スタック

C++プログラミングIII

本日お話する内容

1. スタックの概要
2. スタックの応用
3. スタックの作成

スタック（stack）とは

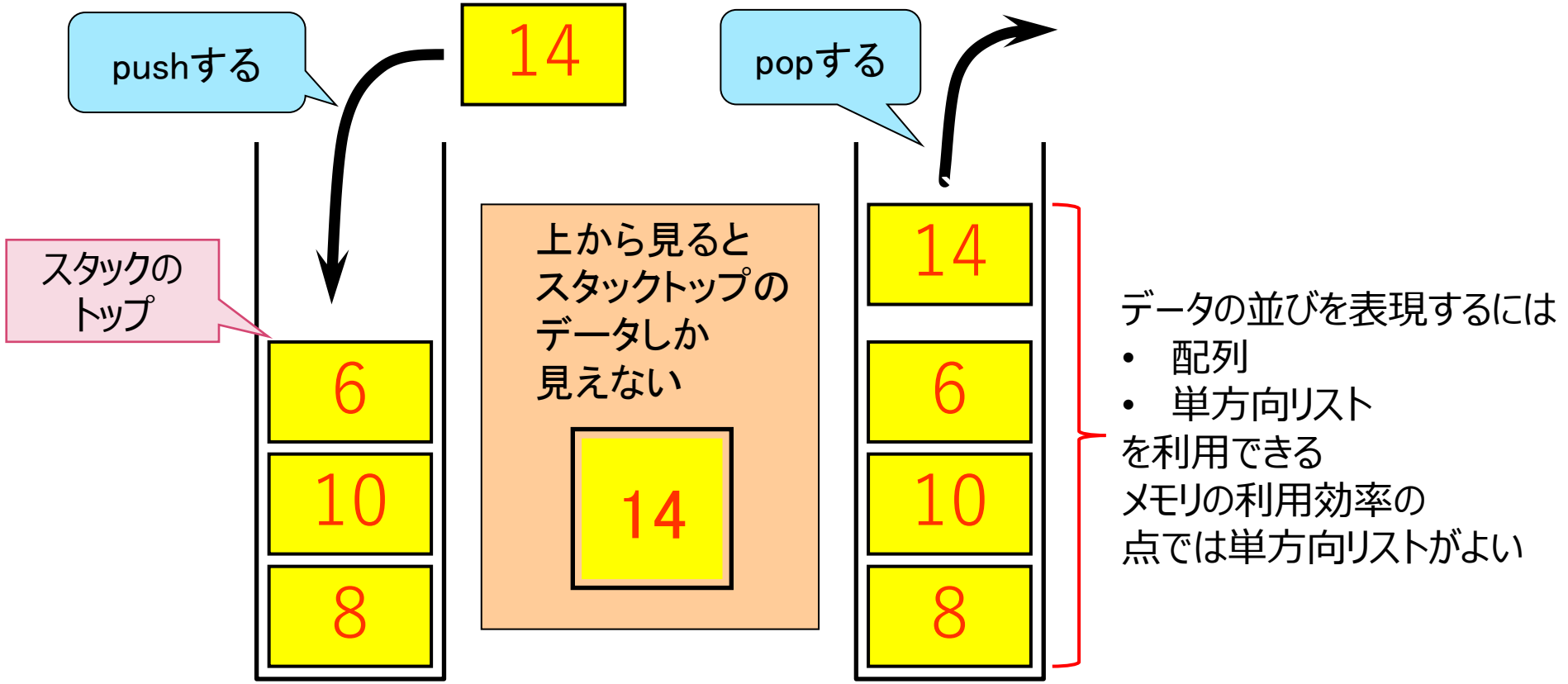
- stack：「積み重ね」
  - スタックの特徴：
    「後入れ，先出し」
    “Last In First Out”： LIFO

