

参照とポインタ

Game Programming B #03

向井 智彦

先週のおさらい

- クラス, メンバ変数, メンバ関数
- アクセス指定子とカプセル化
- オーバーロードとオーバーライド
- 継承, 基底クラスと派生クラス
- 仮想関数と純粋仮想関数
- 抽象クラスとその継承

Game Programming A のおさらい

値渡しと参照渡し

値渡しとは？

- 変数が持つデータの内容を, 別の変数にコピーして渡す操作
 - コピー先で値が書き換えられても, コピー元には影響を及ぼさない
 - 代入演算「=」の挙動
 - 関数の引数の挙動
 - 関数の戻り値の挙動

値渡しの動作確認 (1/2)

```
int main()
{
    int x = 10;
    int y = x;
    x = 5;
    cout << x << ", " << y << endl; // 出力？
}
```

値渡しの動作確認 (2/2)

```
void ZeroClear(int a)
{
    a = 0;
}
int main()
{
    int x = 10;
    ZeroClear(x);
    cout << x << endl; // どんな出力結果？
}
```

参照とは？

- 同一データ/同一変数に別名を与える処理

```
int main()
{
    int x = 10;
    int &y = x; // int型変数への参照
    x = 5;
    cout << x << ", " << y << endl;
    y = 8;
    cout << x << ", " << y << endl;
}
```

参照渡しとは？

- 参照を通じた関数への引数渡し

```
void ZeroClear(int &a)
{
    a = 0;
}
int main()
{
    int x = 10;
    ZeroClear(x);
    cout << x << endl; // どんな出力結果？
}
```


戻り値の代替としての参照渡し

- 関数の出力を受け取るための参照引数

```
void Double(int &output1, int input)
{
    output = input * 2;
}
int main()
{
    int x = 10;
    int y = 0;
    Double(y, x);
    cout << x << ", " << y << endl;
}
```

クラスの参照渡し

```
void Double(Vector3 &v)
{
    double x2 = 2.0 * v.GetX();
    double y2 = 2.0 * v.GetY();
    double z2 = 2.0 * v.GetZ();
    v.Set(x2, y2, z2);
}

Double(vec);
```

クラスメンバ関数への参照渡し

```
class Vector3
{
public:
    void CopyTo(Vector3 &v) {
        v.Set(x, y, z);
    }
    ...
};
```

```
Vector3 a(1.0, 1.0, 1.0);
Vector3 b;
a.CopyTo(b);
```

参照の特徴

- 変数のように後から上書きできない

```
int main()
{
    int x = 10;
    int &y = x;  //yはxの別名
    int z = 0;
    x = 5;
    printf("%d, %d\n", x, y);
    y = z;  //値の代入(≠参照先の変更)
    cout << x << ", " << y << endl;
}
```

参照を使うケース

- 引数として渡したデータの内容を関数側で上書きするとき
 - 複数の戻り値→複数の参照渡し
- 巨大なクラスを関数の引数とするとき
 - 値渡しするとコピー/クローンの計算時間が増大
 - 参照渡しだと「別名」を作る処理のみ

Game Programming A のおさらい & 新しい内容

ポインタ

ポインタ

- 別名の付け方 その2

```
int main()
{
    int x = 10;
    int *y = &x;    //int型変数へのポインタ
    x = 5;
    cout << x << ", " << *y << endl;
    *y = 8;
    cout << x << ", " << *y << endl;
}
```

ポインタを通じた参照渡し

- 参照を通じた関数への引数渡し

```
void ZeroClear(int *a)
{
    *a = 0;
}
int main()
{
    int x = 10;
    ZeroClear(&x);
    cout << x << endl;    // どんな出力?
}
```


戻り値の代替としてのポインタ参照渡し

- 関数の出力を受け取るための参照引数

```
void Double(int *output, int input)
{
    *output = input * 2;
}
int main()
{
    int x = 10;
    int y = 0;
    Double(&y, x);
    cout << x << ", " << y << endl;
}
```

クラスのポインタ渡し

```
void Double(Vector3 *v)
{
    double x2 = 2.0 * v->GetX();
    double y2 = 2.0 * v->GetY();
    double z2 = 2.0 * v->GetZ();
    v->Set(x2, y2, z2);
}
```

v->~~~ の部分は (*v).~~~でもOK

クラスメンバ関数へのポインタ渡し

```
class Vector3
{
public:
    void CopyTo(Vector3 *v) {
        v->Set(x, y, z);
    }
    ...
};
```

```
Vector3 a(1.0, 1.0, 1.0);
Vector3 b;
a.CopyTo(&b);
```

