

C++プログラミングII

第14回 メモリ割当てとスマートポインタ

岡本秀輔

成蹊大学理工学部

学んでほしいこと

- ▶ クラスの重要メンバ関数
 - ▶ デフォルト・コンストラクタ (ctor) 以外もある
 - ▶ オブジェクトの値操作に関する目的とタイミング
- ▶ メモリ管理
 - ▶ 変数はいつどの場所に割り当てられるか？
 - ▶ いつ変数に割り当てられた場所は解放されるか？
 - ▶ C++プログラミング I テキストの 13, 14 章を復習してください
- ▶ ヒープ領域の使い方
 - ▶ 必要な時に**割り当て**（借りて）、
 - ▶ 不要になったら**解放**（返す）

重要なメンバ関数

クラスの重要なメンバ関数

クラス T において重要な役割を果たす関数

- ▶ 必要に応じてコンパイラが自動生成する

呼び名	関数名前	引数	主な用途
通常の ctor	T	1 個以上	引数で初期化
デフォルト ctor	T	なし	デフォルト値
デストラクタ	~T	なし	破壊時後処理
コピー ctor	T	const T&	実引数
コピー代入	operator=	const T&	代入
ムーブ ctor	T	T&&	実引数
ムーブ代入	operator=	T&&	代入

- ▶ ムーブはメモリ管理を目的とするメンバ関数

デストラクタ: ~T()

デストラクタ (dtor) とは変数破壊時の後処理用の関数

- ▶ 変数の破壊時とは：
 - ▶ プログラム終了時 (大域変数)
 - ▶ 局所変数がスコープから外れた
 - ▶ 一時変数の使用が終了した
 - ▶ 明示的な破壊 (pop_back, delete)
- ▶ 主な用途：
 - ▶ メモリ解放、一時ファイル削除, 排他制御ロックの解放
 - ▶ プログラムの動作確認 (デバッグ)

```
using std::cout; // 14class1.hpp
class T {        // 割り当てられたアドレスを出力する
    int a;
public:
    T(int b):a{b}{cout <<"ctor(int):" << &a<<"\n";}
    T()         :a{1}{cout <<"ctor(default):" << &a<<"\n";}
    ~T()        {cout <<"dtor:" << &a<<"\n";}
    int get() const { return a; } // 後で使います
};
```

dtor呼び出し確認

- ▶ アドレスから ctor と dtor の対応関係を確認する

```
#include "14class1.hpp"
```

```
T g{1};
```

```
int main()
```

```
{
```

```
    T a;
```

```
    cout << "\n";
```

```
    {
```

```
        T b;
```

```
    }
```

```
    cout << "\n";
```

```
    {
```

```
        T c{2};
```

```
    }
```

```
    cout << "\n";
```

```
}
```

```
ctor(int):      0x404194
```

```
ctor(default):0x7ffca16c7c9c
```

```
ctor(default):0x7ffca16c7c98
```

```
dtor:           0x7ffca16c7c98
```

```
ctor(int):      0x7ffca16c7c94
```

```
dtor:           0x7ffca16c7c94
```

```
dtor:           0x7ffca16c7c9c
```

```
dtor:           0x404194
```

配列の場合

- ▶ 要素ごとに ctor, dtor が動作する
- ▶ 初期値指定のない要素もデフォルト ctor が動作

```
#include "14class1.hpp"
```

```
T g[2];
```

```
int main()
```

```
{  
    T a[2] {1};  
    std::cout << "\n";  
    {  
        T b[2]{1,2};  
    }  
    std::cout << "\n";  
}
```

```
ctor(default):0x404198
```

```
ctor(default):0x40419c
```

```
ctor(int):      0x7ffd5143d3f8
```

```
ctor(default):0x7ffd5143d3fc
```

```
ctor(int):      0x7ffd5143d3f0
```

```
ctor(int):      0x7ffd5143d3f4
```

```
dtor:           0x7ffd5143d3f4
```

```
dtor:           0x7ffd5143d3f0
```

```
dtor:           0x7ffd5143d3fc
```

```
dtor:           0x7ffd5143d3f8
```

```
dtor:           0x40419c
```

```
dtor:           0x404198
```