見ることができるのは
一番上（top）のデータのみ

- 参考： キュー（queue）
  - 「先入れ，後出し」
  - “First In First Out”： FIFO

スタックに対する操作

- push　データを積む
- pop　データを取り出す

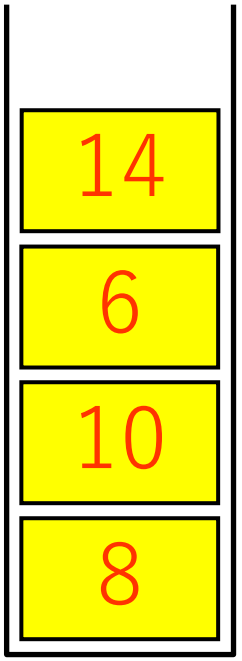


入力データを逆順に出力するプログラム

実行例

```
整数--> 8
整数--> 10
整数--> 6
整数--> 14
整数-->  （Ctrl+D）

スタックに4個のデータがあります.
top -> 14 -> 6 -> 10 -> 8
```



入力データを逆順に出力するプログラム

```
#include <iostream>
#include <stack>
using namespace std;

int main()
{
    stack<int> st1;  // 空のスタックを作る
    int x;           // 入力する整数

    while (cout << "整数--> " && cin >> x)
        st1.push(x);
    cout << "¥n";
    cout << "スタックに" << st1.size() << "個のデータがあります. " << "¥n";
    cout << "top";
    while (!st1.empty()) {                  // スタックが空でない限り以下を繰り返す
        cout << " -> " << st1.top();        // スタックトップのデータを表示
        st1.pop();                          // スタックトップのデータを削除
    }
    cout << "¥n";

    return 0;
}
```

標準ライブラリのstackを利用したプログラム例
stack<int> : int型のデータを扱うスタック
push(x) : xを値に持つノードをpushする
size() : スタックの要素数を返す
empty() : スタックの要素数が0のときtrueを返す
top() : スタックトップの要素の値を返す
pop() : スタックをpopする

本日お話する内容

1. スタックの概要
2. スタックの応用
3. スタックの作成

逆ポーランド記法の数式の計算①

通常の数式	{100−(30+12)*2+5*4}/2

演算子の優先順位を考慮し,
さらに最も内側の括弧内から計算を行う.

逆ポーランド記法の数式	100 30 12 + 2 * − 5 4 * + 2 /

括弧や演算子の優先順位を考える必要がない.
式の先頭から見ていき,
演算子が出てきたらその直前の2つの値に演算を施し,
結果を置き換える.
式の最後まで見終わるときには値がひとつだけ残ることになる.

逆ポーランド記法の数式の計算②

100 30 12 + 2 ＊ − 5 4 ＊ + 2 ／ を計算する
（通常の数式では{100−(30+12)*2+5*4}/2）

1̶3̶6̶ 5̶2̶0̶8̶4̶ 4 / + 2̶1̶*̶2 * +

逆ポーランド記法の数式の計算③

```
#include <iostream>
#include <stack>
using namespace std;

// 先に取り出したほうをbに，後に取り出したほうをaに詰める
void get2numberfromstack(stack<int>& st, int& a, int& b) {
    if (st.empty()) {
        cerr << "エラー：スタックにデータがない¥n";
        exit(EXIT_FAILURE);
    }
    b = st.top();
    st.pop();
    if (st.empty()) {
        cerr << "エラー：スタックにデータがない¥n";
        exit(EXIT_FAILURE);
    }
    a = st.top();
    st.pop();
}
```

逆ポーランド記法の数式の計算④

```
int main() {
    stack<int> st;    // 空のスタックを作る
    string input[200];
    size_t count{0};
    int number, a, b;
    cout << "逆ポーランド記法の数式を入力してください--> ";
    while (cin >> input[count]) {
        if (input[count] == "$") break;
        count++;
    }
    for (size_t i{ 0 }; i < count; i++) {
        if (input[i] == "+") { get2numberfromstack(st, a, b); st.push(a + b); }
        else if (input[i] == "-") { get2numberfromstack(st, a, b); st.push(a - b); }
        else if (input[i] == "*") { get2numberfromstack(st, a, b); st.push(a * b); }
        else if (input[i] == "/") { get2numberfromstack(st, a, b); st.push(a / b); }
        else { number = atof(input[i].c_str()); st.push(number); }
    }
    if (st.size() != 1) {
        cerr << "エラー：スタックにデータが2つ以上ある，もしくは全くない¥n";
        exit(EXIT_FAILURE);
    }
    a = st.top();
    cout << a << "¥n";
    return 0;
}
```