ポインタ変数の特徴

- 通常の変数のように後から上書きできる

```
int main()
{
    int x = 10;
    int *y = &x; //yはxの別名
    int z = 0;
    *y = 5;
    cout << *y << ", " << z << endl;
    y = &z; //参照先の変更
    *y = 7;
    cout << *y << ", " << z << endl;
}
```

参照渡し vs ポインタ渡し

- 多くの場合、参照渡しで十分
- 少し凝ったデータ構造やアルゴリズムを使う場合(本日以降)はポインタが必須

```
void Double(Vector3 &v)
{
    double x2 = 2.0 * v.GetX();
    double y2 = 2.0 * v.GetY();
    double z2 = 2.0 * v.GetZ();
    v.Set(x2, y2, z2);
}
```

```
void Double(Vector3 *v)
{
    double x2 = 2.0 * v->GetX();
    double y2 = 2.0 * v->GetY();
    double z2 = 2.0 * v->GetZ();
    v->Set(x2, y2, z2);
}
```

ポインタを使うケース

- ライブラリ/APIがポインタ使用を想定する場合
- 実行中に参照先を変更する必要がある場合
- プログラム実行中にクラスインスタンスを作る必要があるとき
 - プログラミング中に何個のインスタンスを用意すべきか(≡配列の長さが)わからない場合
 - new/new[] & delete/delete[]

ポインタの正体

- オブジェクトのメモリ位置 (アドレス)
 - `int *p = &a;` は, `int`型変数 `a` のメモリ位置
 - 配列 `int a[];` の `a` は, 先頭要素へのポインタ
- 参照渡し&ポインタ渡し
 - オブジェクトのメモリ位置情報を渡し, その位置が指す変数を直接読み書き
 - 一方, 値渡しは変数の内容をコピー/クローン

ポインタの正体

```
int main()
{
    int value = 0;
    int *ptr = &value;
    int array[5] = {0, 1, 2, 3, 4};
    int *arrayptr = array;
    char str1[6] = "hello";
    char *str2 = str1;
}
```


コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```

アドレス

4GB RAM搭載機は $(4 \times 1024 \times 1024 \times 1024)$ 番地まである

0番 1番 2番 3番 4番 5番 6番 7番 8番 9番 ...

RAM

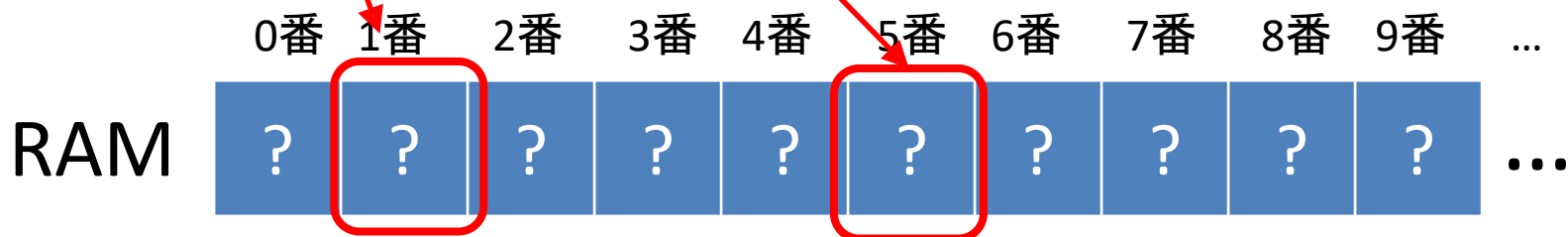


変数をどこかの番地に割り当て、その内容＝データ値を操作

コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```



変数aに割り当て

変数bに割り当て

コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

```
int a, b;
```

```
a = 0; b = 10;
```

