メモリ確保情報を受け渡すサンプルを示す。

関数名や行番号を得るために、どうしてもマクロが必須であることから、残念ながら、やや変則的で可読性を損ねたコードになっている。

C++11 の可変長テンプレート引数と TLS で実現する仕組みを示す。

それが使えない環境では配置 new の活用が不可欠となる。基本的に問題があるのは配置 delete だけなので、配置 new と delete 演算子の組み合わせに処理を変えても良い。ただし、その場合でも、配置 new と一対の配置 delete は用意する必要がある。

なお、このサンプルプログラムでは、前述の「共通アロケータインターフェース」~「グローバル多態アロケータ」を使用し、その一部を置き換えたものとして説明する。以下のサンプルコードには同じコードを再掲載していないので注意。

メモリ確保情報の受け渡しサンプル：

【メモリ確保情報構造体定義】

```
//--------------------
//メモリ確保情報
//※デバッグ情報（参照のみ）
struct ALLOC_INFO
{
	static const std::size_t DEFAULT_ALIGN = sizeof(int);//デフォルトのアラインメントサイズ
	const char* m_fileName;//ファイル名
	const char* m_funcName;//関数名
	const char* m_callPointName;//コールポイント名
	float m_time;//ゲーム時間
	const char* m_typeName;//型名
	std::size_t m_align;//アラインメントサイズ
	ALLOC_INFO(const char* file_name, const char* func_name, const char* call_point_name, const float time,
	                                                const char* type_name, const std::size_t align = DEFAULT_ALIGN) :
		m_fileName(file_name),
		m_funcName(func_name),
		m_callPointName(call_point_name),
		m_time(time),
		m_typeName(type_name),
		m_align(align)
	{}
};
```

【テスト（共通関数）】※「グローバル多態アロケータ」のサンプルプログラムから変更(NEW,DELETE マクロについては後述)

```
//--------------------
//テスト（共通関数）
//※先の共通アロケータインターフェースのテストと異なり、アダプターを受け取らず、そのまま new / delete する
//※呼び出し元で、new / delete の実装を切り替えている
//クラス単体のテスト
void test7_sub1(const char* name)
{
	CTest7* obj_p = NEW(CTest7, name);
	printf("obj_p=0x%p¥n", obj_p);
	DELETE obj_p;
}
//クラスの配列のテスト
void test7_sub2()
{
	CTest7* obj_p = NEWARR(CTest7, 3);
	printf("obj_p=0x%p¥n", obj_p);
	DELETEARR obj_p;
}
//多重継承クラスのテスト
void test7_sub3(const char* name)
{
	CTest7ex* obj_p = NEWALIGN(CDerivedTest7, 16, name);//アップキャスト
	printf("obj_p=0x%p¥n", obj_p);
	DELETE obj_p;
}
```

【グローバル多態アロケータ（標準 new / delete 演算子の置き換え）】※「グローバル多態アロケータ」のサンプルプログラムから変更

```
//---------------------
//共通関数参照
extern const char* getFileName(const char* str);//関数参照：ファイル名取得関数（ディレクトリ部を除いた文字列を返す）
//---------------------
//グローバル多態アロケータ
//※標準 new / delete 演算子の置き換え
//new
void* operator new(const std::size_t size) throw()
{
	CPolyAllocator allocator;
	printf("new(size=%d, poly_allocator=¥"%s¥":%d/%d/%d)¥n", size, allocator->getName(), allocator->getTotal(),
                                                                allocator->getUsed(), allocator->getRemain());
	std::size_t align = ALLOC_INFO::DEFAULT_ALIGN;//アラインメントの処理も追加
	const ALLOC_INFO* info = CPolyAllocator::getAllocInfoWithReset();//メモリ確保情報取得
	if (info)
	{
		align = info->m_align;
		printf("  fileName=¥"%s¥", funcName=¥"%s¥", callPointName=¥"%s¥", time=%.3f, typename=¥"%s¥", align=%d¥n",
				getFileName(info->m_fileName), info->m_funcName, info->m_callPointName, info->m_time,
                                                                info->m_typeName, info->m_align);
	}
	void* p = allocator->alloc(size, align);
	printf("  p=0x%p¥n", p);
	return p;
}
//配列版
void* operator new[](const std::size_t size) throw()
{
	CPolyAllocator allocator;
	printf("new[](size=%d, poly_allocator=¥"%s¥":%d/%d/%d)¥n", size, allocator->getName(), allocator->getTotal(),
                                                                allocator->getUsed(), allocator->getRemain());
	std::size_t align = ALLOC_INFO::DEFAULT_ALIGN;//アラインメントの処理も追加
	const ALLOC_INFO* info = CPolyAllocator::getAllocInfoWithReset();//メモリ確保情報取得
	if (info)
	{
		align = info->m_align;
		printf("  fileName=¥"%s¥", funcName=¥"%s¥", callPointName=¥"%s¥", time=%.3f, typename=¥"%s¥", align=%d¥n",
				getFileName(info->m_fileName), info->m_funcName, info->m_callPointName, info->m_time,
                                                                info->m_typeName, info->m_align);
	}
	void* p = allocator->alloc(size, align);
	printf("  p=0x%p¥n", p);
	return p;
}
//delete
void operator delete(void* p) throw()
{
	CPolyAllocator allocator;
	printf("delete(p=0x%p, poly_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator->getName(), allocator->getTotal(),
                                                                allocator->getUsed(), allocator->getRemain());
	allocator->free(p);
}
//配列版
void operator delete[](void* p) throw()
{
	CPolyAllocator allocator;
	printf("delete[](p=0x%p, poly_allocator=¥"%s¥":%d/%d/%d)¥n", p, allocator->getName(), allocator->getTotal(),
                                                                allocator->getUsed(), allocator->getRemain());
	allocator->free(p);
}
```

【メモリ確保情報付き new 関数】※直接使用せず、NEW マクロでラップして使用する

```
//---------------------
//デバッグ情報収集用関数参照
```

```
extern const char* getCurrentCallPointNameDummy();//コールポイント名取得（ダミー）
extern float getGameTimeDummy();//ゲーム時間取得（ダミー）
//--------------------
//メモリ確保情報付き NEW 処理
template<class T, typename... Tx>
T* newWithInfo(const char* file_name, const char* func_name, const std::size_t align, Tx ...nx)
{
	const char* call_point_name = getCurrentCallPointNameDummy();//コールポイント名取得
	const float game_time = getGameTimeDummy();//ゲーム時間取得
	const ALLOC_INFO info(file_name, func_name, call_point_name, game_time, typeid(T).name(), align);
                                                                        //メモリ確保情報生成
	CPolyAllocator::setAllocInfo(&info);//メモリ確保情報受け渡し
                                        //※ローカル変数のポインタを受け渡すことになるが、
                                        //  後続の処理でのみ参照することと、TLS で保護されるため問題なし
	return new T(nx...);//メモリ確保
}
//配列版
//※【問題点】配列 new にアラインメントが指定された場合、正しいポインタが計算しきれないため、
//            アラインメント指定に対応しない
template<class T, std::size_t array_size>
T* newArrayWithInfo(const char* file_name, const char* func_name)
{
	const char* call_point_name = getCurrentCallPointNameDummy();//コールポイント名取得
	const float game_time = getGameTimeDummy();//ゲーム時間取得
	const ALLOC_INFO info(file_name, func_name, call_point_name, game_time, typeid(T[array_size]).name());
                                                                        //メモリ確保情報生成
	CPolyAllocator::setAllocInfo(&info);//メモリ確保情報受け渡し
                                        //※ローカル変数のポインタを受け渡すことになるが、
                                        //  後続の処理でのみ参照することと、TLS で保護されるため問題なし
	return new T[array_size];//メモリ確保
}
```

【NEW マクロの補助マクロ】

```
//--------------------
//NEW マクロ補助マクロ
#define TO_STRING(s) #s//__LINE__を文字列化するための二重マクロ
#define TO_STRING_EX(s) TO_STRING(s)//__LINE__を文字列化するためのマクロ
#define GET_CONCATENATED_FILE_NAME() __FILE__ "(" TO_STRING_EX(__LINE__) ")"  [" __TIMESTAMP__ "]"
                                                                        //ファイル名合成＆取得マクロ
#define GET_FUNC_NAME() __PRETTY_FUNCTION__//関数名取得マクロ
```

【NEW マクロ】

```
//--------------------
//NEW マクロ
//※メモリ確保情報付き
#define NEW(T, ...) newWithInfo<T>(GET_CONCATENATED_FILE_NAME(), GET_FUNC_NAME(), ALLOC_INFO::DEFAULT_ALIGN,
                                                                        __VA_ARGS__)//NEW
#define NEWALIGN(T, align, ...) newWithInfo<T>(GET_CONCATENATED_FILE_NAME(), GET_FUNC_NAME(), align, __VA_ARGS__)
                                                                        //アラインメント指定付き NEW
#define NEWARR(T, array_size) newArrayWithInfo<T, array_size>(GET_CONCATENATED_FILE_NAME(), GET_FUNC_NAME())//配列 NEW
//#define NEWALIGNARR(T, align, array_size)//アライメント指定付き配列 NEW は正確に計算できないので対応しない
```

【参考：メモリ確保情報を扱わない NEW マクロ】

```
//--------------------
//NEW マクロ
//※メモリ確保情報なし
#define NEW(T, ...) new T(__VA_ARGS__)//NEW
#define NEWALIGN(T, align, ...) new T(__VA_ARGS__)
                                        //アラインメント指定付き NEW（互換用）※実際にはアラインメントは機能しない
#define NEWARR(T, array_size) new T[array_size]//配列 NEW
//#define NEWALIGNARR(T, align, array_size)//アライメント指定付き配列 NEW は正確に計算できないので対応しない
```

【DELETE マクロ】

```
//--------------------
//DELETE マクロ
```

```
#define DELETE delete//DELETE
#define DELETEARR delete[]//配列 DELETE
```

【デバッグ情報収集用関数（ダミー）】

```
//--------------------
//デバッグ情報収集用関数
const char* getCurrentCallPointNameDummy(){ return "(unknown call-point)"; }//コールポイント名取得（ダミー）
float getGameTimeDummy(){ return 0.f; }//ゲーム時間取得（ダミー）
```

【共通関数】

```
#include <string.h>
//ファイル名取得関数（ディレクトリ部を除いた文字列を返す）
const char* getFileName(const char* str)
{
	std::size_t len = strlen(str);
	const char* p = str + len;
	for (; len > 0; --len, --p)
	{
		const char c = *(p - 1);
		if (c == '¥¥' || c == '/')
			return p;
	}
	return str;
}
```