逆ポーランド記法の数式の計算⑤

実行例（プログラム名がrpnの場合）

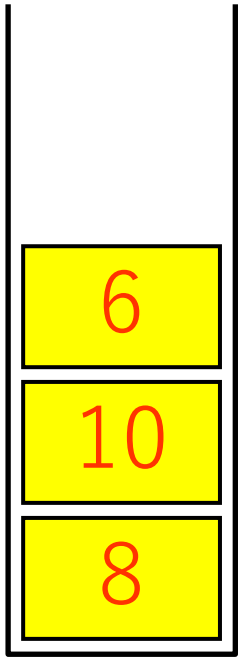
```
% ./rpn
逆ポーランド記法の数式を入力してください--> 100 30 12 + 2 * - 5 4 * + 2 / $
18
% ./rpn
逆ポーランド記法の数式を入力してください--> 50 10 - 20 - $
20
% ./rpn
逆ポーランド記法の数式を入力してください--> 50 10 20 - - $
60
%
```

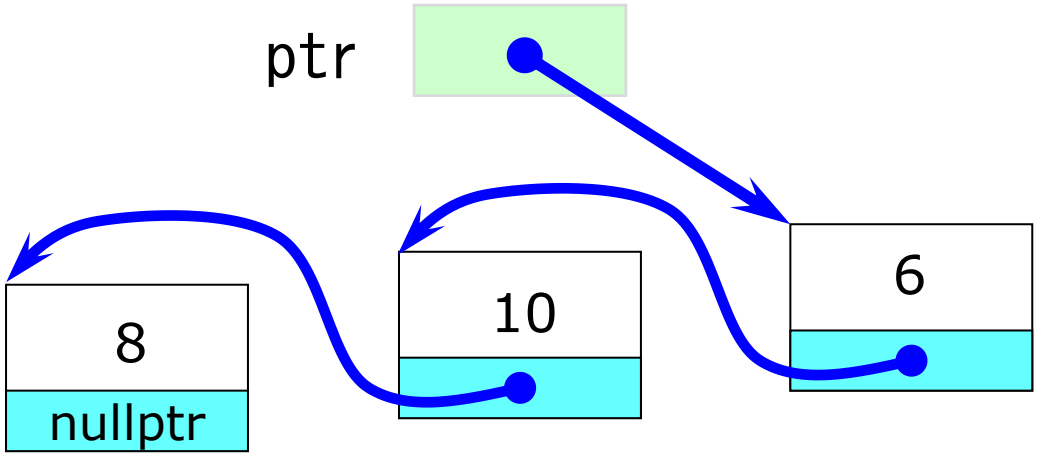
本日お話する内容