ヒープ領域の割り当てと解放

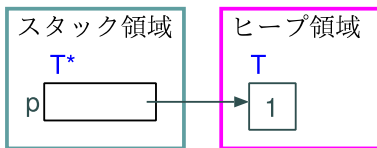
new と delete

new と delete

スコープとは無関係に無名の変数をヒープ領域に割り当てる

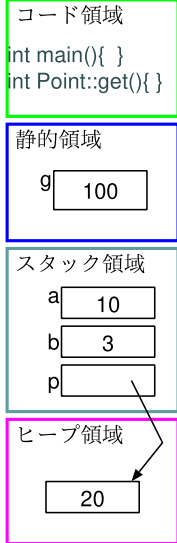
- ▶ 指定方法：
 - ▶ **new 型名**：指定した型の無名変数を割り当てる
 - ▶ **new 型名 初期値**：初期値を指定して ctor を選べる
 - ▶ **delete ポインタ変数**：割り当てた無名変数を解放する
- ▶ ポインタを使って間接的に割り当てた変数を扱う
 - ▶ 配列の引数と同じ構造
 - ▶ `vector` も同様の形式をとる

```
T* p { new T{} }; // ヒープ領域に割り当て
...              // ポインタ p を使った処理
delete p;        // 割り当て領域を解放する
```



C++で使う四つのメモリ領域 (第4回資料)

- ▶ **コード領域**：
 - ▶ 関数やメンバ関数の機械語が置かれる。
- ▶ **静的領域**：
 - ▶ 大域変数や文字列用
 - ▶ 実行開始時に割り当てられ、終了まで使用できる
- ▶ **スタック領域**：
 - ▶ 引数を含む局所変数と、関数呼び出し制御の双方に使う
 - ▶ 関数の呼び出しに割り当てが対応
- ▶ **ヒープ領域**：
 - ▶ 必要に応じて使う変数 (ただし無名)
 - ▶ 割り当てと解放は任意のタイミング (new と delete)



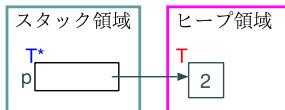
割り当てアドレスの確認

- ▶ `p` はスタック領域のポインタ変数
- ▶ `new` の割り当てで `ctor`,
`delete` の解放で `dtor` が動作

```
#include <iostream>
#include "14class1.hpp"
using std::cout;
int main() {
    T* p {nullptr};
    cout <<"&p = " << &p <<"\n";

    p = new T{2}; // 割り当て
    cout <<"p->get(): "
         << p->get() <<" , "
         <<" " << p <<"\n";

    delete p; // 領域解放
}
```



```
&p = 0x7ffdd0bdec58
ctor(int):      0x1ea92c0
p->get(): 2,    0x1ea92c0
dtor:           0x1ea92c0
```

よくある間違い

- ▶ ポインタがどこ指しているか分からない
 - ▶ ポインタは組み込み型の初期値ルールが適用される

```
T* p;                                // p の初期値は不定
cout << "&p = " << &p << "\n"; // アドレスは出力される
cout << p->get() << "\n";      // クラッシュ!
```

- ▶ 二度解放してしまう

```
T* p { new T{3} };                  // ポインタ宣言と割り当て
cout << p->get() << "\n";           // 3: 使う
delete p;                           // 解放
...                                  // 別の事をする
delete p;                           // 二度目はクラッシュ!
```

安全な使い方

- ▶ `nullptr` を初期値にし、確認してから使う

```
T* p {nullptr};           // 初期値を指定
...
if (p != nullptr)        // 確認してから使う
    cout << p->get() << "\n";
```

- ▶ 解放したら, `nullptr` を代入する

```
T* p { new T{3} };        // ポインタ宣言と割り当て
...
delete p;                 // 解放
p = nullptr;              // アドレスを消去
...                        // 別の事をする
delete p;                 // nullptr は delete 指定可
```

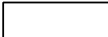
メモリリーク (leak:漏れ)

- ▶ new で割り当てた領域は delete まで解放されない
- ▶ delete を忘れるとメモリが無駄になる (メモリリーク)
 - ▶ プログラムの実行が終われば OS に回収される
 - ▶ 実行が続けば使われないままの領域となる
- ▶ dtor も動作しない

```
auto p { new T{5} };  
cout <<"p->get(): "<< p->get() <<" , "  
      <<" "<< p <<"\n";  
// delete p; // 解放し忘れる
```

```
ctor(int):      0x10a92c0  
p->get(): 5,    0x10a92c0
```