【参考：メタ処理用共通関数：constexpr でコンパイル時に計算を済ませる】
※これが有効に使えるなら、GET_CONCATENATED_FILE_NAME() 内で使用して、コンパイル時に計算を済ませてもよい。
　その場合、operator new() / operator new[]() 内で使用している getFileName() は不要になる。

```
#include <string.h>
//【constexpr 版】ファイル名取得関数（ディレクトリ部を除いた文字列を返す）
//※再帰処理部（直接使用しない）
constexpr const char* getConstFileNameRecursive(const char* str, const std::size_t len)
{
	return len == 0 ?
	str :
		*(str + len - 1) == '¥¥' || *(str + len - 1) == '/' ?
			str + len :
			getConstFileNameRecursive(str, len - 1);
}
//【constexpr 版】ファイル名取得関数（ディレクトリ部を除いた文字列を返す）
constexpr const char* getConstFileName(const char* str)
{
	return getConstFileNameRecursive(str, strlen(str));
}
```

↓（実行結果）

```
----------CStdAllocAdp
new(size=12, poly_allocator="CStdAllocAdp":0/0/0)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", uncName="void __cdecltest7_sub1(const char *)",
                                callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
                                                                ←メモリ確保情報が正しく表示されている
                                                                  （以降、とくに赤字では示さない）
  p=0x012BB7F8
CTest7::Constructor : name="テスト7-1a"
obj_p=0x012BB7F8
CTest7::Destructor : name="テスト7-1a"
delete(p=0x012BB7F8, poly_allocator="CStdAllocAdp":0/0/0)
new[](size=40, poly_allocator="CStdAllocAdp":0/0/0)
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                              callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x012BB7F8
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
```

```
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x012BB7FC
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x012BB7F8, poly_allocator="CStdAllocAdp":0/0/0)
new(size=24, poly_allocator="CStdAllocAdp":0/0/0)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                    callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x012BB830                                          ←アラインメントの指定が正しく反映されている
CTest7::Constructor : name="テスト 7-1b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-1b"
obj_p=0x012BB83C
CDerivedTest7::Destructor : name="テスト 7-1b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-1b"
delete(p=0x012BB830, poly_allocator="CStdAllocAdp":0/0/0)
----------CMonoAllocAdp
new(size=12, poly_allocator="CMonoAllocAdp":128/0/128)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub1(const char *)",
                                  callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
  p=0x00ECFAA8
CTest7::Constructor : name="テスト 7-2a"
obj_p=0x00ECFAA8
CTest7::Destructor : name="テスト 7-2a"
delete(p=0x00ECFAA8, poly_allocator="CMonoAllocAdp":128/128/0)
new[](size=40, poly_allocator="CMonoAllocAdp":128/0/128)
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                           callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x00ECFAA8
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x00ECFAAC
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x00ECFAA8, poly_allocator="CMonoAllocAdp":128/128/0)
new(size=24, poly_allocator="CMonoAllocAdp":128/0/128)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                    callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x00ECFAA8              ←現状の単一バッファアロケータはアラインメント計算に対応しない（今後対応した方が便利）
CTest7::Constructor : name="テスト 7-2b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-2b"
obj_p=0x00ECFAB4
CDerivedTest7::Destructor : name="テスト 7-2b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-2b"
delete(p=0x00ECFAA8, poly_allocator="CMonoAllocAdp":128/128/0)
----------CStackAllocAdp
new(size=12, poly_allocator="CStackAllocAdp":128/0/128)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub1(const char *)",
                                  callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
  p=0x00ECFA04
CTest7::Constructor : name="テスト 7-3a"
obj_p=0x00ECFA04
CTest7::Destructor : name="テスト 7-3a"
delete(p=0x00ECFA04, poly_allocator="CStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CStackAllocAdp":128/12/116)
```

```
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                            callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x00ECFA10
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x00ECFA14
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x00ECFA10, poly_allocator="CStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CStackAllocAdp":128/12/116)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                      callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x00ECFA10                ←アラインメントの指定が正しく反映されている
CTest7::Constructor : name="テスト 7-3b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-3b"
obj_p=0x00ECFA1C
CDerivedTest7::Destructor : name="テスト 7-3b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-3b"
delete(p=0x00ECFA10, poly_allocator="CStackAllocAdp":128/36/92)
----------CDualStackAllocAdp
new(size=12, poly_allocator="CDualStackAllocAdp":128/0/128)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub1(const char *)",
                                  callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
  p=0x00ECF9A8
CTest7::Constructor : name="テスト 7-4a"
obj_p=0x00ECF9A8
CTest7::Destructor : name="テスト 7-4a"
delete(p=0x00ECF9A8, poly_allocator="CDualStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CDualStackAllocAdp":128/12/116)
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                            callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x00ECF980
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x00ECF984
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x00ECF980, poly_allocator="CDualStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CDualStackAllocAdp":128/12/116)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                      callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x00ECF990                ←アラインメントの指定が正しく反映されている
CTest7::Constructor : name="テスト 7-4b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-4b"
obj_p=0x00ECF99C
CDerivedTest7::Destructor : name="テスト 7-4b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-4b"
delete(p=0x00ECF990, poly_allocator="CDualStackAllocAdp":128/36/92)
----------CIStackAllocAdp on CStackAllocAdp
new(size=12, poly_allocator="CIStackAllocAdp":128/0/128)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub1(const char *)",
```

```
                                callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
  p=0x00ECF84C
CTest7::Constructor : name="テスト 7-5a"
obj_p=0x00ECF84C
CTest7::Destructor : name="テスト 7-5a"
delete(p=0x00ECF84C, poly_allocator="CIStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CIStackAllocAdp":128/12/116)
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                            callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x00ECF858
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x00ECF85C
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x00ECF858, poly_allocator="CIStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CIStackAllocAdp":128/12/116)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                        callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x00ECF860                ←アラインメントの指定が正しく反映されている
CTest7::Constructor : name="テスト 7-5b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-5b"
obj_p=0x00ECF86C
CDerivedTest7::Destructor : name="テスト 7-5b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト 7-5b"
delete(p=0x00ECF860, poly_allocator="CIStackAllocAdp":128/44/84)
----------CIStackAllocAdp on CDualStackAllocAdp
new(size=12, poly_allocator="CIStackAllocAdp":128/0/128)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub1(const char *)",
                                callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
  p=0x00ECF77C
CTest7::Constructor : name="テスト 7-6a"
obj_p=0x00ECF77C
CTest7::Destructor : name="テスト 7-6a"
delete(p=0x00ECF77C, poly_allocator="CIStackAllocAdp":128/12/116)
new[](size=40, poly_allocator="CIStackAllocAdp":128/12/116)
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                            callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x00ECF788
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
CTest7::Constructor : name="(default)"
CTest7::DefaultConstructor : name="(default)"
obj_p=0x00ECF78C
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
CTest7::Destructor : name="(default)"
delete[](p=0x00ECF788, poly_allocator="CIStackAllocAdp":128/52/76)
new(size=24, poly_allocator="CIStackAllocAdp":128/12/116)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                        callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x00ECF790                ←アラインメントの指定が正しく反映されている
CTest7::Constructor : name="テスト 7-6b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト 7-6b"
obj_p=0x00ECF79C
```

```
CDerivedTest7::Destructor : name="テスト7-6b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト7-6b"
delete(p=0x00ECF790, poly_allocator="CIStackAllocAdp":128/44/84)
----------CPoolAllocAdp
new(size=12, poly_allocator="CPoolAllocAdp":120/0/120)
  fileName="main.cpp(3105)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub1(const char *)",
                                  callPointName="(unknown call-point)", time=0.000, typename="class CTest7", align=4
  p=0x00ECF6AC
CTest7::Constructor : name="テスト7-7a"
obj_p=0x00ECF6AC
CTest7::Destructor : name="テスト7-7a"
delete(p=0x00ECF6AC, poly_allocator="CPoolAllocAdp":120/24/96)
new[](size=40, poly_allocator="CPoolAllocAdp":120/0/120)
  fileName="main.cpp(3118)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub2(void)",
                               callPointName="(unknown call-point)", time=0.000, typename="class CTest7 [3]", align=4
  p=0x00000000
obj_p=0x00000000
new(size=24, poly_allocator="CPoolAllocAdp":120/0/120)
  fileName="main.cpp(3131)  [Thu Feb 27 00:41:32 2014]", funcName="void __cdecltest7_sub3(const char *)",
                         callPointName="(unknown call-point)", time=0.000, typename="class CDerivedTest7", align=16
  p=0x00ECF6C4                ←プールアロケータはアラインメント計算に対応しない（今後も不要）
CTest7::Constructor : name="テスト7-7b"
CTest7ex::Constructor
CDerivedTest7::Constructor : name="テスト7-7b"
obj_p=0x00ECF6D0
CDerivedTest7::Destructor : name="テスト7-7b"
CTest7ex::Destructor
CTest7::Destructor : name="テスト7-7b"
delete(p=0x00ECF6C4, poly_allocator="CPoolAllocAdp":120/24/96)
```

▼ コールポイントについて

上記のメモリ確保情報では、中に「コールポイント」というものを扱っている。

これは、要するに、メモリ確保処理を呼び出す元になった処理のことである。

メモリ確保処理は共通処理に隠ぺいされることも多く、実行時の関数名を記録するだけでは、実際にどのような目的でメモリ確保が行われたのかがわからないことも多い。そのため、コールポイントという仕組みにより、その呼び出し元の処理がなんであったかを扱い、状況を分かり易くする。

コールポイントは、別紙の「効果的なデバッグログとアサーション」で提案するシステムである。

▼ 【応用サンプル】一時スタックアロケータで標準ライブラリを活用

グローバル多態アロケータを応用し、標準ライブラリを有効活用するサンプルを示す。

std::string を使用して文字列を操作するサンプルで説明する。ワークバッファ（一時スタックアロケータ）で文字列を作成し、内容が確定したのち、常駐データに移し替える処理を行う。

一時スタックアロケータで標準ライブラリを活用するサンプル：

【常駐データを定義】

```
//--------------------
//常駐データ（想定）
CStackAllocatorWithBuff<1024> s_parmaDataBuff;//常駐データ用バッファ
static char* s_message1 = nullptr;//常駐メッセージ１
static char* s_message2 = nullptr;//常駐メッセージ２
```

【ワークバッファを定義】

```
//--------------------
//ワークバッファ
CStackAllocatorWithBuff<4096> s_tempStack;//一時スタックアロケータ
```