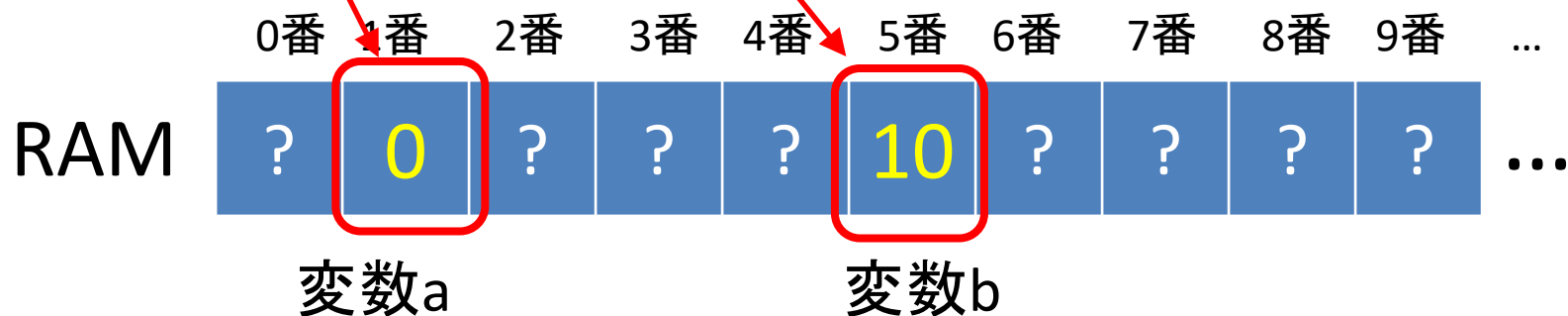
```
int *p;
```

```
p = &a;
```

```
*p = 20;
```

```
p = &b;
```

