


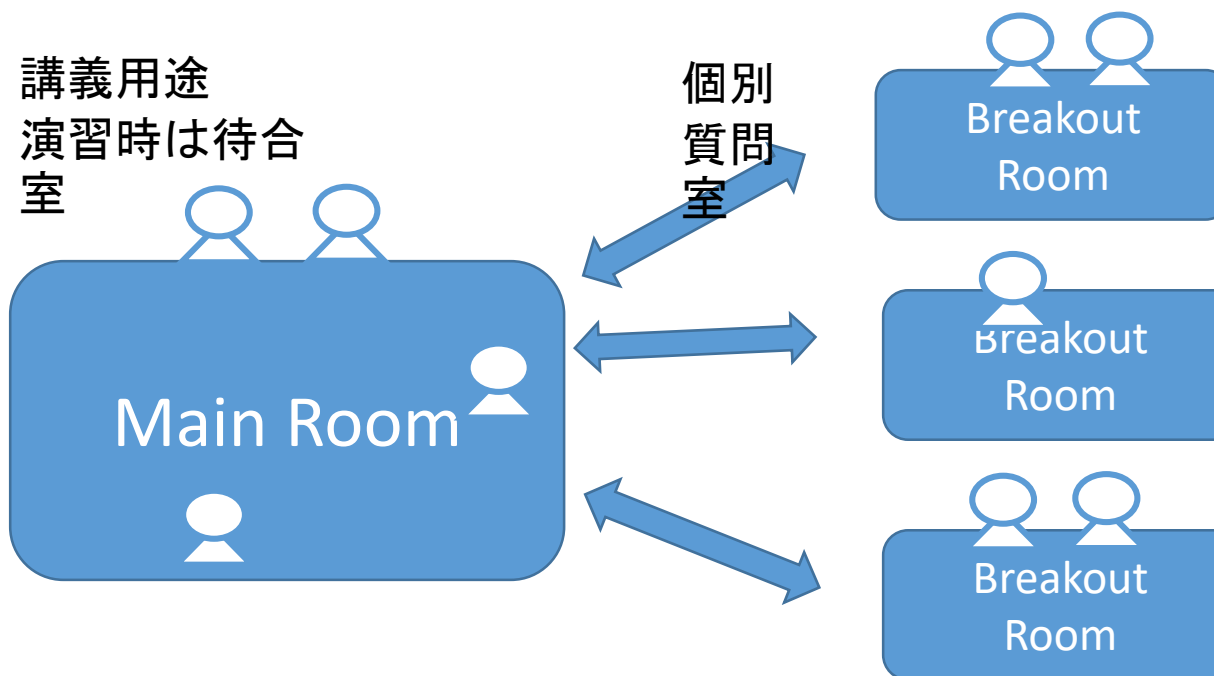
応用アルゴリズム演習 —初日—



鎌田十三郎

今年はオンライン開催

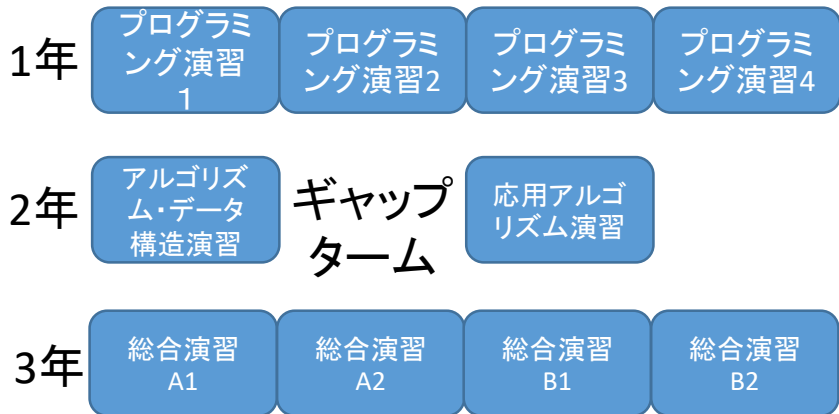
- 演習時間帯は、基本 Zoom に接続してください。
- 鎌田の方で一括で対応します。
- ブレークアウトルームを質問部屋として利用予定
- 授業中の告知や、時間外窓口としてSlack併用