【テスト】

```
#include <string.h>//strcpy 用
#include <string>//std::string 用
//--------------------
//テスト
void test8a()
{
	//一時スタックアロケータ使用
	{
		printf("----------一時スタックアロケータで文字列操作¥n");
		CTempPolyStackAllocator poly_allocator(s_tempStack, CIStackAllocAdp::AUTO_REWIND);
		{
			std::string str1 = "文字列と";
			std::string str2 = "文字列を";
			std::string str3 = "結合するような処理では、";
			std::string str4 = "std::string が";
			std::string str5 = "やっぱり便利!";
			std::string str = str1 + str2 + str3 + str4 + str5;
			str += "¥n" "しかし、素直に strcat()を使った方が高速";
			str += "¥n" "std::string は、find(), replace(), substr() などを使いたい時に便利";
			str += "¥n" "また、「効率化のために一時バッファでサイズを見積もってコピー」のような処理にも¥n 効果的";
			str = "【一時スタックアロケータ版】¥n" + str;
			printf("strpcy()¥n");
			const std::size_t size = str.length() + 1;
			s_message1 = static_cast<char*>(s_ s_parmaDataBuff.allocN(size, 1));
		#ifdef USE_STRCPY_S
			strcpy_s(s_message1, size, str.c_str());
		#else//USE_STRCPY_S
			strcpy(s_message1, str.c_str());
		#endif//USE_STRCPY_S
		}
	}
	//標準アロケータ（malloc / free）使用
	{
		printf("----------標準アロケータで文字列操作¥n");
		{
			std::string str1 = "文字列と";
			std::string str2 = "文字列を";
			std::string str3 = "結合するような処理では、";
			std::string str4 = "std::string が";
			std::string str5 = "やっぱり便利!";
			std::string str = str1 + str2 + str3 + str4 + str5;
			str += "¥n" "しかし、素直に strcat()を使った方が高速";
			str += "¥n" "std::string は、find(), replace(), substr() などを使いたい時に便利";
			str += "¥n" "また、「効率化のために一時バッファでサイズを見積もってコピー」のような処理にも¥n 効果的";
			str = "【標準アロケータ版】¥n" + str;
			const std::size_t size = str.length() + 1;
			s_message2 = static_cast<char*>(s_parmaDataBuff.allocN(size, 1));
		#ifdef USE_STRCPY_S
			strcpy_s(s_message2, size, str.c_str());
		#else//USE_STRCPY_S
```

```
                strcpy(s_message2, str.c_str());
            #endif//USE_STRCPY_S
            }
    }
    {
            printf("----------文字列操作の結果表示¥n");
            printf("s_message1=¥"%s¥"¥n", s_message1);
            printf("-----¥n");
            printf("s_message2=¥"%s¥"¥n", s_message2);
            printf("-----¥n");
            printf("s_parmaDataBuff=%d/%d/%d¥n", s_parmaDataBuff.getTotal(), s_parmaDataBuff.getUsed(),
                                                                        s_parmaDataBuff.getRemain());
    }
}
```

↓（実行結果）

```
----------一時スタックアロケータで文字列操作
new(size=8, poly_allocator="CStackAllocAdp":4096/0/4096)
  p=0x00989DD0
new(size=8, poly_allocator="CStackAllocAdp":4096/8/4088)
  p=0x00989DD8
new(size=8, poly_allocator="CStackAllocAdp":4096/16/4080)
  p=0x00989DE0
new(size=32, poly_allocator="CStackAllocAdp":4096/24/4072)
  p=0x00989DE8
new(size=8, poly_allocator="CStackAllocAdp":4096/56/4040)
  p=0x00989E08
new(size=8, poly_allocator="CStackAllocAdp":4096/64/4032)
  p=0x00989E10
new(size=8, poly_allocator="CStackAllocAdp":4096/72/4024)
  p=0x00989E18
new(size=32, poly_allocator="CStackAllocAdp":4096/80/4016)
  p=0x00989E20
new(size=8, poly_allocator="CStackAllocAdp":4096/112/3984)
  p=0x00989E40
delete(p=0x00989E18, poly_allocator="CStackAllocAdp":4096/120/3976)
new(size=48, poly_allocator="CStackAllocAdp":4096/120/3976)
  p=0x00989E48
delete(p=0x00989E20, poly_allocator="CStackAllocAdp":4096/168/3928)
new(size=8, poly_allocator="CStackAllocAdp":4096/168/3928)
  p=0x00989E78
new(size=71, poly_allocator="CStackAllocAdp":4096/176/3920)
  p=0x00989E80
delete(p=0x00989E48, poly_allocator="CStackAllocAdp":4096/247/3849)
new(size=8, poly_allocator="CStackAllocAdp":4096/247/3849)
  p=0x00989EC8
new(size=8, poly_allocator="CStackAllocAdp":4096/256/3840)
  p=0x00989ED0
delete(p=0x00989EC8, poly_allocator="CStackAllocAdp":4096/264/3832)
delete(p=0x00989E78, poly_allocator="CStackAllocAdp":4096/264/3832)
delete(p=0x00989E40, poly_allocator="CStackAllocAdp":4096/264/3832)
new(size=112, poly_allocator="CStackAllocAdp":4096/264/3832)
  p=0x00989ED8
delete(p=0x00989E80, poly_allocator="CStackAllocAdp":4096/376/3720)
new(size=176, poly_allocator="CStackAllocAdp":4096/376/3720)
  p=0x00989F48
delete(p=0x00989ED8, poly_allocator="CStackAllocAdp":4096/552/3544)
new(size=272, poly_allocator="CStackAllocAdp":4096/552/3544)
  p=0x00989FF8
delete(p=0x00989F48, poly_allocator="CStackAllocAdp":4096/824/3272)
new(size=8, poly_allocator="CStackAllocAdp":4096/824/3272)
  p=0x0098A108
new(size=288, poly_allocator="CStackAllocAdp":4096/832/3264)
  p=0x0098A110
new(size=8, poly_allocator="CStackAllocAdp":4096/1120/2976)
```

←(new 演算子処理内のプリント文により)std::string がスタックアロケータを使用して処理していることがわかる

```
  p=0x0098A230
delete(p=0x0098A108, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x00989FF8, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x0098A230, poly_allocator="CStackAllocAdp":4096/1128/2968)
strpcy()          ←この時点で文字列をコピー
delete(p=0x0098A110, poly_allocator="CStackAllocAdp":4096/1128/2968)    ←std::string のデストラクタで後始末
delete(p=0x00989ED0, poly_allocator="CStackAllocAdp":4096/1128/2968)      （スタックアロケータは削除要求を無視）
delete(p=0x00989E10, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x00989E08, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x00989DE8, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x00989DE0, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x00989DD8, poly_allocator="CStackAllocAdp":4096/1128/2968)
delete(p=0x00989DD0, poly_allocator="CStackAllocAdp":4096/1128/2968)
----------標準アロケータで文字列操作
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)      ←一時スタックアロケータの使用を終えると、自動的に標準アロケータ
  p=0x00C7E460                                          に切り替わっている
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E4B0
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E500
new(size=32, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E550
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E5B8
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E608
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E658
new(size=32, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E6A8
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E710
delete(p=0x00C7E658, poly_allocator="CStdAllocAdp":0/0/0)
new(size=48, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E760
delete(p=0x00C7E6A8, poly_allocator="CStdAllocAdp":0/0/0)
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E658
new(size=71, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E7D8
delete(p=0x00C7E760, poly_allocator="CStdAllocAdp":0/0/0)
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E6A8
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E760
delete(p=0x00C7E6A8, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E658, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E710, poly_allocator="CStdAllocAdp":0/0/0)
new(size=112, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E658
delete(p=0x00C7E7D8, poly_allocator="CStdAllocAdp":0/0/0)
new(size=176, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E7B0
delete(p=0x00C7E658, poly_allocator="CStdAllocAdp":0/0/0)
new(size=272, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E8A8
delete(p=0x00C7E7B0, poly_allocator="CStdAllocAdp":0/0/0)
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E7B0
new(size=288, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7EA00
new(size=8, poly_allocator="CStdAllocAdp":0/0/0)
  p=0x00C7E800
delete(p=0x00C7E7B0, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E8A8, poly_allocator="CStdAllocAdp":0/0/0)
```

```
delete(p=0x00C7E800, poly_allocator="CStdAllocAdp":0/0/0)
strpcy()          ←この時点で文字列をコピー
delete(p=0x00C7EA00, poly_allocator="CStdAllocAdp":0/0/0)    ←std::string のデストラクタで後始末
delete(p=0x00C7E760, poly_allocator="CStdAllocAdp":0/0/0)      （標準アロケータはきちんとメモリ解放）
delete(p=0x00C7E608, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E5B8, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E550, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E500, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E4B0, poly_allocator="CStdAllocAdp":0/0/0)
delete(p=0x00C7E460, poly_allocator="CStdAllocAdp":0/0/0)
----------文字列操作の結果表示
s_message1="【一時スタックアロケータ版】
文字列と文字列を結合するような処理では、std::string がやっぱり便利!
しかし、素直に strcat()を使った方が高速
std::string は、find(), replace(), substr() などを使いたい時に便利
また、「効率化のために一時バッファでサイズを見積もってコピー」のような処理にも
効果的"
-----
s_message2="【標準アロケータ版】
文字列と文字列を結合するような処理では、std::string がやっぱり便利!
しかし、素直に strcat()を使った方が高速
std::string は、find(), replace(), substr() などを使いたい時に便利
また、「効率化のために一時バッファでサイズを見積もってコピー」のような処理にも
効果的"
-----
s_parmaDataBuff=1024/568/456  ←実際の文字列長の分だけ常駐データ用のメモリを消費している
                                （処理中は、一時スタックアロケータだけで 1128 バイトを消費していた）
s_tempStack=4096/0/4096       ←一時スタックアロケータは、自動巻き戻しが効いて使用量 0 に戻っている
```

▼ 【応用サンプル】標準ライブラリのクラスを内包したクラス

もう一つ、有効な応用サンプルを示す。

標準ライブラリのハッシュテーブル（std::unordered_map）を内包したクラスを作成する。

ハッシュテーブルはゲームプログラミングでも使いどころが多い。高速検索のための常駐テーブルに向いていて、一度データが確定すると、以後メモリ操作が（基本的には）発生しない点も扱い易さの一つ。

問題点としては、テーブルのデータを追加する過程でヒープメモリを使用する点。データが常駐することから、他のデータが削除されてもゴミのようにヒープ上に点在し続ける可能性がある。

この問題の対処として、通常は、カスタムアロケータを作成することや、標準ライブラリを使用しないことで対応する。ここでは、グローバル多態アロケータとスタックアロケータを活用し、クラス内のバッファを使用することで対応する。処理の状態を確認するために、クラスの処理にはプリント文を仕込んでいる。

なお、std::unordered_map は C++11 から追加されたクラスである。赤黒木による連想配列コンテナの std::map とほぼ同じインターフェースを備える。