スタック領域

T*
p 

ヒープ領域

T


1 次元配列の割り当て

- ▶ `new T[n]`: 配列の割り当て
 - ▶ `new int[3] {1,2,3}`などと初期値の指定も可
- ▶ `delete[]`: `new` で割り当てた領域の解放
 - ▶ `delete[]` を忘れればメモリリーク発生
 - ▶ `delete` でもコンパイルできるが解放されない (バグ)

```
const size_t n {3};  
T* p { new T[n] }; // 配列  
  
for (size_t i=0; i<n; i++)  
    cout << p[i].get() << " ";  
cout << "\n";  
  
delete[] p; // 配列用
```

```
ctor(default):0x22eb2c8  
ctor(default):0x22eb2cc  
ctor(default):0x22eb2d0  
1 1 1  
dtor:          0x22eb2d0  
dtor:          0x22eb2cc  
dtor:          0x22eb2c8
```

多次元配列の割り当て

- ▶ 多次元配列も `new` で指定できる
 - ▶ `new T[2][3]` : 2x3 の 2 次元配列
 - ▶ `new T[2][3][4]` : 2x3x4 の 3 次元配列
 - ▶ 下記 `p` の型は `T(*p)[m]` (引数の `T p[][m]` 相当)
- ▶ `delete[]` による解放は 1 次元配列と同じ

```
const size_t n{3}, m{4};  
auto p { new T[n][m] }; // 2 次元配列  
  
for (size_t i = 0; i < n; i++)  
    for (size_t j = 0; j < m; j++)  
        cout << p[i][j].get() << " ";  
cout << "\n";  
  
delete[] p; // 配列の解放は共通形式
```


注意点

- ▶ new と delete は原始的なメモリ管理操作と考えてよい
- ▶ 過去に delete 忘れのプログラムが多数見つかったている
- ▶ delete と delete[] の間違えも多数
 - ▶ エラーが出ないので気がつかない
- ▶ delete 後にポインタでアクセスすると未定義の動作

```
p = q;  
delete q;  
p->x = 1; // クラッシュ!  
// ポインタは簡単にコピーできるので危険
```

- ▶ ctor で new をして、dtor で delete をすれば...
 - ▶ コピー ctor やコピー代入を考慮すればうまくいく
 - ▶ 長い間その方法論が議論された
 - ▶ スマートポインタの登場

スマートポインタ

スマートポインタとは

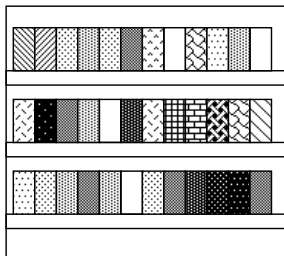
- ▶ 組み込み型のポインタのように振る舞うクラス
 - ▶ `*`, `->`, または `[]` 演算子の多重定義
 - ▶ `==`, `!=` 演算子の多重定義, `nullptr` との比較
- ▶ `delete` によるメモリ解放の自動化
 - ▶ `dtor` がうまくやってくれる

種類は二つ

- ▶ `unique_ptr`:
 - ▶ 省メモリ
 - ▶ コピー代入/コピー ctor 不可
 - ▶ 関数の引数にはリファレンスを使う
 - ▶ `std::move` による所有権の譲渡を指定 (後述)
- ▶ `shared_ptr`:
 - ▶ 管理用メモリを使う: カウンタが付加される
 - ▶ コピー代入/コピー ctor に制限なし
 - ▶ 最終使用者を監視する機能

ヒープ領域管理を本棚で考えると

- ▶ 本棚から本を取り出すことを考える
- ▶ 読み終わった本は本棚に戻してほしい
- ▶ `unique_ptr` の方法
 - ▶ 必ず一人で読み、読んだ人が片付ける原則
 - ▶ 立ち去る際には、片付けず他者に託しても良い
 - ▶ 託された者は確実に片付ける
- ▶ `shared_ptr` の方法
 - ▶ 複数人で1冊の本を回し読みしても良い
 - ▶ 途中で誰かが居なくなってもよい
 - ▶ 最後の人が片付ける（本を放置して立ち去らない）