注

- ブレークアウトルームからメインに戻る際は、「ルームを退出」ボタンを押せばよいです。
- その際、「ルーム／セッションを退出」を選びましょう。
- 「ミーティングを退出」を選択すると、Zoom から抜けちゃいます。再度入りなおしてください。

新カリの演習科目



プログラミング演習1-4	プログラミングの基礎（制御構文、関数、scope、配列、構造体、ポインタ、malloc、I/O、makeなど）
アルゴリズム・データ構造演習	講義と連動し、アルゴリズムの基礎を扱う(stack, queue, heap, sort, linked list, tree, searchなど)
応用アルゴリズム演習	応用課題を通して、基本アルゴリズムの応用例・動的計画法・問題のグラフ表現などについて学ぶ
総合演習A1	数値解析、最適化、アルゴリズムのいくつかの代表的な手法に焦点を当てる。(講義連携) シンプレックス法、ガウスの消去法、ニュートン法、多層パーセプトロン
総合演習A2	数値解析、最適化、アルゴリズムを横断した総合的な問題解決方法を体得する。(page rank を題材に random walk, べき乗法、QR法)
総合演習B1	マイクロ系、マクロ系、社会系シミュレーションについての俯瞰的演習(各2コマ、講義連携)
総合演習B2	いくつかのグループに分かれ、各自が選択した1つのテーマに取り組む(プロジェクトベース)

演習の趣旨

具体的なプログラム課題へのアルゴリズムの適用を通して、アルゴリズムの理解を深めるとともに、プログラミング技術の向上を図る。

■ 目標

- プログラミング能力を確かなものにする。
- 既習データ構造・アルゴリズムの理解を深め、
- より複雑なアルゴリズムの学習をおこなう。
- 各種問題に対してプログラム中の問題表現、適切なアルゴリズムの選択および計算量などの見積りができるようになる

注意事項 & アナウンス

- 分からないことを放置せず、早めに質問・対処する
 - できる限り演習時間中に対処する
 - これ以外に、**オンライン・対面質問時間**も設けます
 - 遅れている人は、演習時間外にも頑張りましょう
- カンニング行為(プログラムコピーとか)厳禁
- 利用プログラミング言語は、原則、C 言語
 - サポートはしないですが、C++, Java などでの課題提出も許可します

評価



- 出席状況, レポートにより評価する
- 理由なく欠席した場合は、1回につき10点, 遅刻は5点減点します。
- やむを得ない事情(PC やネットワークトラブルなど)の場合, その旨すぐに担当教員に連絡すること。

演習時間外の質問について



- 演習のない週に質問時間帯を設定する予定
 - といっても、ヘッドセット付けてまっているのは嫌なので、Slack 上でリクエストくれたら Zoom で対応って感じ。
- それ以外の時間帯は Slack で質問しましょう
 - 問題によっては、Slack でスケジュール合わせて Zoom で対応します

演習内容



■ テーマ

- プログラム理解を深める
- 基本データ構造・アルゴリズムを使う
- より高度なアルゴリズムの学習

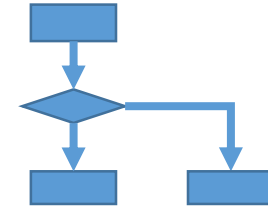
■ スケジュール(予定)

- 10/4, 18, 11/1, 8, 12/6, 20, 1/17
- 10/4: 復習(基本データ構造、スコープ、再帰)、デバッガの利用
- 基本データ構造と探索問題
- 動的計画法
- 問題の理解: グラフ
- 優先度キューと最短路探索