1. スタックの概要
2. スタックの応用
3. スタックの作成

スタックを表現するデータ構造

- スタックのデータの並びを表現するには
  配列や単方向リストなどが利用できる
- メモリの利用効率の点では単方向リストが良い

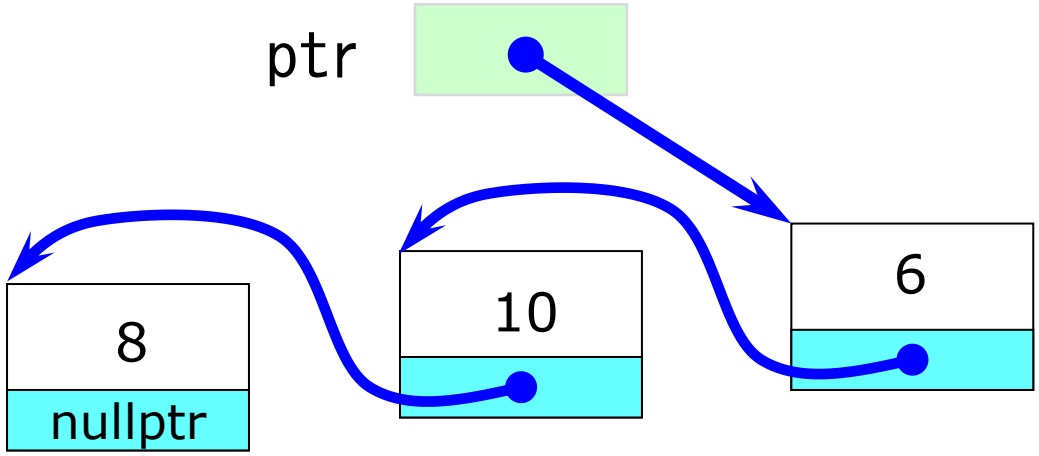




単方向リスト

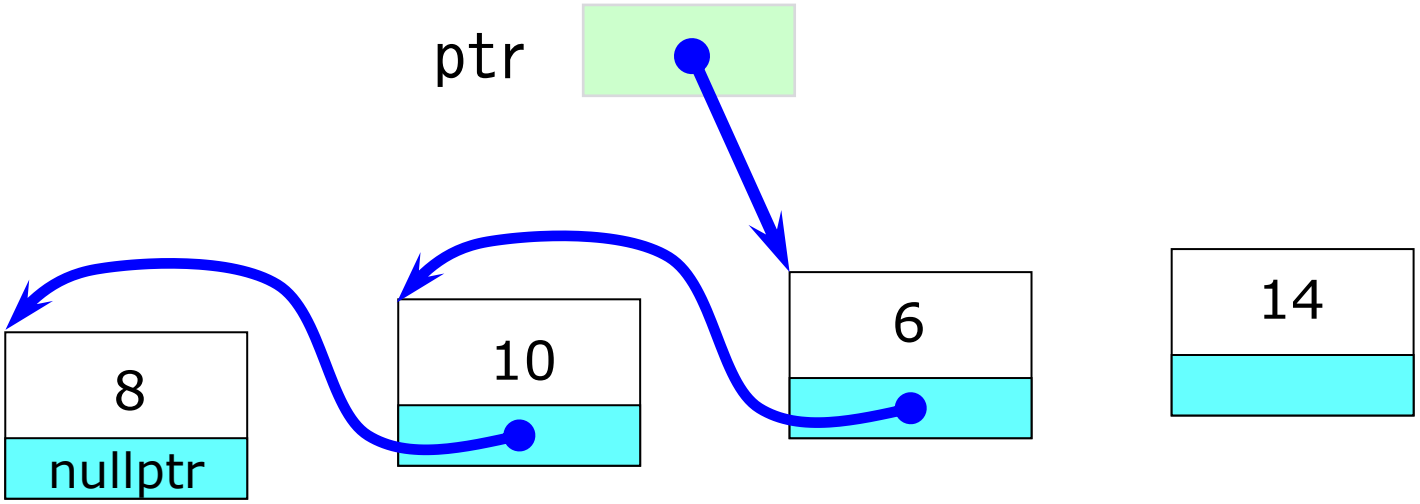
単方向リスト

- 一連のノードが，データの値と次のデータへのリンクを持つデータ構造
- 各データにはノードを先頭から順に辿ることでアクセスする
- データはリンクの方向にしか辿れない