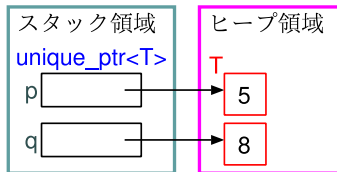
unique_ptr

unique_ptrの基本

- ▶ ヘッダファイル:<memory>
- ▶ 型名: std::unique_ptr<T> T はテンプレート引数
- ▶ メモリ割り当ての方法 (関数を使う)
 - ▶ x = std::make_unique<T>(T の ctor 引数);
 - ▶ 細かい操作が必要ならば new と ctor を組み合わせる

```
std::unique_ptr<T> p;    // p{nullptr};  
p = std::make_unique<T>(5);  
cout << p->get() << "\n"; // 5  
  
auto q {std::make_unique<T>(8)};  
cout << q->get() << "\n"; // 8
```

```
ctor(int):    0x910eb0  
5  
ctor(int):    0x9112e0  
8  
dtor:         0x9112e0  
dtor:         0x910eb0
```

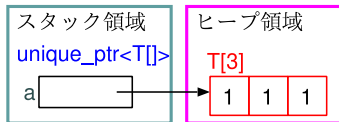


unique_ptrの基本：配列

- ▶ ヘッダファイル:<memory>
- ▶ 型名: std::unique_ptr<T[]>
- ▶ メモリ割り当ての方法 (関数を使う)
 - ▶ y = std::make_unique<T[]>(配列の要素数);
 - ▶ 多次元配列の指定は**仮引数の型宣言**と同じ形式
 - ▶ 細かい操作が必要ならば new と ctor を指定する

```
auto a {std::make_unique<T[]>(3)}; // ctor(default)
cout << a[2].get() <<"\n"; // 1
```

```
ctor(default):0x13a0eb8
ctor(default):0x13a0ebc
ctor(default):0x13a0ec0
1
dctor:         0x13a0ec0
dctor:         0x13a0ebc
dctor:         0x13a0eb8
```



コピーは不可

- ▶ `unique_ptr` はコピーが指定できない。
- ▶ コピー代入とコピー ctor が削除されている
 - ▶ 下記のエラーでも `operator=` の削除が確認できる

```
std::unique_ptr<int> p, q;  
p = std::make_unique<int>(5);  
q = p; // error
```

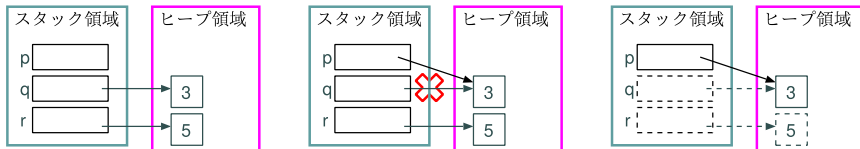
```
$ g++ -std=c++17 uniq-error.cpp  
uniq-error.cpp: 関数 ‘int main()’ 内:  
uniq-error.cpp:10:8: エラー: use of deleted function ‘std::unique_ptr  
<_Tp, _Dp>& std::unique_ptr<_Tp, _Dp>::operator=(const std::unique_ptr  
<_Tp, _Dp>&) [with _Tp = int; _Dp = std::default_delete<int>]’  
    q = p; // error  
    ^
```

```
In file included from /usr/include/c++/8/memory:80,  
                 from uniq-error.cpp:4:  
/usr/include/c++/8/bits/unique_ptr.h:398:19: 備考: ここで宣言されています  
    unique_ptr& operator=(const unique_ptr&) = delete;
```


他者に託す指定 (所有権の譲渡)

▶ std::move() 関数を使う

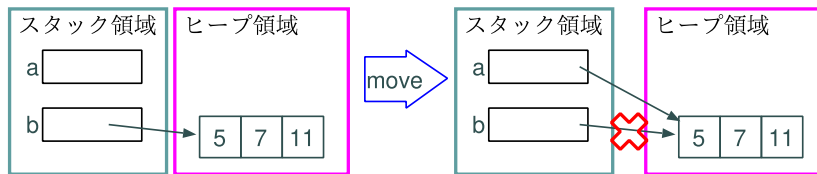
```
int main() {  
    std::unique_ptr<T> p{nullptr};  
    {  
        auto q { std::make_unique<T>(3) };  
        auto r { std::make_unique<T>(5) };  
        cout << r->get() << "\n"; // 5  
        p = std::move(q); // p に託す  
    } // r の指す領域の自動解放  
    if (p != nullptr)  
        cout << p->get() << "\n"; // 3  
} // p の指す領域の自動解放
```