復習: プログラミング言語

■ 計算機

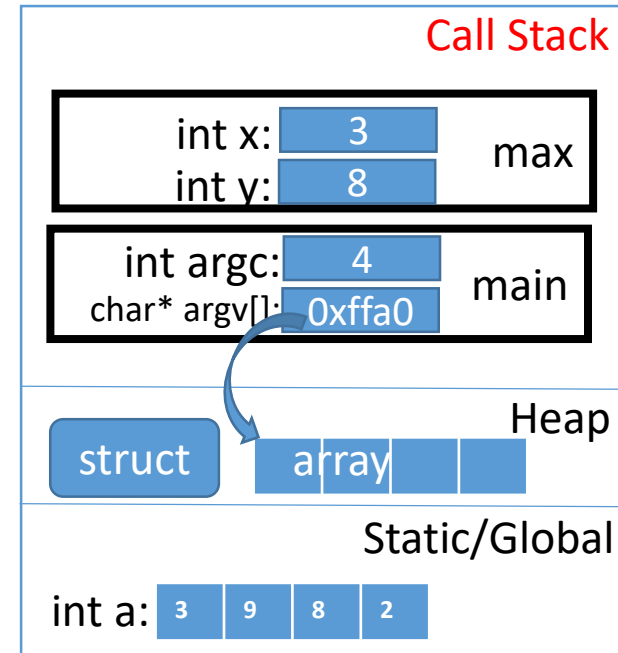
- CPU: 演算 + 制御構文
- メモリ: データ構造



- I/O

```
Start Page EAN2AN_TYPE.bin Config.xml frmV2.cs frmReadEan.cs DBBase.cs L
00000000 11 00 00 00 00 6C 5E 05 12 00 00 00 00 FE 30 01
00000010 E9 03 00 00 00 0F 54 01 EA 03 00 00 00 10 54 01
00000020 E4 1E 00 00 00 5F BD 07 CD 4E 00 00 00 36 80 01
00000030 74 75 00 00 00 62 7A 00 C5 85 00 00 00 C1 0F 06
00000040 C5 85 00 00 00 AC C8 05 E4 38 01 00 00 A7 07 00
00000050 74 3A 01 00 00 A8 07 00 89 4C 01 00 00 6D 1C 00
00000060 F4 5F 01 00 00 56 07 00 57 60 01 00 00 56 07 00
00000070 58 60 01 00 00 57 07 00 71 60 01 00 00 55 07 00
00000080 73 60 01 00 00 53 07 00 BC 60 01 00 00 58 07 00
00000090 20 61 01 00 00 A6 07 00 00 62 01 00 00 54 07 00
000000a0 0E 64 01 00 00 7B 33 03 43 64 01 00 00 A2 AA 00
000000b0 45 64 01 00 00 83 37 02 4B 64 01 00 00 36 F0 02
```

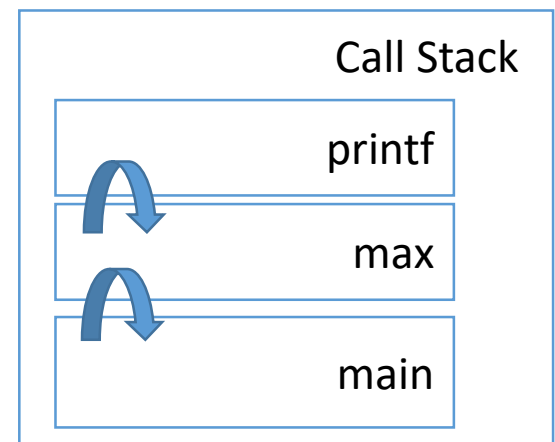
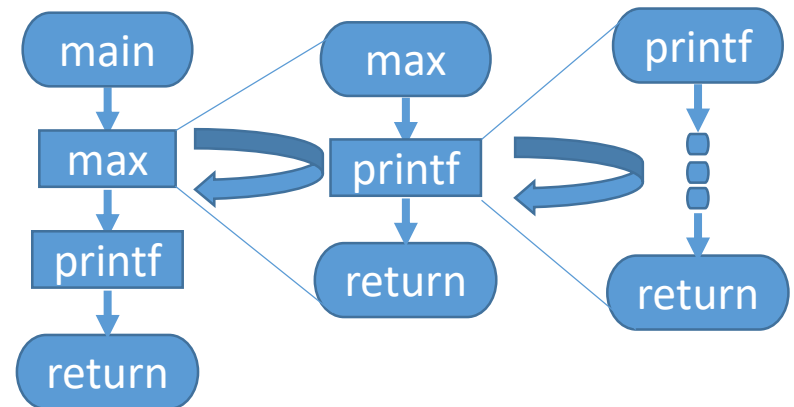
■ プログラムの実行イメージを しっかりつけましょう



復習：関数呼び出し

- 関数呼び出し: 終わったらcallerに戻る
 - 関数スタック

```
int max(int x, int y) {  
    printf("max(%d, %d) is called¥n", x,y);  
    if(x<y) return y;  
    return x;  
}  
int main(int argv, char* argv[]) {  
    int a = 10;  
    int b = max(a*3, a+4);  
    printf("result: %d¥n", b);  
    return 0;  
}
```