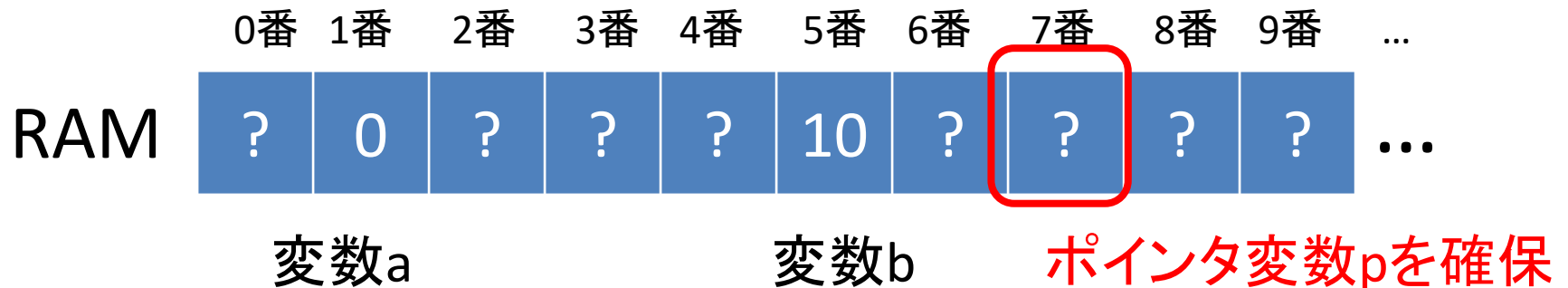
```
*p = 30;
```



コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```



コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

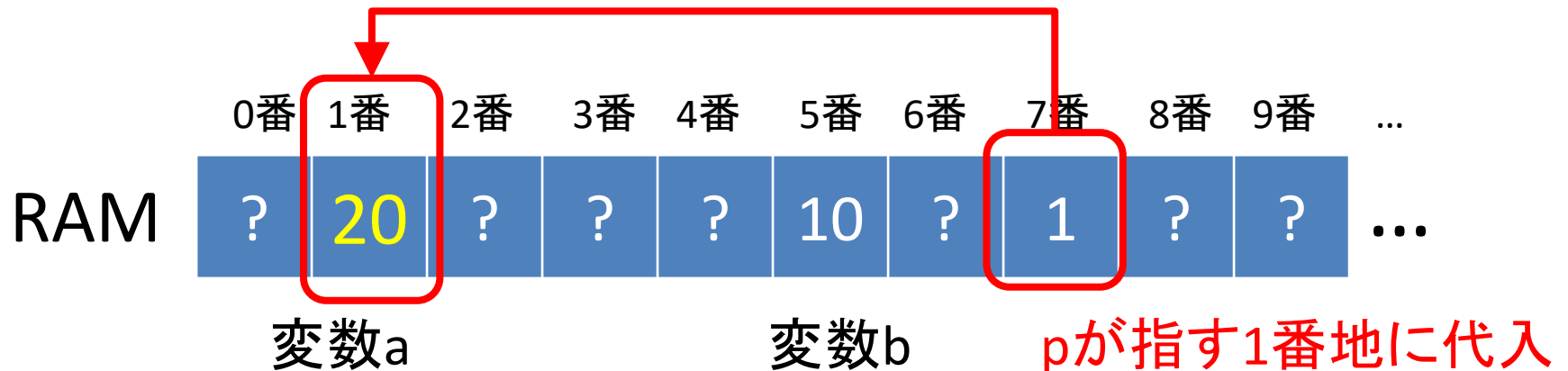
```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```



コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

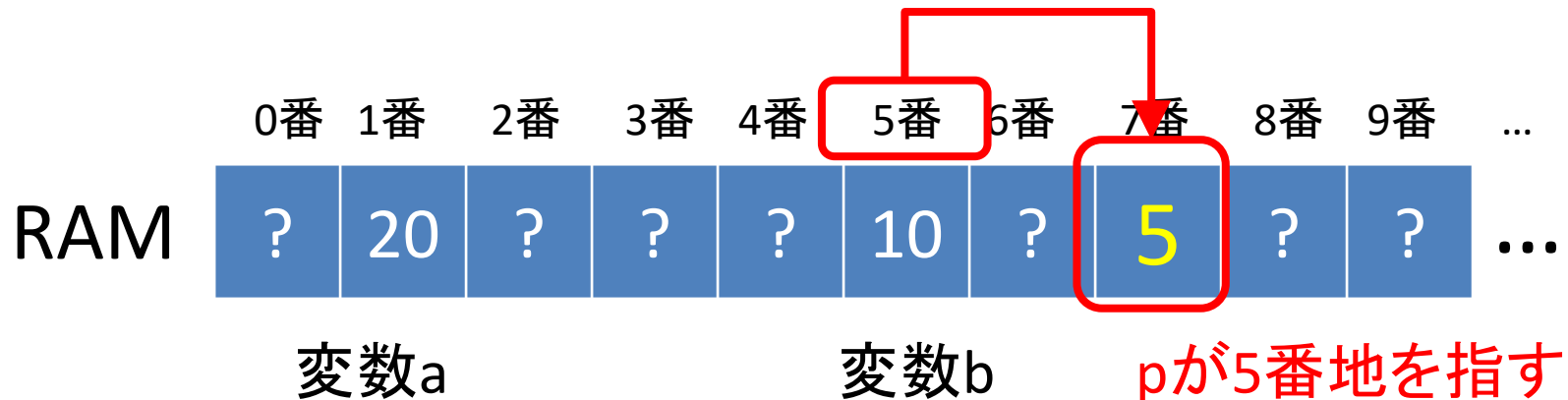
```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```



コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

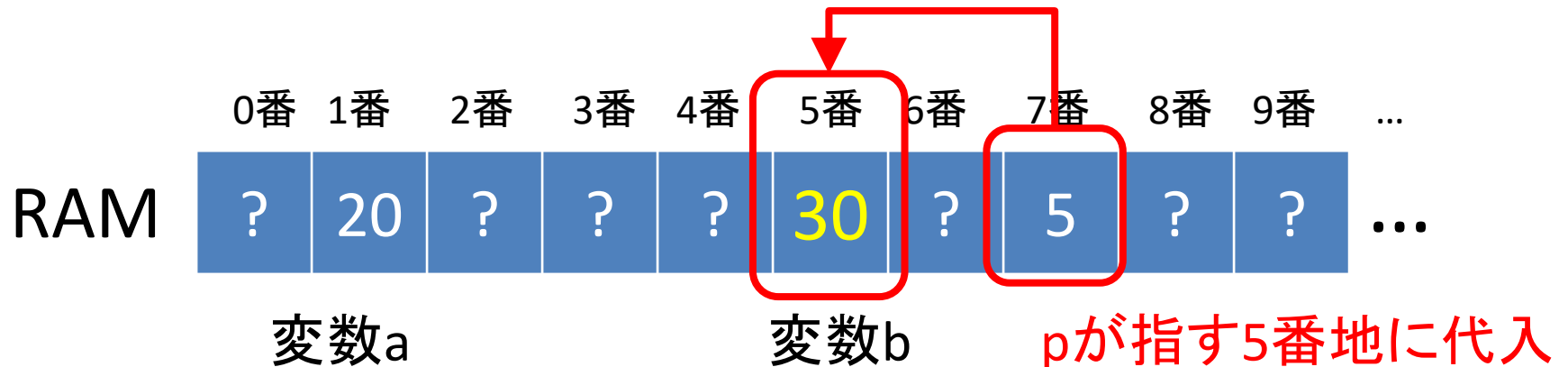
```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```



コンピュータアーキテクチャとの関連

かなり簡単化して説明するため厳密には嘘が多い...