std::move の役割

std::move は値を相手に託す目的で指定する

- ▶ 所有権の譲渡とも言う
 - ▶ `a=std::move(b)` 後の `b` の値は再設定後まで使用不可
 - ▶ 実際には引数の型を変換し呼ばれる関数を変更するだけ
 - ▶ `std::move()` は代入や実引数との組み合わせが必要
- ▶ 通常の代入や実引数
 - ▶ `a = b` で `a` のコピー代入のメンバ関数
 - ▶ `a(b)` で `a` のコピー ctor
- ▶ ムーブ指定の代入と実引数
 - ▶ `a = std::move(b)` で `a` のムーブ代入のメンバ関数
 - ▶ `a(std::move(b))` で `a` のムーブ ctor

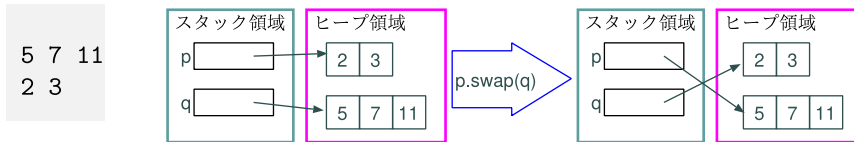


swap() メンバ関数

▶ お互いの指している先を交換

(第 4 回の MyVec を確認)

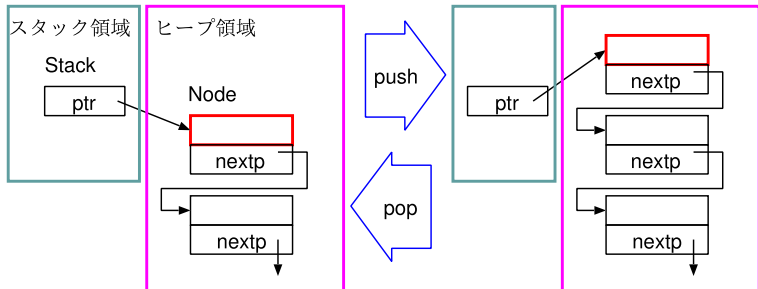
```
auto p { std::make_unique<int[]>(2) };  
p[0] = 2; p[1] = 3;  
auto q { std::make_unique<int[]>(3) };  
q[0] = 5; q[1] = 7; q[2] = 11;  
p.swap(q);  
std::cout  
    << p[0] << " " << p[1] << " " << p[2] << "\n"  
    << q[0] << " " << q[1] << "\n";
```



スタック データ構造の実装

スタックとは

- ▶ LIFO:後入れ先出しのデータ構造
- ▶ 操作: push, pop, top
- ▶ 実装方法は配列でもリストでも良い
- ▶ 今回作る構造: ptr がスタックの**トップ**を指す
 - ▶ Node 構造体をヒープ領域に割り当ててつないでいく
 - ▶ Stack クラスの ptr 変数で Node 構造体のつながりを管理
 - ▶ ~Node で解放のタイミングを確認



想定する使い方

- ▶ `std::stack` と同じ使い方を考える
- ▶ `unique_ptr` 版を作成する

```
#include <iostream>
#include <memory>
//
// クラスの定義を作っていく
//
int main() {
    Stack s;      // std::stack<int> s; に置き換えられる
    s.push(1);
    s.push(2);
    s.push(3);
    std::cout << s.top() << "\n"; // 3
    s.pop();
    std::cout << s.top() << "\n"; // 2
}
```