また、CRC 算出処理については、別紙の「効果的なテンプレートテクニック」にサンプルプログラムを掲載している。

標準ハッシュテーブルを内包したクラスのサンプル：

【共通関数参照と型定義】

```
//--------------------
//型定義
typedef unsigned int crc32_t;//CRC32 型

//--------------------
//共通関数参照
crc32_t calcCRC32(const char* str); //文字列から CRC 算出
```

【ハッシュテーブルクラス定義】

```
//--------------------
//標準ライブラリを利用した固定バッファハッシュテーブルクラス
#include <string.h>//strcpy 用
#include <unordered_map>//C++11 ハッシュテーブル
class CTest8
{
public:
	//型
	//データ型
	struct DATA
	{
		char m_name[20];//名前
		int m_age;//年齢
		//コンストラクタ
		DATA(const char* name, const int age)
		{
		#ifdef USE_STRCPY_S
			strcpy_s(m_name, sizeof(m_name), name);
		#else//USE_STRCPY_S
			strcpy(m_name, name);
		#endif//USE_STRCPY_S
			m_age = age;
		}
	};
public:
	//メソッド
	//データ追加
	void addData(const DATA& data)
	{
		printf("CTest8::addData()¥n");
		CTempPolyStackAllocator poly_allocator(m_stack);//多態アロケータをクラス内スタックに変更
		m_table->emplace(calcCRC32(data.m_name), data);//コンテナ内に要素を構築
		//m_table->insert(std::make_pair(calcCRC32(data.m_name), data));//insert を使用する場合
	}
	//データ参照
	const DATA* find(const crc32_t name_crc) const
	{
		printf("CTest8::find()¥n");
		const auto& obj = m_table->find(name_crc);//キー（CRC）でテーブル検索
		if (obj == m_table->cend())//見つからなかったか？
			return nullptr;
		return &obj->second;//データ部を返す
	}
	const DATA* find(const char* name) const
	{
		return find(calcCRC32(name));
	}
public:
	//コンストラクタ
```

```
	CTest8()
	{
		printf("CTest8::Consructor()¥n");
		CTempPolyStackAllocator poly_allocator(m_stack);//多態アロケータをクラス内スタックに変更
		printf("new unordered_map¥n");
		m_table = new std::unordered_map<crc32_t, DATA>();//ハッシュテーブル生成
		printf("unordered_map::reserve¥n");
		m_table->reserve(10);//あらかじめテーブル数を予約（少しでもメモリ効率をよくするため）
		printf("--¥n");
	}
	//デストラクタ
	~CTest8()
	{
		printf("CTest8::Destructor()¥n");
		CTempPolyStackAllocator poly_allocator(m_stack);//多態アロケータをスタックに変更
		printf("delete unordered_map¥n");
		delete m_table;//ハッシュテーブルを破棄
		printf("--¥n");
	}
private:
	//フィールド
	std::unordered_map<crc32_t, DATA>* m_table;//ハッシュテーブル
	CStackAllocatorWithBuff<1024> m_stack;//クラス内スタック
};
```

【テスト】

```
//--------------------
//テスト
void test8b()
{
	//標準ライブラリのクラスを内包したクラスのテスト
	{
		printf("----------標準ライブラリのクラスを内包したクラスを操作¥n");
		CTempPolyStackAllocator poly_allocator(s_tempStack, CIStackAllocAdp::AUTO_REWIND);//一時スタックアロケータ
		{
			//テーブル登録用のデータを準備
			CTest8::DATA data1("太郎", 40);
			CTest8::DATA data2("次郎", 30);
			CTest8::DATA data3("三郎", 20);

			//クラスのインスタンスを生成（一時スタックアロケータに作成）
			printf("new CTest8¥n");
			CTest8* table = new CTest8();

			//テーブルにデータを登録
			table->addData(data1);
			table->addData(data2);
			table->addData(data3);

			//テーブルからデータを参照
			const CTest8::DATA* ref1 = table->find("太郎");
			const CTest8::DATA* ref2 = table->find("次郎");
			const CTest8::DATA* ref3 = table->find("三郎");
			const CTest8::DATA* ref4 = table->find("四朗");
			if (ref1) printf("ref1: name=¥"%s¥", age=%d¥n", ref1->m_name, ref1->m_age);
			if (ref2) printf("ref2: name=¥"%s¥", age=%d¥n", ref2->m_name, ref2->m_age);
			if (ref3) printf("ref3: name=¥"%s¥", age=%d¥n", ref3->m_name, ref3->m_age);
			if (ref4) printf("ref4: name=¥"%s¥", age=%d¥n", ref4->m_name, ref4->m_age);

			//インスタンス破棄
			printf("delete CTest8¥n");
			delete table;
		}
	}
}
```

↓（実行結果）

```
----------標準ライブラリのクラスを内包したクラスを操作
new CTest8
new(size=1044, poly_allocator="CStackAllocAdp":4096/0/4096)  ←まず、CTest8 のインスタンスが一時スタックアロケータ上に
  p=0x00BBA910                                                 生成されている（スタックサイズで判断）
CTest8::Consructor()
new unordered_map
new(size=40, poly_allocator="CStackAllocAdp":1024/0/1024)    ←続いて、クラス内のハッシュテーブルは、クラス内スタック
  p=0x00BBA924                                                 上に生成されている（スタックサイズで判断）
new(size=36, poly_allocator="CStackAllocAdp":1024/40/984)    ※インスタンス生成時に何度かメモリ確保が行われている
  p=0x00BBA94C                                                 ことがわかる
new(size=8, poly_allocator="CStackAllocAdp":1024/76/948)
  p=0x00BBA970
new(size=8, poly_allocator="CStackAllocAdp":1024/84/940)
  p=0x00BBA978
new(size=64, poly_allocator="CStackAllocAdp":1024/92/932)
  p=0x00BBA980
unordered_map::reserve
new(size=128, poly_allocator="CStackAllocAdp":1024/156/868)
  p=0x00BBA9C0
delete(p=0x00BBA980, poly_allocator="CStackAllocAdp":1024/284/740)
--
CTest8::addData()
new(size=36, poly_allocator="CStackAllocAdp":1024/284/740)   ←reserve していても、追加のたびにメモリは確保される
  p=0x00BBAA40
CTest8::addData()
new(size=36, poly_allocator="CStackAllocAdp":1024/320/704)
  p=0x00BBAA64
CTest8::addData()
new(size=36, poly_allocator="CStackAllocAdp":1024/356/668)
  p=0x00BBAA88
CTest8::find()     ←find()ではとくにメモリ確保が行われない
CTest8::find()       （キーが std::string だったりするとメモリ確保が発生するので注意）
CTest8::find()
CTest8::find()
ref1: name="太郎", age=40
ref2: name="次郎", age=30
ref3: name="三郎", age=20
                          ←「四郎」の検索は失敗（ハッシュテーブルに記録されていないため）
delete CTest8
CTest8::Destructor()      ←delete により、CTest8 のデストラクタが実行
delete unordered_map
delete(p=0x00BBA9C0, poly_allocator="CStackAllocAdp":1024/392/632)     ←続いて、クラス内のハッシュテーブルが
delete(p=0x00BBA978, poly_allocator="CStackAllocAdp":1024/392/632)       正しくクラス内スタックから削除されようと
delete(p=0x00BBAA40, poly_allocator="CStackAllocAdp":1024/392/632)       していることがわかる（スタックサイズで判断）
delete(p=0x00BBAA64, poly_allocator="CStackAllocAdp":1024/392/632)       ※ただし、スタックは実際にメモリ解放を
delete(p=0x00BBAA88, poly_allocator="CStackAllocAdp":1024/392/632)         行わない
delete(p=0x00BBA94C, poly_allocator="CStackAllocAdp":1024/392/632)
delete(p=0x00BBA970, poly_allocator="CStackAllocAdp":1024/392/632)
delete(p=0x00BBA924, poly_allocator="CStackAllocAdp":1024/392/632)
--
delete(p=0x00BBA910, poly_allocator="CStackAllocAdp":4096/1044/3052)   ←最終的に、CTest8 のインスタンスが
                                                                         一時スタックアロケータから削除されようと
                                                                         していることがわかる（スタックサイズで判断）
                                                                         ※ただし、スタックは実際にメモリ解放を
                                                                           行わない
                                                                         ※一時スタックアロケータは、この後の自動
                                                                           巻き戻し処理で元の状態に戻る
```

■ スマートスタックアロケータ

最後に、マルチスレッドで有効なスタックアロケータを説明する。

▼ マルチスレッドのためのスタックアロケータ

マルチスレッドでフレームアロケータや一時スタックアロケータを使用することを考えた場合、スレッドごとにインスタンスを用意できるとロックフリーで高速な処理が実現できてよい。反面、メモリの無駄が大きく、スレッドごとに最適なサイズを決めるのも難しい。

ロック制御が必要になるが、複数のスレッドでアロケータを共有すればメモリは効率化できる。しかし、「自動巻き戻し」のような機能は他のスレッドに影響を与えてしまうので使えない。つまり、一時スタックアロケータとしてスレッド間で共通利用するするこは不可能である。

これらの問題を解決し、スレッド間でスタックアロケータを共有するための仕組みとして、「スマートスタックアロケータ」を用いる。

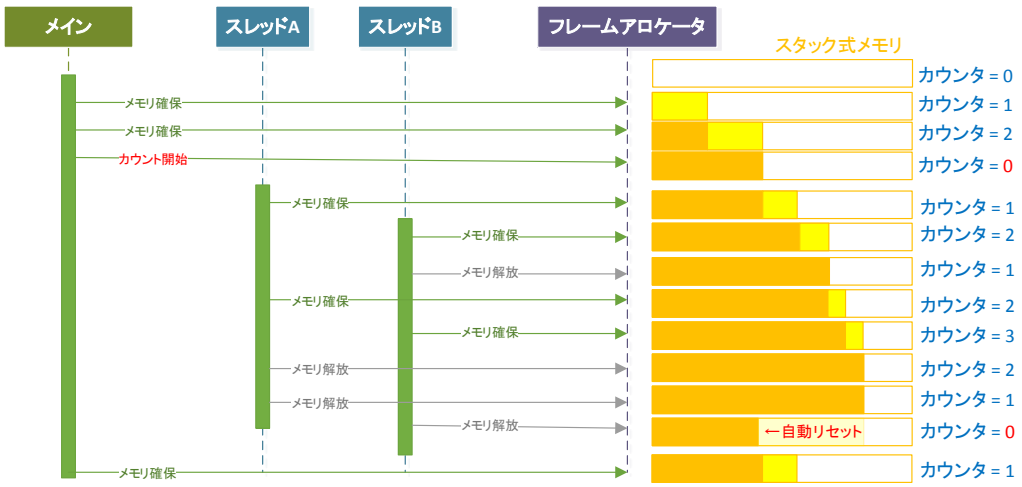
● スマートスタックアロケータの仕組み

スマートスタックアロケータは、以下の点で通常のスタックアロケータと異なる。

- 標準的にロック制御を備え、排他的にメモリの確保と解放を行う。
- 参照カウンタを持ち、メモリ確保の際にカウントアップする。
- メモリの開放のインターフェースを持ち、参照カウンタをカウントダウンする。
- 参照カウンタが 0 になった時点で、自動的にマーカーを初期位置に戻す。
- 任意のタイミングでカウントを開始し、初期位置を設定することができる。