復習：データ構造 (1/3)

- 値：10, 14 とか
- 式：i, i+3 とか
式を評価して値に、
で、値を変数に格納

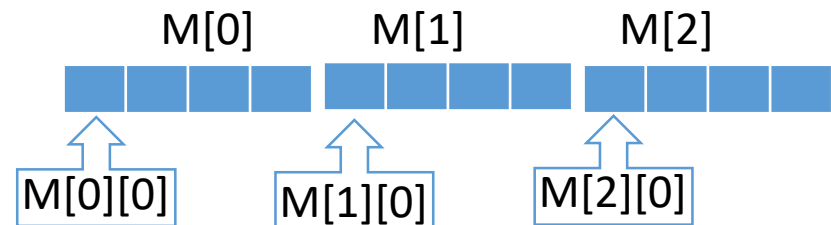
- 変数: `int i;`
 - `int` の箱,
 - 「整数型の**値**」を格納

`int i:` 10

- 配列: `int a[4];`
 - `int` 4個分の箱

`int a[4]:` 3 9 8 2
a[0] a[1] a[2] a[3]

- 多次元配列: `int M[3][4];`
 - 「`int` 4個の箱」が3個



```
for(i=0; i<3; i++)  
  for(j=0; j<4; j++)  
    r += M[i][j];
```

復習: データ構造 (2/3)

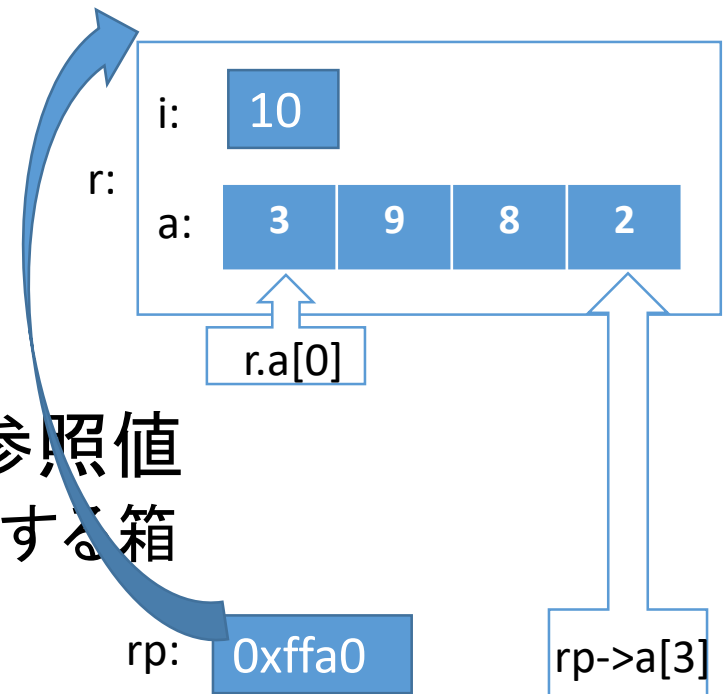
- 構造体(struct):
複数の変数を持った箱

```
struct record {  
    int i;  
    int a[4];  
};  
struct record r;
```

型宣言
変数宣言

- ポインタ: 各データ型への参照値
 - ポインタ変数: ポインタを格納する箱

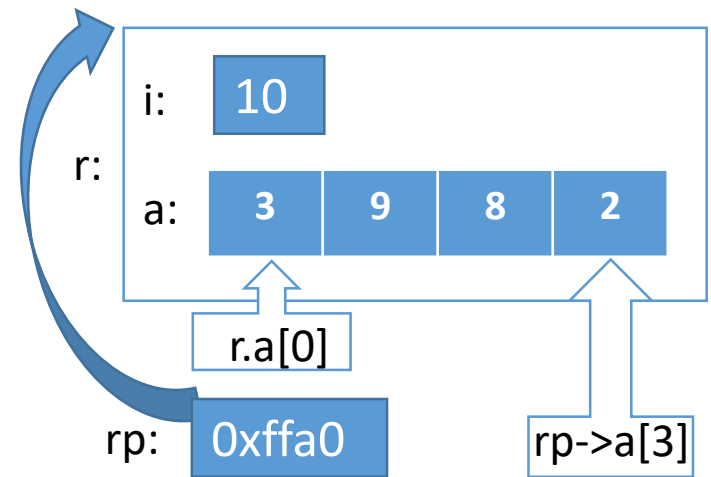
```
Struct record * rp;
```



復習：データ構造 (3/3)