unique_ptr 版スタック

unique_ptr版スタックの実装

- ▶ using で長い型名を省略する

```
using Ptr = std::unique_ptr<struct Node>;

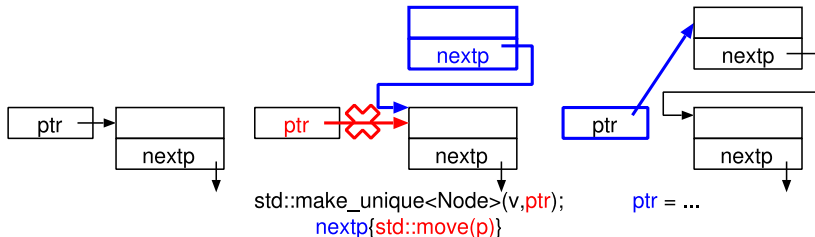
struct Node {
    int value;
    Ptr nextp;
    Node(int a, Ptr& p)
        :value{a}, nextp{std::move(p)} {}
    ~Node() { std::cout <<"dtor: " << value <<"\n"; }
};

class Stack {
    Ptr ptr;
public:
    void push(int v){
        ptr = std::make_unique<Node>(v,ptr); }
    void pop() { ptr = std::move(ptr->nextp); }
    int top() const { return ptr->value; }
};
```


push メンバ関数

- ▶ ctor にリファレンスで ptr を渡し（コピーではない）,
- ▶ メンバ初期化リストでムーブして nextp に託す

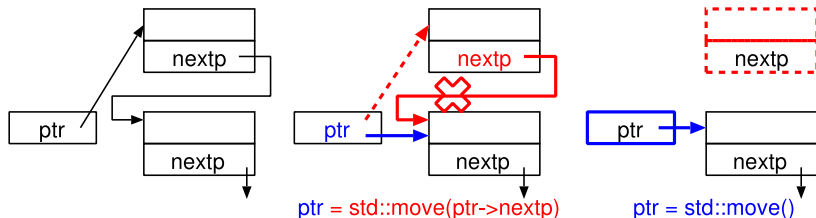
```
struct Node {  
    ...  
    Node(int a, Ptr& p)  
        :value{a}, nextp{std::move(p)} {}  
};  
class Stack {  
    void push(int v){  
        ptr = std::make_unique<Node>(v,ptr); }  
}
```



pop メンバ関数

- ▶ 先頭の構造体の nextp の値を ptr に託す
- ▶ ptr が元々指していた領域は自動的に解放される

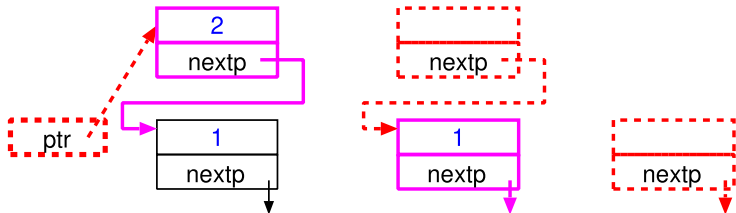
```
class Stack {  
    ...  
    void pop() { ptr = std::move(ptr->nextp); }  
};
```



デストラクタと実行結果

- ▶ 指定は不要
- ▶ Stack の破壊時に自動解放が連鎖して発生
- ▶ ptr のデストラクタ動作し
 - ▶ 先頭が解放され、
 - ▶ 連動して 2 番目が解放され、
 - ▶ 連動して 3 番目が解放され、...

```
3  
dtor: 3  
2  
dtor: 2  
1  
dtor: 1
```



まとめ

自分の言葉でまとめましょう。

- ▶ クラスの重要なメンバ関数
- ▶ `new` と `delete` によるメモリ管理
- ▶ `std::unique_ptr` による解放の自動化
- ▶ スタック データ構造の作成方法の一つ

補足

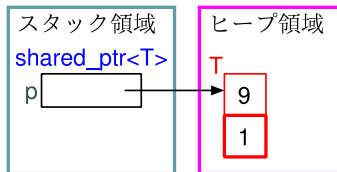
shared_ptr

shared_ptrの基本

- ▶ ヘッダファイル:<memory>
- ▶ 型名: std::shared_ptr<T>
- ▶ メモリ割り当ての方法 (関数を使う)
 - ▶ x = std::make_shared<T>(T の ctor 引数);
- ▶ アドレス取得可能, カウンタが付随

```
std::shared_ptr<T> p;    // p {nullptr};  
p = std::make_shared<T>(9);  
cout <<"p->get(): " << p->get() <<" , " // 9  
      <<" " << p <<"\n";
```

```
ctor(int):      0x12ddec0  
p->get(): 9,    0x12ddec0  
dtor:          0x12ddec0
```