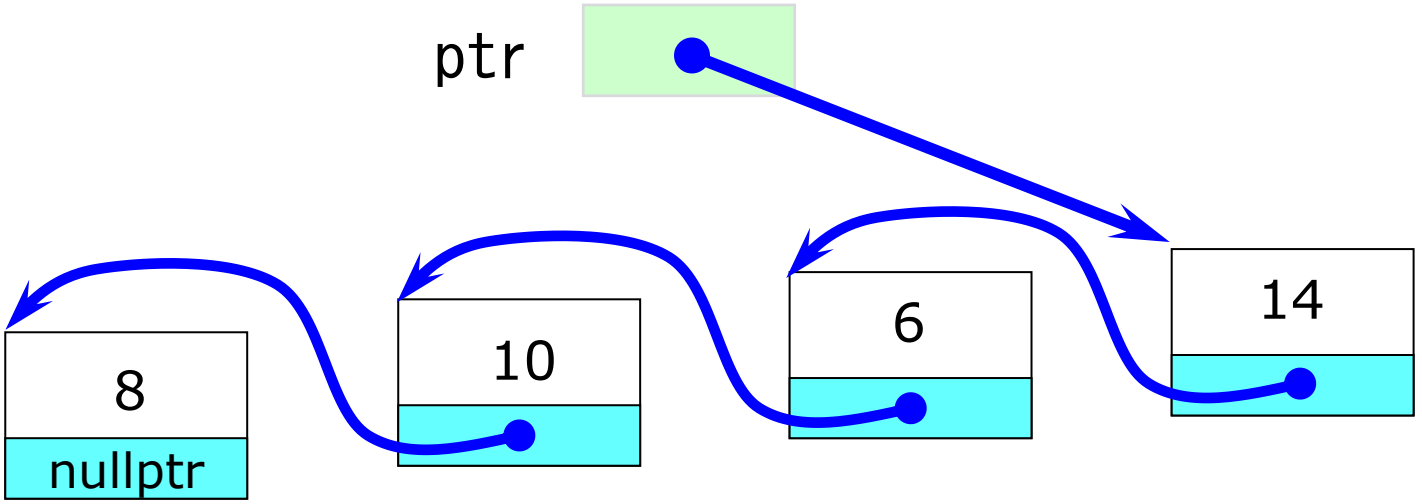
単方向リストに対する操作①

- 先頭にデータを挿入する
  1. 挿入するオブジェクトを作成する



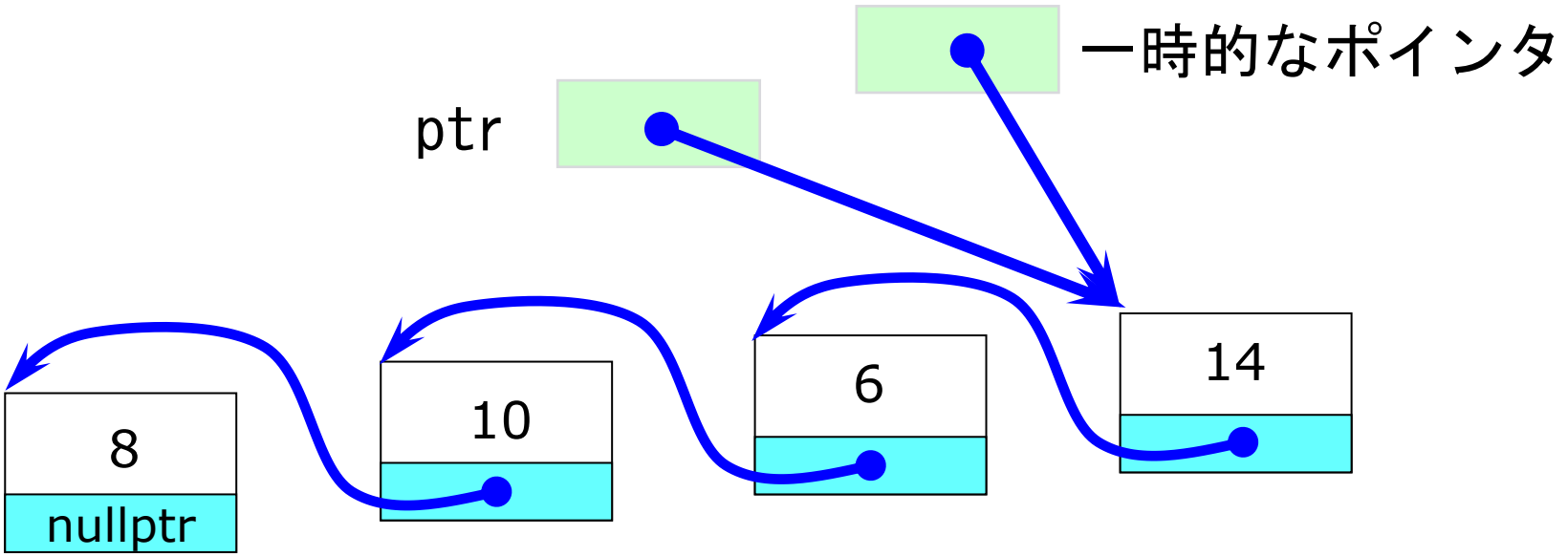
単方向リストに対する操作②

- 先頭にデータを挿入する
  1. 挿入するオブジェクトを作成する
  2. 挿入するオブジェクトのnextがそれまでのptrを，ptrが挿入するオブジェクトを指すようにする