これにより、スレッド間でスタックアロケータを安全に共通利用しつつ、メモリが不要になった時点で自動的に開放されるようにできる。

スマートスタックアロケータの使用イメージ：



スマートスタックアロケータの使用には、十分に注意を払わなければならない。以下、その注意点を示す。

- 確保したメモリは必ず明示的に解放しなければならない。
  - 通常、スタックアロケータで確保したメモリは、明示的に開放しないことも多い。
  - 一つでも解放漏れがあると、一切のメモリが解放されず、すぐにバッファを使い尽くしてしまう。
- 全てのメモリが解放されるタイミングがあることを保証しなければならない。
  - 長期的なメモリ確保（開放するのが数フレーム後）を行ってはならない。
  - 複数のスレッドで絶え間ないメモリ確保と解放が継続するような処理に使ってはならない。

なお、スマートスタックアロケータは、双方向スタックアロケータとして機能し、かつ、共通のスタックアロケータインターフェースに対応する。

● 【注意】「スマートスタックアロケータ」という用語について

「スマートスタックアロケータ」という用語は本書における造語であることに注意。「スマートポインタ」と同様に、参照カウンタを使って自動的に破棄する動作を行うことから命名したものである。

▼ スマートスタックアロケータのサンプル

スマートスタックアロケータのサンプルを示す。

前述の「スタックアロケータ」、「共通アロケータインターフェース」、「グローバル多態アロケータ」を踏まえたものとして説明する。

マルチスレッド対応のために、別紙の「効率化と安全性のためのロック制御」に掲載する「スレッド ID クラス」と「軽量スピンロッククラス」を使用する。また、双方向スタックアロケートと同様の処理が多いが、幾つかの処理をアトミック操作に変更する。

スマートスタックアロケータのサンプル：

【スマートスタックアロケータクラス定義】

```
//--------------------
//スマートスタックアロケータクラス
//※非スレッドセーフ（処理速度優先）
//※インターフェースを実装するが、高速化のために virtual メソッドに頼らず操作可能
class CSmartStackAllocator : public IStackAllocator
{
public:
	//型
	typedef int counter_t;//カウンタ型
public:
	//アクセッサ
	std::size_t getTotal() const override { return m_buffSize; }//全体のメモリ量を取得
	std::size_t getUsed() const override { return m_usedN.load() + m_buffSize - m_usedR.load(); }
                                                                    //使用中のメモリ量を取得
	std::size_t getRemain() const override { return m_usedR.load() - m_usedN.load(); }//残りのメモリ量を取得
	E_ORDERED getDefaultOrdered() const { return m_defaultOrdered.load(); }//デフォルトのスタック順を取得
	void setDefaultOrdered(const E_ORDERED ordered)//デフォルトのスタック順を更新
	{
		m_defaultOrdered.store(ordered == REVERSE ? REVERSE : NORMAL);
	}
	const byte* getBuff() const { return m_buffPtr; }//バッファ取得を取得
	const byte* getNowPtrN() const { return m_buffPtr + m_usedN.load(); }//現在のバッファ位置（正順）を取得
	const byte* getNowPtrR() const { return m_buffPtr + m_usedR.load(); }//現在のバッファ位置（逆順）を取得
	const byte* getNowPtrD() const { return getNowPtr(m_defaultOrdered.load()); }//現在のバッファ位置を取得
	const byte* getNowPtr(const E_ORDERED ordered) const {
		return ordered == DEFAULT ? getNowPtrD() : ordered == REVERSE ? getNowPtrR() : getNowPtrN(); }
                                                                    //現在のバッファ位置を取得
	const byte* getNowPtr() const override { return getNowPtrD(); }//現在のバッファ位置を取得
	marker_t getMarkerN() const { return m_usedN.load(); }//現在のマーカー（正順）を取得
	marker_t getMarkerR() const { return m_usedR.load(); }//現在のマーカー（逆順）を取得
	marker_t getMarkerD() const { return getMarker(m_defaultOrdered.load()); }//現在のマーカーを取得
	marker_t getMarker(const E_ORDERED ordered) const {
		return ordered == DEFAULT ? getMarkerD() : ordered == REVERSE ? getMarkerR() : getMarkerN(); }
                                                                    //現在のマーカーを取得
	marker_t getMarker() const override { return getMarkerD(); }//現在のマーカーを取得
	marker_t getBeginN() const { return m_beginN.load(); }//開始マーカー（正順）を取得
	marker_t getBeginR() const { return m_beginR.load(); }//開始マーカー（逆順）を取得
	marker_t getBeginD() const { return getBegin(m_defaultOrdered.load()); }//開始マーカーを取得
	marker_t getBegin(const E_ORDERED ordered) const {
			return ordered == DEFAULT ? getBeginD() : ordered == REVERSE ? getBeginR() : getBeginN(); }
                                                                    //開始マーカーを取得
	marker_t getBegin() const { return getBeginD(); }//開始マーカーを取得
	marker_t getCounterN() const { return m_counterN.load(); }//メモリ確保カウンタ（正順）を取得
	marker_t getCounterR() const { return m_counterR.load(); }//メモリ確保カウンタ（逆順）を取得
	marker_t getCounterD() const { return getCounter(m_defaultOrdered.load()); }//メモリ確保カウンタを取得
	marker_t getCounter(const E_ORDERED ordered) const {
		return ordered == DEFAULT ? getCounterD() : ordered == REVERSE ? getCounterR() : getCounterN(); }
                                                                    //メモリ確保カウンターを取得
	marker_t getCounter() const { return getCounterD(); }//メモリ確保カウンタを取得
public:
	//メソッド
	//メモリ確保（正順）
	void* allocN(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
	{
		//高速化のために、一度ロックの範囲外で単純空き領域判定
		if (m_usedN.load() + size > m_usedR.load())
```

```
                return nullptr;
            m_lock.lock();//ロック取得
            m_counterN.fetch_add(1);
            const uintptr_t now_ptr = reinterpret_cast<uintptr_t>(m_buffPtr)+m_usedN.load();//現在のポインタ位置算出
            const uintptr_t align_diff = align > 0 ? now_ptr % align == 0 ? 0 : align - now_ptr % align : 0;
                                                                                    //アラインメント計算
            const marker_t next_used = m_usedN.load() + align_diff + size;//次のマーカー算出
            if (next_used > m_usedR.load())//メモリオーバーチェック（符号なしなので、範囲チェックは大判定のみでOK）
            {
                printf("normal-stack overflow!(size=%d+align=%d, remain=%d)¥n",
                                                        size, align_diff, m_usedR.load() - m_usedN.load());
                m_lock.unlock();//ロック解放
                return nullptr;
            }
            const uintptr_t alloc_ptr = now_ptr + align_diff;//メモリ確保アドレス算出
            m_usedN.store(next_used);//マーカー更新
            m_lock.unlock();//ロック解放
            return reinterpret_cast<void*>(alloc_ptr);
        }
        //メモリ確保（逆順）
        void* allocR(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
        {
            //高速化のために、一度ロックの範囲外で単純空き領域判定
            if (m_usedN.load() > m_usedR.load() - size)
                return nullptr;
            m_lock.lock();//ロック取得
            m_counterR.fetch_add(1);
            const uintptr_t now_ptr = reinterpret_cast<uintptr_t>(m_buffPtr)+m_usedR.load();//現在のポインタ位置算出
            const uintptr_t alloc_ptr_tmp = now_ptr - size;//メモリ確保アドレス算出（暫定）
            const uintptr_t align_diff = align > 0 ? alloc_ptr_tmp % align == 0 ? 0 : alloc_ptr_tmp % align : 0;
                                                                                    //アラインメント計算
            const marker_t next_used = m_usedR.load() - size - align_diff;//次のマーカー算出
            if (next_used < m_usedN.load() || next_used > m_buffSize)
                                        //メモリオーバーチェック（オーバーフローして値が大きくなる可能性もチェック）
            {
                printf("reversed-stack overflow!(size=%d+align=%d, remain=%d)¥n",
                                                        size, align_diff, m_usedR.load() - m_usedN.load());
                m_lock.unlock();//ロック解放
                return nullptr;
            }
            const uintptr_t alloc_ptr = now_ptr - size - align_diff;//メモリ確保アドレス算出
            m_usedR.store(next_used);//マーカー更新
            m_lock.unlock();//ロック解放
            return reinterpret_cast<void*>(alloc_ptr);
        }
        //メモリ確保
        void* allocD(const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
        {
            return alloc(m_defaultOrdered.load(), size, align);
        }
        //メモリ確保
        void* alloc(const E_ORDERED ordered, const std::size_t size, const std::size_t align = DEFAULT_ALIGN)
        {
            return ordered == DEFAULT ? allocD(size, align) : ordered == REVERSE ? allocR(size, align) :
                                                                                    allocN(size, align);
        }
        //メモリ確保
        void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN) override
        {
            return allocD(size, align);
        }
    private:
        //メモリ破棄（正順）
        void freeN(void* p)
        {
```

```
		//ポインタの範囲チェック
		//※高速化のために、一度ロックの範囲外でチェック
		//　（常に freeN 関数と freeD が一組で呼び出されるため）
		if (m_counterN.load() <= 0)
			return;
		const marker_t now_n = reinterpret_cast<uintptr_t>(p)-reinterpret_cast<uintptr_t>(m_buffPtr);
		if (now_n < m_beginN.load() || now_n > m_usedN.load())
			return;
		//ポインタが正順スタックとして適正のため、処理開始
		m_lock.lock();//ロック取得
		if (m_counterN.load() <= 0 || now_n < m_beginN.load() || now_n > m_usedN.load())//範囲の再チェック
		{
			m_lock.unlock();//ロック解放
			return;
		}
		m_counterN.fetch_sub(1);
		if (m_counterN.load() == 0)
			m_usedN.store(m_beginN.load());
		m_lock.unlock();//ロック解放
	}
	//メモリ破棄（逆順）
	void freeR(void* p)
	{
		//ポインタの範囲チェック
		//※高速化のために、一度ロックの範囲外でチェック
		//　（常に freeN 関数と freeD が一組で呼び出されるため）
		if (m_counterR.load() <= 0)
			return;
		const marker_t now_r = reinterpret_cast<uintptr_t>(p)-reinterpret_cast<uintptr_t>(m_buffPtr);
		if (now_r >= m_beginR.load() || now_r < m_usedR.load())
			return;
		//ポインタが逆順スタックとして適正のため、処理開始
		m_lock.lock();//ロック取得
		if (m_counterR.load() <= 0 || now_r >= m_beginR.load() || now_r < m_usedR.load())//範囲の再チェック
		{
			m_lock.unlock();//ロック解放
			return;
		}
		m_counterR.fetch_sub(1);
		if (m_counterR.load() == 0)
			m_usedR.store(m_beginR.load());
		m_lock.unlock();//ロック解放
	}
public:
	//メモリ破棄
	void free(void* p)
	{
		//ポインタをチェックして処理するので正順と逆順の両方の処理をまとめて実行する
		freeN(p);//正順
		freeR(p);//逆順
	}
	//マーカー位置を記憶してメモリ確保のカウントを開始（正順）
	//※メモリ確保カウンタをリセット
	void beginN()
	{
		m_lock.lock();//ロック取得
		m_beginN.store(m_usedN.load());
		m_counterN.store(0);
		m_lock.unlock();//ロック解放
	}
	//マーカー位置を記憶してメモリ確保のカウントを開始（逆順）
	//※メモリ確保カウンタをリセット
	void beginR()
	{
		m_lock.lock();//ロック取得
```

```
		m_beginR.store(m_usedR.load());
		m_counterR.store(0);
		m_lock.unlock();//ロック解放
	}
	//マーカー位置を記憶してメモリ確保のカウントを開始
	//※メモリ確保カウンタをリセット
	void beginD()
	{
		begin(m_defaultOrdered.load());
	}
	//マーカー位置を記憶してメモリ確保のカウントを開始
	//※メモリ確保カウンタをリセット
	void begin(const E_ORDERED ordered)
	{
		ordered == DEFAULT ? beginD() : ordered == REVERSE ? beginR() : beginN();
	}
	//マーカー位置を記憶してメモリ確保のカウントを開始
	//※メモリ確保カウンタをリセット
	void begin()
	{
		beginD();
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す（正順）
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※マーカー指定版
	void backN(const marker_t marker_n)
	{
		m_lock.lock();//ロック取得
		if (marker_n > m_usedR.load())//符号なしなので、範囲チェックは大判定のみで OK
		{
			m_lock.unlock();//ロック解放
			return;
		}
		m_usedN.store(marker_n);
		m_beginN.store(marker_n);
		m_counterN.store(0);
		m_lock.unlock();//ロック解放
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す（正順）
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※ポインタ指定版
	void backN(const void* p)
	{
		const marker_t marker = reinterpret_cast<uintptr_t>(p)-reinterpret_cast<uintptr_t>(m_buffPtr);
		backN(marker);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す（逆順）
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※マーカー指定版
	void backR(const marker_t marker_r)
	{
		m_lock.lock();//ロック取得
		if (marker_r < m_usedN.load() || marker_r > m_buffSize)//メモリオーバーチェック
		{
			return;
			m_lock.unlock();//ロック解放
		}
		m_usedR.store(marker_r);
		m_beginR.store(marker_r);
		m_counterR.store(0);
		m_lock.unlock();//ロック解放
	}
```

```
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す（逆順）
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※ポインタ指定版
	void backR(const void* p)
	{
		const marker_t marker = reinterpret_cast<uintptr_t>(p)-reinterpret_cast<uintptr_t>(m_buffPtr);
		backR(marker);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※マーカー指定版
	void backD(const marker_t marker)
	{
		back(m_defaultOrdered.load(), marker);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※ポインタ指定版
	void backD(const void* p)
	{
		back(m_defaultOrdered.load(), p);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※マーカー指定版
	void back(const E_ORDERED ordered, const marker_t marker)
	{
		ordered == DEFAULT ? backD(marker) : ordered == REVERSE ? backR(marker) : backN(marker);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※ポインタ指定版
	void back(const E_ORDERED ordered, const void* p)
	{
		ordered == DEFAULT ? backD(p) : ordered == REVERSE ? backR(p) : backN(p);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※マーカー指定版
	void back(const marker_t marker) override
	{
		backD(marker);
	}
	//【このクラスでは非推奨メソッド】
	//メモリを以前のマーカーに戻す
	//※開始マーカーとメモリ確保カウンタもリセットする
	//※ポインタ指定版
	void back(const void* p) override
	{
		backD(p);
	}
	//メモリ破棄（正順）
	//※開始マーカーとメモリ確保カウンタもリセットする
	void clearN()
	{
		m_lock.lock();//ロック取得
		m_usedN.store(0);
		m_beginN.store(0);
		m_counterN.store(0);
```

```
            m_lock.unlock();//ロック解放
        }
        //メモリ破棄（逆順）
        //※開始マーカーとメモリ確保カウンタもリセットする
        void clearR()
        {
            m_lock.lock();//ロック取得
            m_usedR.store(m_buffSize);
            m_beginR.store(0);
            m_counterR.store(0);
            m_lock.unlock();//ロック解放
        }
        //メモリ破棄
        //※開始マーカーとメモリ確保カウンタもリセットする
        void clearD()
        {
            clear(m_defaultOrdered.load());
        }
        //メモリ破棄（両方）
        //※開始マーカーとメモリ確保カウンタもリセットする
        void clearNR()
        {
            clearN();
            clearR();
        }
        //メモリ破棄
        //※開始マーカーとメモリ確保カウンタもリセットする
        void clear(const E_ORDERED ordered)
        {
            ordered == DEFAULT ? clearD() : ordered == REVERSE ? clearR() : clearN();
        }
        //メモリ破棄
        //※開始マーカーとメモリ確保カウンタもリセットする
        void clear() override
        {
            clearD();
        }
        //メモリ破棄（全て）
        //※開始マーカーとメモリ確保カウンタもリセットする
        void clearAll() override
        {
            clearNR();
        }
    public:
        //コンストラクタ
        CSmartStackAllocator(void* buff_p, const std::size_t buff_size, const E_ORDERED default_ordered = NORMAL) :
            m_buffPtr(reinterpret_cast<byte*>(buff_p)),//バッファ先頭アドレス
            m_buffSize(buff_size)//バッファサイズ
        {
            m_usedN.store(0);//マーカー（正順）
            m_usedR.store(buff_size);//マーカー（逆順）
            m_beginN.store(0);//開始マーカー（正順）※カウントを開始した位置（自動開放でこの位置に戻す）
            m_beginR.store(buff_size);//開始マーカー（逆順）※カウントを開始した位置（自動開放でこの位置に戻す）
            m_counterN.store(0);//メモリ確保カウンタ（正順）
            m_counterR.store(0);//メモリ確保カウンタ（逆順）
            setDefaultOrdered(default_ordered);//デフォルトのスタック順
        }
        //デストラクタ
        ~CSmartStackAllocator() override
        {}
    private:
        //フィールド
        byte* m_buffPtr;//バッファ先頭アドレス
        const std::size_t m_buffSize;//バッファサイズ
        std::atomic<marker_t> m_usedN;//マーカー（正順）
```

```
    std::atomic<marker_t> m_usedR;//マーカー（逆順）
    std::atomic<marker_t> m_beginN;//開始マーカー（正順）※カウントを開始した位置（自動開放でこの位置に戻す）
    std::atomic<marker_t> m_beginR;//開始マーカー（逆順）※カウントを開始した位置（自動開放でこの位置に戻す）
    std::atomic<counter_t> m_counterN;//メモリ確保カウンタ（正順）
    std::atomic<counter_t> m_counterR;//メモリ確保カウンタ（逆順）
    std::atomic<E_ORDERED> m_defaultOrdered;//デフォルトのスタック順
    CSpinLock m_lock;//ロック
};
```