```
int a, b;  
a = 0; b = 10;  
int *p;  
p = &a;  
*p = 20;  
p = &b;  
*p = 30;
```



ポインタ変数の演算

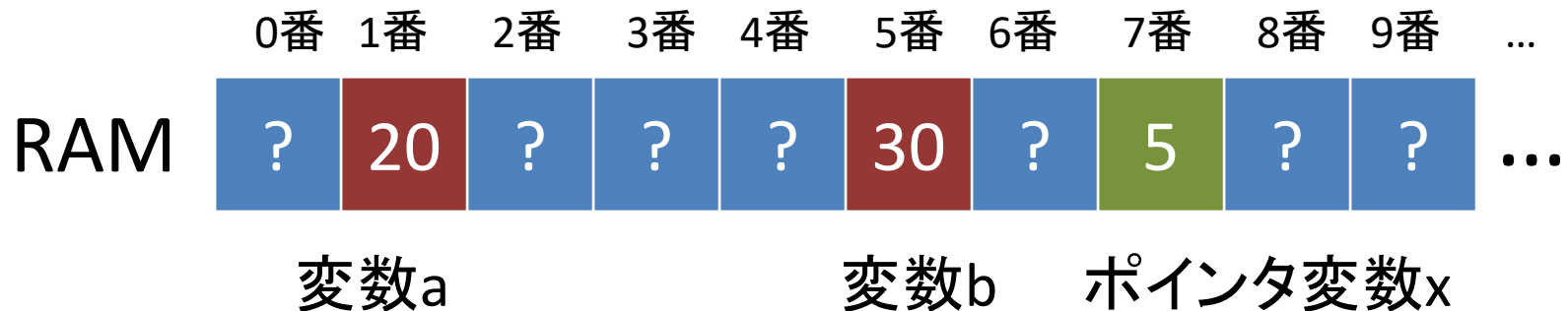
※講義のための例. 本来やってはいけない操作

```
int *x = &b;           // ?
```

```
int *y = x - 4;        // ?
```

```
int z = *y;            // ?
```

```
int w = *(x - 1);      // ?
```



配列とポインタ変数の演算

& は不要

```
int a[6] = {1, 2, 3, 4, 5, 6};
```

```
int *b = a + 2;    // ?
```

```
int c = *(b + 3);  // ?
```

```
int d = b[2];      // ?
```



配列 `a[6]`

特殊なポインタ : nullptr

- ヌルポインタ
- どの番地も指していない

```
int *a = nullptr;
int b = 0;
*a = 0; // エラー
a = &b;
if (a != nullptr) { //nullチェック
    std::cout << *a;
}
```

特殊なポインタ : this