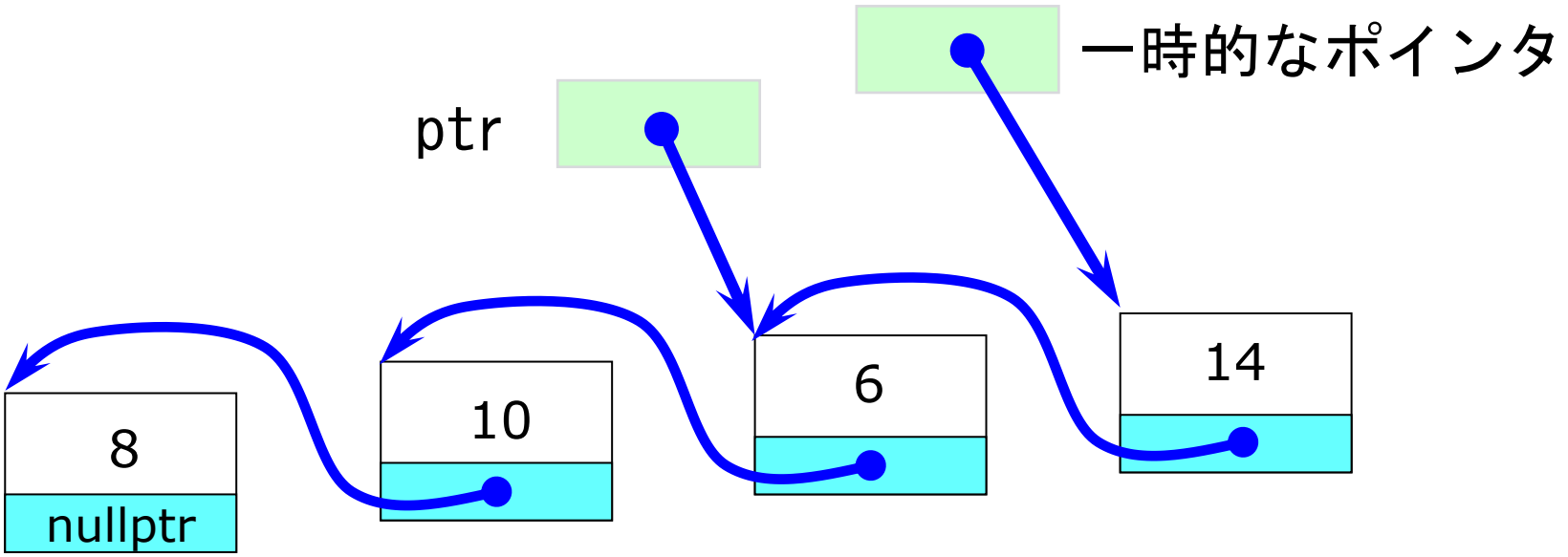
単方向リストに対する操作③

- 先頭からデータを削除する
  1．ptrを一時的なポインタに代入する



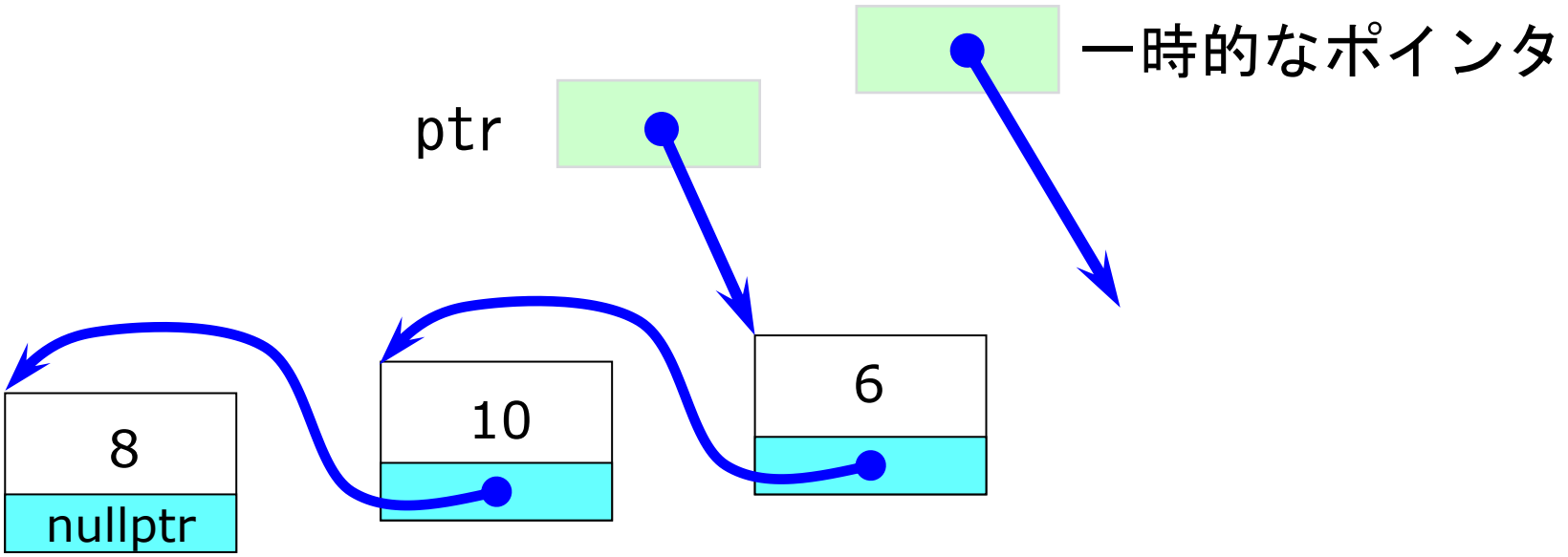
単方向リストに対する操作④

- 先頭からデータを削除する
  1. ptrを一時的なポインタに代入する
  2. ptrが先頭のオブジェクトのnextを指すようにする



単方向リストに対する操作⑤

- 先頭からデータを削除する
  1. ptrを一時的なポインタに代入する
  2. ptrが先頭のオブジェクトのnextを指すようにする
  3. 一時的なポインタを使ってオブジェクトを削除する



スタックの設計①

整数データを扱う単方向リストのノード

```
class Node {
    int value; Node* nextp;
public:
    Node(int a, Node* p = nullptr) :value{ a }, nextp{ p } {}
    int getData() const { return value; }
    Node* getNext() const { return nextp; }
    void setNext(Node* pNext) { nextp = pNext; }
};
```

スタックの設計②

単方向リストを用いたスタック

```
class Stack {
    Node* ptr{ nullptr }; // スタックトップ（単方向リストの先頭）
public:
    ~Stack() { while (!empty()) pop(); }    // デストラクタ
    void push(int);                         // 値xを持つノードをスタックトップに追加