【参考：スマートスタックアロケータ用配置 new / 配置 delete 定義】※このサンプルでは不要

```
//--------------------
//配置 new/配置 delete
//※メモリ使用状況を確認するためにマーカーを表示
//配置 new
void* operator new(const std::size_t size, CSmartStackAllocator& allocator, const std::size_t align,
                                        const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
    printf("placement new(size=%d, smart_allocator.marker=%d,%d, align=%d, ordered=%d)¥n",
                                            size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
    return allocator.alloc(ordered, size, align);
}
void* operator new(const std::size_t size, CSmartStackAllocator& allocator,
                                        const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
    const std::size_t align = IStackAllocator::DEFAULT_ALIGN;
    printf("placement new(size=%d, smart_allocator.marker=%d,%d, (align=%d), ordered=%d)¥n",
                                            size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
    return allocator.alloc(ordered, size, align);
}
//配列版
void* operator new[](const std::size_t size, CSmartStackAllocator& allocator,
                const std::size_t align, const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
    printf("placement new(size=%d, smart_allocator.marker=%d,%d, align=%d, ordered=%d)¥n",
                                            size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
    return allocator.alloc(ordered, size, align);
}
void* operator new[](const std::size_t size, CSmartStackAllocator& allocator,
                                        const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) throw()
{
    const std::size_t align = IStackAllocator::DEFAULT_ALIGN;
    printf("placement new(size=%d, smart_allocator.marker=%d,%d, (align=%d), ordered=%d)¥n",
                                            size, allocator.getMarkerN(), allocator.getMarkerR(), align, ordered);
    return allocator.alloc(ordered, size, align);
}
//配置 delete
void operator delete(void* p, CSmartStackAllocator& allocator, const std::size_t align,
                                                                        const IStackAllocator::E_ORDERED) throw()
{
    printf("placement delete(p=0x%p, smart_allocator.marker=%d,%d, align)¥n",
                                                            p, allocator.getMarkerN(), allocator.getMarkerR());
    allocator.free(p);
}
void operator delete(void* p, CSmartStackAllocator& allocator, const IStackAllocator::E_ORDERED) throw()
{
    printf("placement delete(p=0x%p, smart_allocator.marker=%d,%d)¥n",
                                                            p, allocator.getMarkerN(), allocator.getMarkerR());
    allocator.free(p);
}
//配列版
void operator delete[](void* p, CSmartStackAllocator& allocator, const std::size_t align,
                                                            const IStackAllocator::E_ORDERED ordered) throw()
{
    printf("placement delete[](p=0x%p, smart_allocator.marker=%d,%d, align)¥n",
                                                            p, allocator.getMarkerN(), allocator.getMarkerR());
```

```
	allocator.free(p);
}
void operator delete[](void* p, CSmartStackAllocator& allocator, const IStackAllocator::E_ORDERED ordered) throw()
{
	const std::size_t align = IStackAllocator::DEFAULT_ALIGN;
	printf("placement delete[](p=0x%p, smart_allocator.marker=%d,%d)¥n",
		p, allocator.getMarkerN(), allocator.getMarkerR());
	allocator.free(p);
}
```