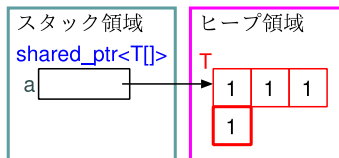


shared_ptr の基本 (配列)

- ▶ ヘッダファイル:<memory>
- ▶ 型名: std::shared_ptr<T[]>
- ▶ メモリ割り当ての方法 (関数が**使えない**)
 - ▶ `y = std::shared_ptr<T[]>{ new T[N] };`
 - ▶ 多次元配列は引数の型と同じ形式
 - ▶ `std::make_shared<T[]>()` 配列版は c++20 から

```
auto a {std::shared_ptr<T[]>{ new T[3] }};  
cout << a[2].get() << "\n"; // 1
```

```
ctor(default):0x1bcdeb8  
ctor(default):0x1bcdeb8  
ctor(default):0x1bcdec0  
1  
dctor:         0x1bcdec0  
dctor:         0x1bcdeb8  
dctor:         0x1bcdeb8
```

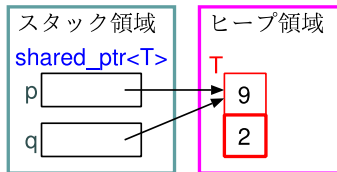


コピーの確認

- ▶ use_count() でコピー数が分かる

```
std::shared_ptr<T> p {nullptr};  
cout << p.use_count() << "\n"; // 0  
{  
    auto q { std::make_shared<T>(9) };  
    cout << q.use_count() << "\n"; // 1  
    p = q; // コピー OK  
    cout << q.use_count() << "\n"; // 2  
}  
if (p!=nullptr)  
    cout << p.use_count() << "\n"; // 1
```

```
0  
ctor(int):    0xe8a2d0  
1  
2  
1  
dtor:        0xe8a2d0
```



生ポインタ版スタック

生ポインタ版の概要

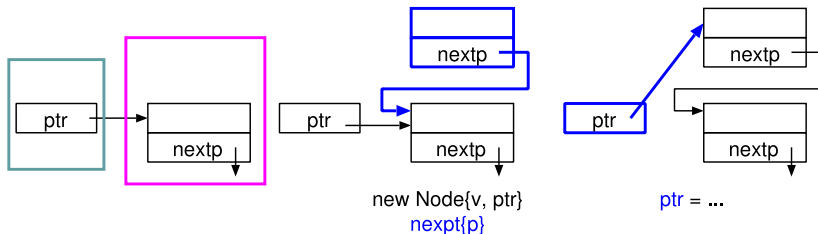
- ▶ Node をヒープ領域に割り当ててつないでいく
- ▶ ~Node で解放のタイミングを確認
- ▶ Stack 変数を使って Node のつながりを管理

```
struct Node {  
    int    value;  
    Node* nextp;  
    Node(int a, Node* p):value{a},nextp{p}{}  
    ~Node() { std::cout <<"dtor: " << value <<"\n"; }  
};  
  
class Stack {  
    Node* ptr {nullptr};  
public:  
    void push(int v);  
    void pop();  
    ~Stack();  
    int top() const { return ptr->value; }  
};
```

push メンバ関数

- ▶ ctor で nextp を設定し、その後に ptr を更新

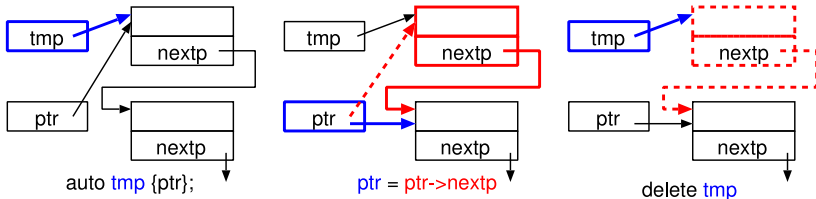
```
struct Node {  
    Node(int a, Node* p):value{a},nextp{p}{}  
    ...  
};  
class Stack {  
    Node* ptr {nullptr};  
    ...  
    void push(int v) { ptr = new Node{v,ptr}; }  
};
```