```
class Vector2
{
public:
    Vector2(double ix, double iy) {
        this->x = ix;    this->y = iy;
    }
    Vector2& operator =(const Vector2 &src)
    {
        this->x = src.x; this->y = src.y;
        return *this;
    }
private:
    double x, y;
};
```

まとめ

- 値渡しと参照渡し(おさらい)
- ポインタ, メモリアドレス
- ポインタ変数の演算
- 配列とポインタ, ポインタ演算
- 特殊なポインタ `nullptr`, `this`
- ポインタを活用したデータ構造
 - リスト
 - ツリー

講義しなかったこと

- 型とデータ長
 - char型: 1Byte, int型: 4Byte, int*型: 8Byte
 - ハードウェアをガリガリ使うようなプログラミングでは必須知識（例: GPU、特殊デバイス）

Further Readings

- 「データ構造」と「アルゴリズム」を冠する本
 - 効率の良いプログラムを開発する上で必須知識
- ポインタについて
 - 理解が追いつかなかった場合は, "相性の良い" webサイトや書籍で独習

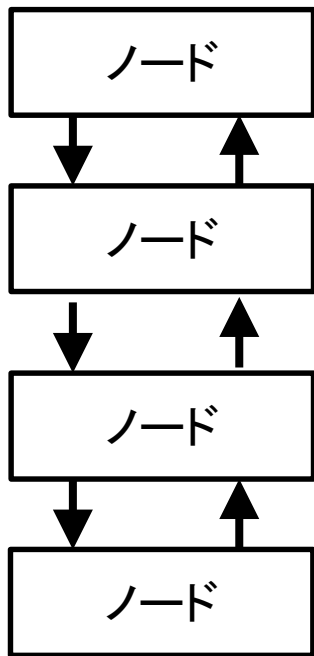
基礎的なデータ構造

ポインタの応用

ポインタを活用したデータ構造

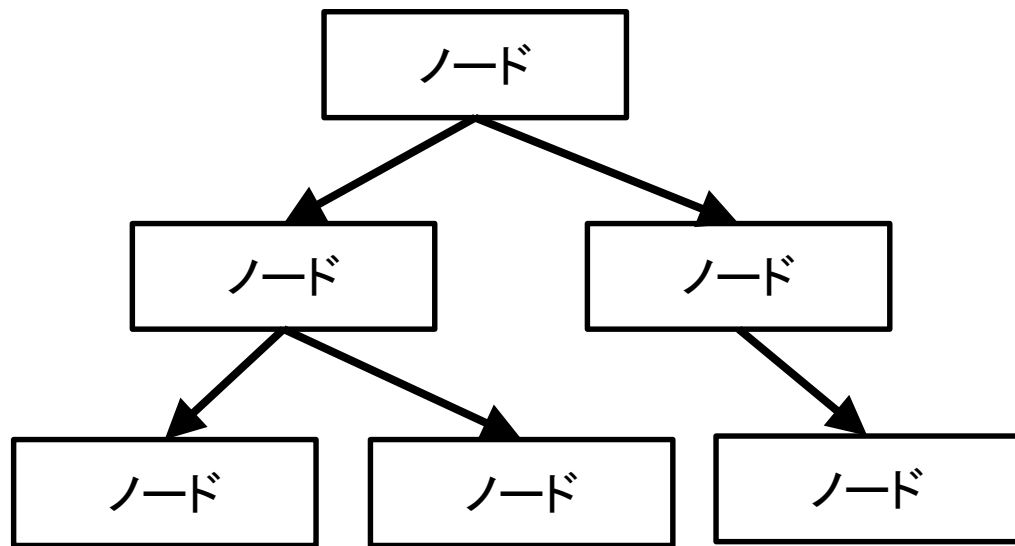
リスト

データ(ノード)の直線状の
数珠繋ぎ



ツリー

枝分かれして接続するデータ
今回は2つに枝分かれ(二分木)

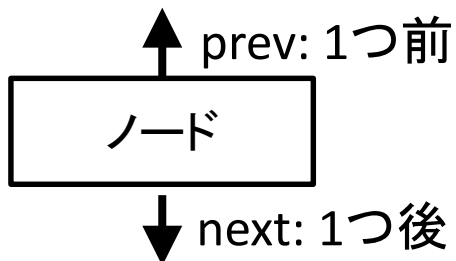


→ ポインタ : ノードの接続

ポインタを活用したデータ構造

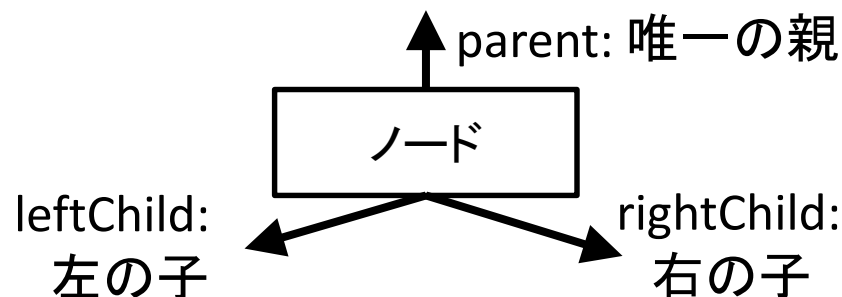
リスト

```
class ListNode
{
private:
    ListNode *prev;
    ListNode *next;
};
```



ツリー

```
class BinaryTreeNode
{
private:
    BinaryTreeNode *parent;
    BinaryTreeNode *leftChild;
    BinaryTreeNode *rightChild;
};
```



List&Tree 演習

- 二分木ノード BinaryTreeNode クラスの宣言と実装
 - ListNode.h, ListNode.cpp, main.cpp を参考に
- 実装した二分木ノードを用いて右図の二分木を構築したうえで、「1, 2, 3, 4, 5, 6, 7」の順に数字を出力するように二分木を操作
 - メンバ関数 Parent() を追加
 - 宣言済みのLeft(), Right() を参考に
 - 出力には BinaryTreeNode::Print()