- typedef:
データ型に名前をつける

```
typedef struct record {  
    int i;           型宣言  
    int a[4];       + typedef  
} record_t, *record_tp;  
record_t r;        構造体変数  
record_tp rp;     ポインタ変数
```



- ポインタ演算：省略☺
 - この演習では使いません

鎌田は好き
だけど

復習：各種変数 & メモリ領域

- 局所変数 (local, auto)
 - 関数内で宣言された変数

プログラムの内部構造

Call Stack

- 関数呼び出し関係を表すスタック

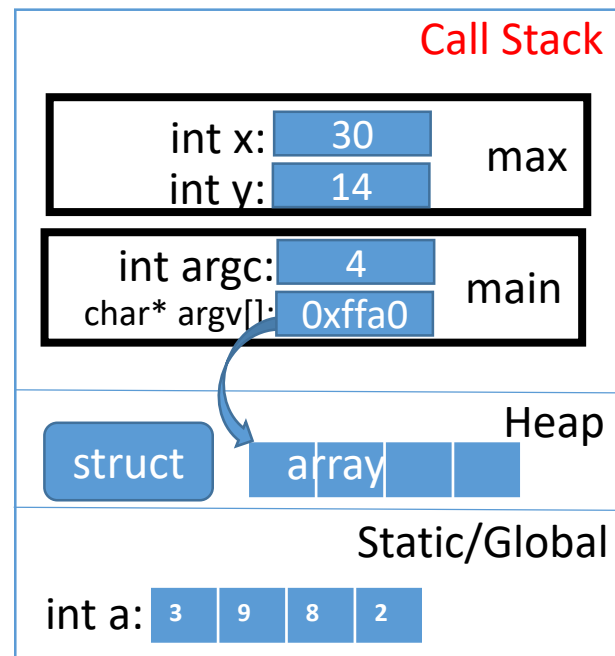
Stack Frame

- 各「関数呼び出し」相当
- 局所変数をここに配置

- ヒープ: malloc で領域確保

- 大域変数 (static):

- 変数はプログラム実行に対して1つ確保するだけ

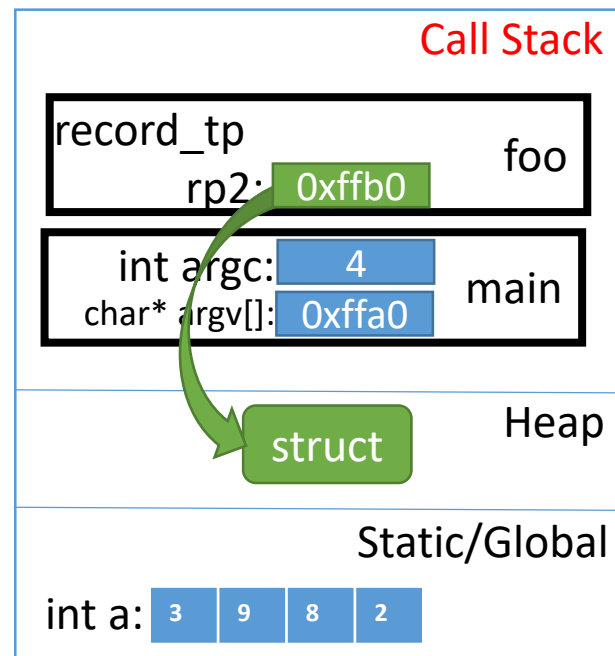


復習：メモリの動的確保

■ malloc

```
void foo() {  
    record_tp rp2 =  
    (record_tp)malloc(sizeof(record_t));  
}
```

- サイズを調べて (sizeof)、
- メモリ確保して (malloc)、
- 対応するポインタ型にキャスト



変数のスコープ&関数呼び出し

■ {} で切った範囲でのみ変数が有効に

- 名前が被ったら内側優先 (でも普通は変えるよね)
- 変数はドンドン使ってOK

```
void search(node_tp node) {  
    if(node->visited==1) {  
        node_tp s = node->s;  
        if(s != NULL) search(s);  
    }  
}
```