    void pop();                             // スタックトップのノードを削除する
    int top() const;                        // スタックトップのノードの値を返す
    bool empty() const { return ptr == nullptr; }   // スタックが空かの判定
    size_t size() const;                    // スタックの要素数を返す
};
```

上記のpush (), top(), pop (), size()を
Nodeクラスを利用してどのように作るか？

スタックの設計③

単方向リストを用いたスタック

```
// 値xを持つノードをスタックトップに追加する
void Stack::push(int v) {

}

// スタックトップのノードの値を返す
int Stack::top() const {

}

// スタックトップのノードを削除し，スタックトップのnextをスタックトップにする
void Stack::pop() {

}

// スタックの要素数を返す
size_t Stack::size() const {

}
```

スタック設計のポイント

- スタックに出し入れするデータのデータ構造クラスを定義する
  （今回の例ではNodeクラス）
  - 格納したいデータのメンバに加えて，単方向リスト用のポインタを準備する
  - 各データメンバへのアクセス用関数を作成する

- スタック自体のクラスを定義する
  （今回の例ではStackクラス）
  - スタック用の基本的な関数push()，top()，pop()を，
    スタックに出し入れするデータのデータ構造を用いて作成する
  - スタックの情報を取得する関数empty()，size()を作成する

本日はここまでです

お疲れさまでした