【参考：スマートスタックアロケータ用 delete 関数定義】※このサンプルでは不要

```
//デストラクタ呼び出し付き delete
template<class T>
void delete_ptr(T*& p, CSmartStackAllocator& allocator)
{
	printf("delete_ptr(p=0x%p, smart_allocator.marker=%d,%d)¥n", p, allocator.getMarkerN(), allocator.getMarkerR());
	if (!p)
		return;
	p->~T();//デストラクタ呼び出し
	operator delete(p, allocator, IStackAllocator::DEFAULT);
	p = nullptr;//安全のためにポインタを初期化
}
//配列版
template<class T>
void delete_ptr(T*& p, const std::size_t array_num, CSmartStackAllocator& allocator)
{
	printf("delete_ptr(p=0x%p, array_num=%d, smart_allocator.marker=%d,%d)¥n",
		p, array_num, allocator.getMarkerN(), allocator.getMarkerR());
	if (!p)
		return;
	for (std::size_t i = 0; i < array_num; ++i)
		p->~T();//デストラクタ呼び出し
	operator delete[](p, allocator, IStackAllocator::DEFAULT);
	p = nullptr;//安全のためにポインタを初期化
}
#endif
```

【バッファ付スマートスタックアロケータクラス定義】

```
//--------------------
//バッファ付きスマートスタックアロケータテンプレートクラス
//※静的バッファを使用したければ、固定バッファシングルトン化する
template<std::size_t SIZE>
class CSmartStackAllocatorWithBuff : public CSmartStackAllocator
{
public:
	//定数
	static const std::size_t BUFF_SIZE = SIZE;//バッファサイズ
public:
	//コンストラクタ
	CSmartStackAllocatorWithBuff(const E_ORDERED default_ordered = NORMAL) :
		CSmartStackAllocator(m_buff, BUFF_SIZE, default_ordered)
	{}
	//デストラクタ
	~CSmartStackAllocatorWithBuff() override
	{}
private:
	//フィールド
	byte m_buff[BUFF_SIZE];//バッファ
};
```

【スマートスタックアロケータアダプタークラス】※スマートスタックアロケータ用のアダプター

```
//--------------------
//スマートスタックアロケータアダプタークラス
class CSmartStackAllocAdp : public CIStackAllocAdp
{
```

```
public:
	//型
	typedef CSmartStackAllocator ALLOCATOR_TYPE;//アロケータ型
public:
	//アクセッサ
	const char* getName() const override{ return "CSmartStackAllocAdp"; }//アロケータ名取得
	CSmartStackAllocator& getAllocator(){ return *static_cast<CSmartStackAllocator*>(&m_allocator); }
                                                                                    //アロケータ取得
	const CSmartStackAllocator& getAllocator() const { return *static_cast<CSmartStackAllocator*>(&m_allocator); }
                                                                                    //アロケータ取得
	IStackAllocator::E_ORDERED getOrdered() const { return m_ordered; }//スタック順取得
	void setOrdered(const IStackAllocator::E_ORDERED ordered){ m_ordered = ordered; }//スタック順更新
public:
	//メソッド
	//メモリ確保
	void* alloc(const std::size_t size, const std::size_t align = DEFAULT_ALIGN, const ALLOC_INFO* info = nullptr)
                                                                                    override
	{
		CSmartStackAllocator& allocator = getAllocator();
		return allocator.alloc(m_ordered, size, align);//非 virtual メンバーを使う
	}
	//メモリ解放
	void free(void* p) override
	{
		CSmartStackAllocator& allocator = getAllocator();
		return allocator.free(p);
	}
public:
	//デフォルトコンストラクタ
	CSmartStackAllocAdp() = delete;//コンストラクタ引数必須
	//コンストラクタ
	//※自動巻き戻しには対応しない
	CSmartStackAllocAdp(CSmartStackAllocator& stack,
                                        const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) :
		CIStackAllocAdp(stack, NOREWIND),//スマートスタックアロケータ
		m_ordered(ordered)//スタック順
	{}
	//デストラクタ
	~CSmartStackAllocAdp() override
	{}
private:
	//フィールド
	IStackAllocator::E_ORDERED m_ordered;//スタック順
};
```

【一時多態アロケータクラス（スマートスタックアロケータ用）定義】

```
//--------------------
//一時多態アロケータクラス：スマートスタックアロケータ用
class CTempPolySmartStackAllocator : public CTempPolyAllocatorWithStackAdp<CSmartStackAllocAdp>
{
public:
	//デフォルトコンストラクタ
	CTempPolySmartStackAllocator() = delete;//コンストラクタ引数必須
	//コンストラクタ
	CTempPolySmartStackAllocator(CSmartStackAllocator& allocator,
                                        const IStackAllocator::E_ORDERED ordered = IStackAllocator::DEFAULT) :
		CTempPolyAllocatorWithStackAdp<CSmartStackAllocAdp>(allocator, ordered)
	{}
	//デストラクタ
	~CTempPolySmartStackAllocator()
	{}
};
```

【テスト用クラス定義】

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

```
//テスト用クラス
class CTest9
{
public:
	//デフォルトコンストラクタ
	CTest9() :
		CTest9("(default)")//他のコンストラクタ呼び出し（C++11で追加された仕様）
		//m_name("(default)")//C++11が使えない場合は普通に初期化
	{
		printf("CTest9::DefaultConstructor : name=¥"%s¥"¥n", m_name);
	}
	//コンストラクタ
	CTest9(const char* name) :
		m_name(name)
	{
		printf("CTest9::Constructor : name=¥"%s¥"¥n", m_name);
	}
	//デストラクタ
	virtual ~CTest9()
	{
		printf("CTest9::Destructor : name=¥"%s¥"¥n", m_name);
	}
protected:
	//フィールド
	const char* m_name;//名前
	int m_dummy;//ダミー
};
```

【テスト（スレッド用関数）】

```
//-------------------
//スレッドテスト
#include <thread>//C++11スレッド
#include <chrono>//C++11時間
//スマートスタックアロケータ
static CSmartStackAllocatorWithBuff<1024> s_stackForThread;
void test9thread_n(const char* name)
{
	CTempPolySmartStackAllocator poly_allocator(s_stackForThread, DSA_NORMAL);
                                                        //多態アロケータをスマートスタック（正順）に設定
	CTest9* obj_p = new CTest9(name);
	std::this_thread::sleep_for(std::chrono::seconds(1));//1秒スリープ
	delete obj_p;
}
void test9thread_r(const char* name)
{
	CTempPolySmartStackAllocator poly_allocator(s_stackForThread, DSA_REVERSE);
                                                        //多態アロケータをスマートスタック（逆順）に設定
	CTest9* obj_p = new CTest9(name);
	std::this_thread::sleep_for(std::chrono::seconds(1));//1秒スリープ
	delete obj_p;
}
```

【テスト】

```
//-------------------
//テスト
void test9()
{
	{
		printf("----------スレッドテスト¥n");
		//【想定】スレッド開始前に、同じバッファを他の用途に少し使う
		//		（これは常駐するデータで破棄しないものとする）
		int* parmanent_data1 = nullptr;
		char* parmanent_data2 = nullptr;
		float* parmanent_data3 = nullptr;
		{
```

```
        CTempPolySmartStackAllocator poly_allocator(s_stackForThread, DSA_NORMAL);
                                                    //多態アロケータをスマートスタック（正順）に設定
        parmanent_data1 = new int[2];
        parmanent_data2 = new char[3];
        parmanent_data3 = new float[4];
    }
    unsigned int* parmanent_data4 = nullptr;
    unsigned char* parmanent_data5 = nullptr;
    double* parmanent_data6 = nullptr;
    {
        CTempPolySmartStackAllocator poly_allocator(s_stackForThread, DSA_REVERSE);
                                                    //多態アロケータをスマートスタック（逆順）に設定
        parmanent_data4 = new unsigned int[5];
        parmanent_data5 = new unsigned char[6];
        parmanent_data6 = new double[7];
    }
    printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
                                        s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
                                        s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
                                        s_stackForThread.getCounterN(), s_stackForThread.getCounterR());

    //カウント開始（自動リセット位置を更新）
    //ここまでの情報を保護し、以降使用するメモリは、全て解放された時点で
    //マーカーを自動的に戻す
    printf("beginN()¥n");
    s_stackForThread.beginN();//正順メモリ確保のカウントを開始
    printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
                                        s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
                                        s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
                                        s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
    printf("beginR()¥n");
    s_stackForThread.beginR();//逆順メモリ確保のカウントを開始
    printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
                                        s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
                                        s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
                                        s_stackForThread.getCounterN(), s_stackForThread.getCounterR());

    //スレッド開始①
    {
        std::thread th1 = std::thread(test9thread_n, "スレッドテスト9-1(N)");
        std::thread th2 = std::thread(test9thread_n, "スレッドテスト9-2(N)");
        std::thread th3 = std::thread(test9thread_r, "スレッドテスト9-3(R)");
        std::thread th4 = std::thread(test9thread_r, "スレッドテスト9-4(R)");
        std::this_thread::sleep_for(std::chrono::milliseconds(500));//500msec スリープ
        printf("sleep(500msec)¥n");
        printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
                                        s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
                                        s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
                                        s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
        //スレッド終了待ち
        th1.join();
        th2.join();
        th3.join();
        th4.join();
        printf("joined¥n");
        printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
                                        s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
                                        s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
                                        s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
    }

    //スレッド開始②
    {
        std::thread th5 = std::thread(test9thread_n, "スレッドテスト9-5(N)");
        std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msec スリープ
```

```
		printf("sleep(200msec)¥n");
		std::thread th6 = std::thread(test9thread_n, "スレッドテスト9-6(N)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		std::thread th7 = std::thread(test9thread_r, "スレッドテスト9-7(R)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		std::thread th8 = std::thread(test9thread_r, "スレッドテスト9-8(R)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
		                                s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
		                                s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
		                                s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
		{
		    //多態アロケータはTLSでアロケータをスレッドごとに分けて使っているので、
		    //他のスレッドが動作中に異なるアロケータを使っても問題なし
		    CSmartStackAllocatorWithBuff<128> allocator;
		    CTempPolySmartStackAllocator poly_allocator(allocator);
		    CTest9* obj_p = new CTest9("スレッドテスト(割り込み)");
		    delete obj_p;
		    printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
		                                s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
		                                s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
		                                s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
		}
		{
		    //カウント開始前に確保したメモリを途中で破棄してもカウンタには影響しない
		    //※カウント開始位置以前の位置にあるメモリは、カウンタに影響しない
		    CTempPolySmartStackAllocator poly_allocator(s_stackForThread);
		    delete[] parmanent_data1;
		    delete[] parmanent_data2;
		    delete[] parmanent_data3;
		    delete[] parmanent_data4;
		    delete[] parmanent_data5;
		    delete[] parmanent_data6;
		    printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
		                                s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
		                                s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
		                                s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
		}
		std::thread th9 = std::thread(test9thread_n, "スレッドテスト9-9(N)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		std::thread th10 = std::thread(test9thread_n, "スレッドテスト9-10(N)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		std::thread th11 = std::thread(test9thread_r, "スレッドテスト9-11(R)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		std::thread th12 = std::thread(test9thread_r, "スレッドテスト9-12(R)");
		std::this_thread::sleep_for(std::chrono::milliseconds(200));//200msecスリープ
		printf("sleep(200msec)¥n");
		printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
		                                s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
		                                s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
		                                s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
		//スレッド終了待ち
		th5.join();
		th6.join();
		th7.join();
		th8.join();
		th9.join();
		th10.join();
		th11.join();
```

```
                th12.join();
                printf("joined¥n");
                printf("marker=(%d,%d), begin=(%d,%d), counter=(%d,%d)¥n",
                                                s_stackForThread.getMarkerN(), s_stackForThread.getMarkerR(),
                                                s_stackForThread.getBeginN(), s_stackForThread.getBeginR(),
                                                s_stackForThread.getCounterN(), s_stackForThread.getCounterR());
            }
        }
}
```