pop メンバ関数

- ▶ tmp に後で delete をする領域を覚えておく

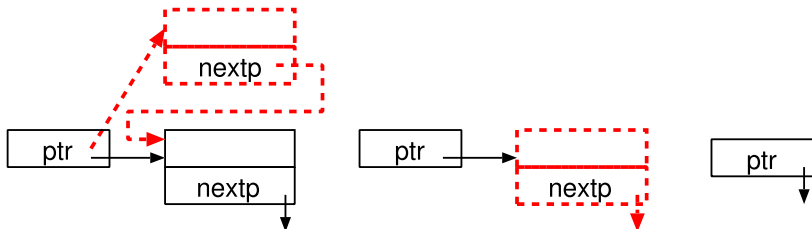
```
class Stack {  
    ...  
    void pop() {  
        auto tmp { ptr }; // 解放する対象を覚えておく  
        ptr = ptr->nextp; // 先頭を変更する  
        delete tmp;      // 解放  
    }  
    ...  
};
```



デストラクタ

- ▶ 先頭から順番に pop していく
- ▶ この処理を作り忘れることが多い

```
class Stack {  
    ...  
    ~Stack() {  
        while (ptr != nullptr) // empty() が欲しくなる  
            pop();  
    }  
    ...  
};
```



shared_ptr 版スタック

shared_ptr 版スタックの実装

▶ std::move もなくなる

```
using Ptr = std::shared_ptr<struct Node>;

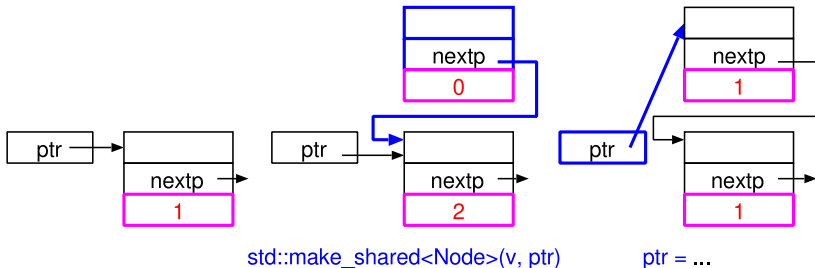
struct Node {
    int value;
    Ptr nextp;
    Node(int a, Ptr& p):value{a},nextp{p}{}
    ~Node() { std::cout <<"dtor: " << value <<"\n"; }
};

class Stack {
    Ptr ptr;
public:
    void push(int v) {
        ptr = std::make_shared<Node>(v, ptr); }
    void pop() { ptr = ptr->nextp; }
    int top() const { return ptr->value; }
};
```


push メンバ関数

- ▶ カウンタは入ってくる矢印の数に対応する
- ▶ コピー ctor とコピー代入がカウンタに関係する

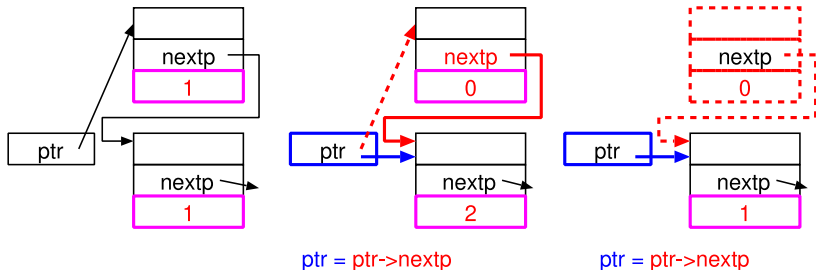
```
struct Node {  
    Node(int a, Ptr& p):value{a},nextp{p}{}  
};  
class Stack {  
    void push(int v) {  
        ptr = std::make_shared<Node>(v, ptr);  
    }  
}
```



pop メンバ関数

- ▶ コピー時にカウンタが0になったら自動解放
- ▶ ptr が元々指していた領域のカウンタが0になっている

```
class Stack {  
    ...  
    void pop() { ptr = ptr->nextp; }  
    ...  
};
```



デストラクタと実行結果

- ▶ 指定は不要
- ▶ Stack の破壊時に自動解放が連鎖発生
- ▶ ptr のデストラクタ動作し
 - ▶ 先頭のカウンタが 0 になり解放され、
 - ▶ 連動して 2 番目が解放され、
 - ▶ 連動して 3 番目が解放され、 ...
- ▶ この例ではカウンタが通常 1 なので unique_ptr で十分

```
3  
dtor: 3  
2  
dtor: 2  
1  
dtor: 1
```