■ 関数呼び出し

- 値渡し(call by value): 引数の評価値を渡すだけ
- 呼び出し側(caller)と呼ばれた側(callee)は別スコープ

```
void max(int x, int y) {  
    if(x<y) return y;  
    return x;  
}  
  
int main(void) {  
    int a = 10;  
    int b = max(a*3, a+4);  
    printf("result %d\n", b);  
    return 0;  
}
```

int x:	30	max
int y:	14	
int a:	10	main
int b:	??	

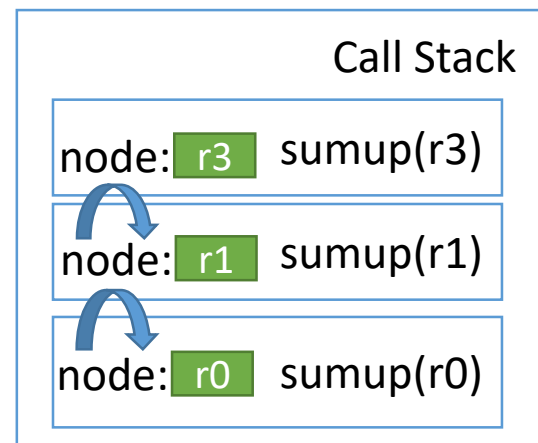
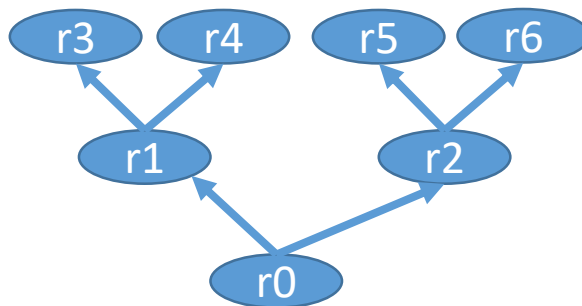
再帰呼び出し

- 世の中、再帰的に表現した方が分かりやすいことが多い

- 例：木構造の探索

```
sumup(node) = sumup(node->left)  
              +sumup(node->right) + node->weight;
```

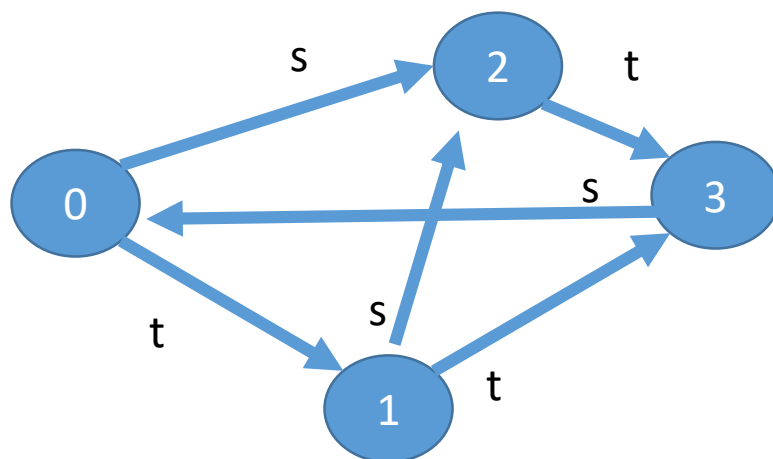
- 再帰呼び出しを使えば、素直に表現可能。
素敵！



課題

■ 有向グラフを深さ優先探索してみよう

- 今日、プログラムはあげます
- 代わりに、処理の流れを
しっかり追っかけましょう
- 設問は長いので Web ページで。



```
void search(node_tp node) {  
    node->visited++;  
    printNode(node);  
    if(node->visited==1) {  
        node_tp s = node->s;  
        node_tp t = node->t;  
        if(s != NULL) search(s);  
        if(t != NULL) search(t);  
    }  
}
```