↓（実行結果）

```
----------スレッドテスト
new[](size=8, poly_allocator="CSmartStackAllocAdp":1024/0/1024)   ←【想定】常駐データのためのメモリ確保：ここから
  p=0x013EB93C
new[](size=3, poly_allocator="CSmartStackAllocAdp":1024/8/1016)
  p=0x013EB944
new[](size=16, poly_allocator="CSmartStackAllocAdp":1024/11/1013)
  p=0x013EB948
new[](size=20, poly_allocator="CSmartStackAllocAdp":1024/28/996)
  p=0x013EBD28
new[](size=6, poly_allocator="CSmartStackAllocAdp":1024/48/976)
  p=0x013EBD20
new[](size=56, poly_allocator="CSmartStackAllocAdp":1024/56/968)  ←【想定】常駐データのためのメモリ確保：ここまで
  p=0x013EBCE8
marker=(28,940), begin=(0,1024), counter=(3,3)
beginN()
marker=(28,940), begin=(28,1024), counter=(0,3)                   ←正順スタックのカウンタ開始
beginR()                                                            （マーカー記憶とカウンタリセット）
marker=(28,940), begin=(28,940), counter=(0,0)                    ←逆順スタックのカウンタ開始
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/112/912)     （マーカー記憶とカウンタリセット）
  p=0x013EB958
CTest9::Constructor : name="スレッドテスト9-1(N)"
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/124/900)
  p=0x013EB964
CTest9::Constructor : name="スレッドテスト9-2(N)"
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/124/900)
  p=0x013EBCDC
CTest9::Constructor : name="スレッドテスト9-3(R)"
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/136/888)
  p=0x013EBCD0
CTest9::Constructor : name="スレッドテスト9-4(R)"
sleep(500msec)
marker=(52,916), begin=(28,940), counter=(2,2)                    ←ここまでに各スレッドで計4回メモリ確保
CTest9::Destructor : name="スレッドテスト9-1(N)"                                （現在のマーカー：正順,逆順，
delete(p=0x013EB958, poly_allocator="CSmartStackAllocAdp":1024/160/864)          開始マーカー：正順,逆順，
CTest9::Destructor : name="スレッドテスト9-3(R)"                                 現在のカウンタ:正順,逆順）
CTest9::Destructor : name="スレッドテスト9-4(R)"
delete(p=0x013EBCD0, poly_allocator="CSmartStackAllocAdp":1024/160/864)
CTest9::Destructor : name="スレッドテスト9-2(N)"
delete(p=0x013EB964, poly_allocator="CSmartStackAllocAdp":1024/160/864)
delete(p=0x013EBCDC, poly_allocator="CSmartStackAllocAdp":1024/160/864)
joined
marker=(28,940), begin=(28,940), counter=(0,0)                    ←スレッドが全て終了すると、全てのメモリが破棄され
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/112/912)     カウンタが0になったため、マーカーが開始位置に
  p=0x013EB958                                                      戻っている
CTest9::Constructor : name="スレッドテスト9-5(N)"
sleep(200msec)
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/124/900)
  p=0x013EB964
CTest9::Constructor : name="スレッドテスト9-6(N)"
sleep(200msec)
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/136/888)
  p=0x013EBCDC
CTest9::Constructor : name="スレッドテスト9-7(R)"
sleep(200msec)
```

```
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/148/876)
  p=0x013EBCD0
CTest9::Constructor : name="スレッドテスト9-8(R)"
sleep(200msec)
marker=(52,916), begin=(28,940), counter=(2,2)                    ←この時点のメモリ使用状態
                                                                    ここまでに各スレッドで計4回メモリ確保
new(size=12, poly_allocator="CSmartStackAllocAdp":128/0/128)      ←割り込みで別のアロケータを使ったメモリ操作
  p=0x004CFA04
CTest9::Constructor : name="スレッドテスト(割り込み)"
CTest9::Destructor : name="スレッドテスト(割り込み)"
delete(p=0x004CFA04, poly_allocator="CSmartStackAllocAdp":128/12/116)  ←割り込みで別のアロケータを使ったメモリ操作
marker=(52,916), begin=(28,940), counter=(2,2)                    ←この時点のメモリ使用状態(割り込みの影響なし)
delete[](p=0x013EB93C, poly_allocator="CSmartStackAllocAdp":1024/160/864)←常駐データをこのタイミングで破棄：ここから
delete[](p=0x013EB944, poly_allocator="CSmartStackAllocAdp":1024/160/864)
delete[](p=0x013EB948, poly_allocator="CSmartStackAllocAdp":1024/160/864)
delete[](p=0x013EBD28, poly_allocator="CSmartStackAllocAdp":1024/160/864)
delete[](p=0x013EBD20, poly_allocator="CSmartStackAllocAdp":1024/160/864)
delete[](p=0x013EBCE8, poly_allocator="CSmartStackAllocAdp":1024/160/864) ←常駐データをこのタイミングで破棄：ここまで
marker=(52,916), begin=(28,940), counter=(2,2)                    ←この時点のメモリ使用状態（常駐データ破棄の
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/160/864)     影響なし）
  p=0x000FB970
CTest9::Constructor : name="スレッドテスト9-9(N)"
CTest9::Destructor : name="スレッドテスト9-5(N)"
delete(p=0x000FB958, poly_allocator="CSmartStackAllocAdp":1024/172/852)
sleep(200msec)
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/172/852)
  p=0x000FB97C
CTest9::Constructor : name="スレッドテスト9-10(N)"
CTest9::Destructor : name="スレッドテスト9-6(N)"
delete(p=0x000FB964, poly_allocator="CSmartStackAllocAdp":1024/184/840)
sleep(200msec)
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/184/840)
  p=0x000FBCC4
CTest9::Constructor : name="スレッドテスト9-11(R)"
CTest9::Destructor : name="スレッドテスト9-7(R)"
delete(p=0x000FBCDC, poly_allocator="CSmartStackAllocAdp":1024/196/828)
sleep(200msec)
new(size=12, poly_allocator="CSmartStackAllocAdp":1024/196/828)
  p=0x000FBCB8
CTest9::Constructor : name="スレッドテスト9-12(R)"
CTest9::Destructor : name="スレッドテスト9-8(R)"
delete(p=0x000FBCD0, poly_allocator="CSmartStackAllocAdp":1024/208/816)
sleep(200msec)
marker=(76,892), begin=(28,940), counter=(2,2)                    ←ここまでにさらにメモリの確保と破棄が
CTest9::Destructor : name="スレッドテスト9-9(N)"                    繰り返されたため、カウンタは計4と
delete(p=0x000FB970, poly_allocator="CSmartStackAllocAdp":1024/208/816)  変わらないが、メモリの使用量は増え続けている
CTest9::Destructor : name="スレッドテスト9-10(N)"
delete(p=0x000FB97C, poly_allocator="CSmartStackAllocAdp":1024/208/816)
CTest9::Destructor : name="スレッドテスト9-11(R)"
delete(p=0x000FBCC4, poly_allocator="CSmartStackAllocAdp":1024/160/864)
CTest9::Destructor : name="スレッドテスト9-12(R)"
delete(p=0x000FBCB8, poly_allocator="CSmartStackAllocAdp":1024/160/864)
joined
marker=(28,940), begin=(28,940), counter=(0,0)  ←スレッドが全て終了すると、全てのメモリが破棄されカウンタが0に
                                                  なったため、マーカーが開始位置に戻っている
```

▼ 【応用】デバッグログ出力スレッドでの活用

スマートスタックアロケータは、前述の注意点にもある通り、安易な利用は危険でもあるので、局所的な利用にとどめたほうがよいかもしれない。

スマートスタックアロケータを活用できる具体的な処理としては、「デバッグログ出力」が考えられる。

● デバッグログ出力処理

デバッグログ出力処理は、デバッグ用のプリント文の処理のことである。プリント文はどのスレッドからでも実行されるが、その結果を逐一（コンソール等に）出力していたのでは本来の処理をブロックして処理効率を損ねてしまう。その対処として、プリントしたメッセ―ジを一旦メモリにキューイングし、専用スレッドがそれを拾ってコンソール等に出力するようにする。この時のメッセージのキューイングにスマートスタックアロケータを用いる。

● デバッグログ出力処理の性質とスマートスタックアロケータの相性

デバッグログ出力処理とスマートスタックアロケータは相性がよい。

デバッグログ出力処理（プリント文）は、それが止む暇がないほど連続し続けることはそうそうない。溜まっているメッセージを全て出力すればメモリを空にして良い。

また、メモリ限界に達したら、記録せずに捨ててしまってもかまわない。（そういう出力を行う方が悪いものとして良い）

なお、スマートスタックアロケータは、メモリを確保したスレッドと解放するスレッドが異なっていても問題にならない。

● リングバッファとの比較

本来の用途は「キュー」なので、スタックよりもリングバッファが向いている。

しかし、このケースのようにメモリをリセットできるタイミングがあるなら、リングバッファよりもずっとメモリを単純に扱うことができる。

リングバッファの場合だと、境界でループするデータを扱う必要があり、書き込みも読み込みもひと手間かかる。

ただし、データの書き込みが止まないようなケースでは、やはりリングバッファを用いたほうが良い。

リングバッファを標準的なメモリアロケータの一つとして用意するのも良い方法である。

- **デバッグロギングシステム**

別紙の「効果的なデバッグログとアサーション」にて、デバッグロギングシステムについて、詳しく説明している。

■■以上■■

■ 索引

索引項目が見つかりません。

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

以　上