デバッガを使ってみよう



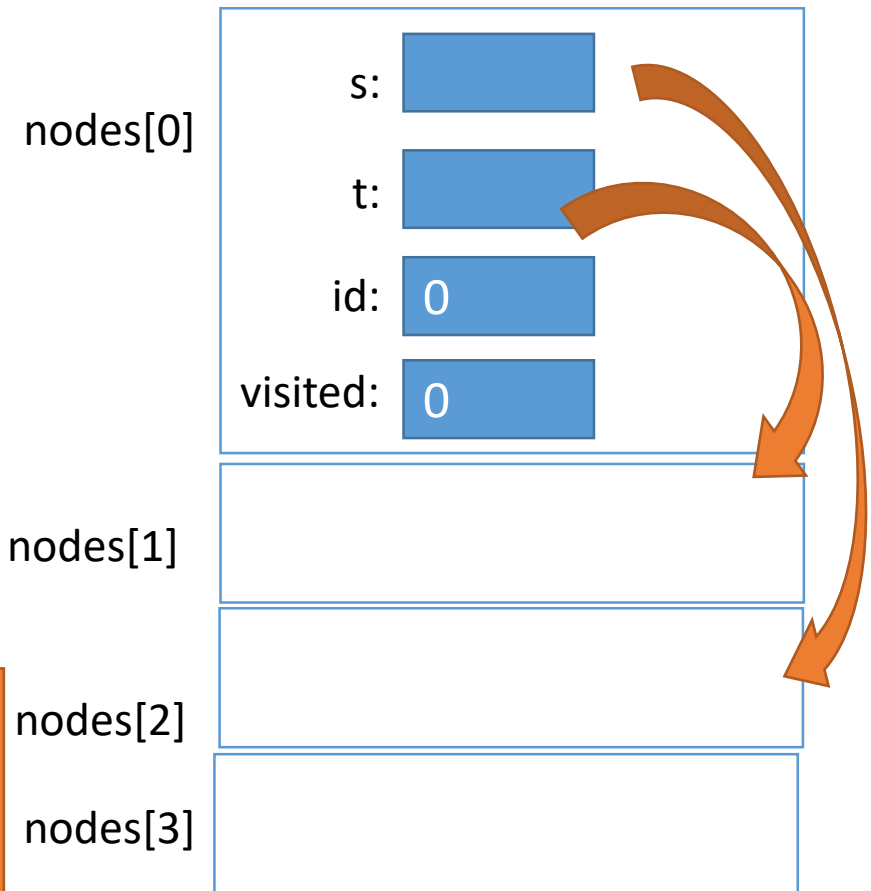
- 統合開発環境上でデバッガを使ってもらいます
 - 関数の呼び出し関係なども視覚的に理解しましょう
 - ポインタがアドレスというのも実感しましょう
- 今年は、VS code および GitPod を紹介します

データ構造

- グラフ構造の例
(各ノードは s, t で
他のノードを参照可能)

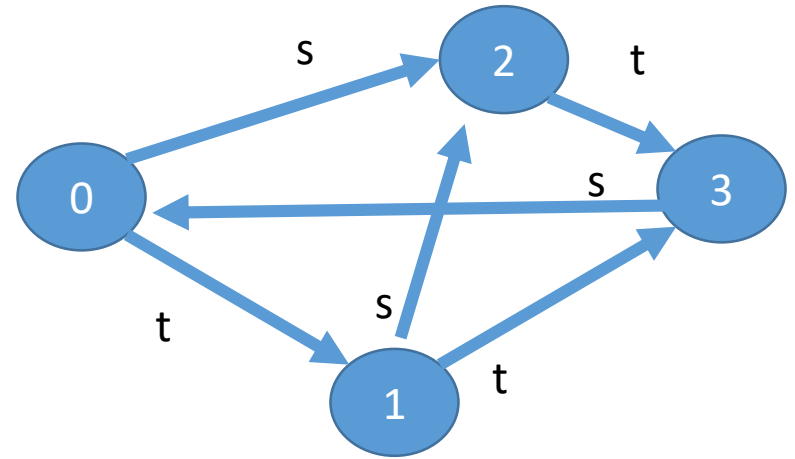
```
typedef struct node {  
    struct node * s;  
    struct node * t;  
    int id;  
    int visited;  
} node_t, * node_tp;
```

```
void link(node_tp node, node_tp s, node_tp t) {  
    node->s = s; node->t = t;  
} /* 例: link(&nodes[0], &nodes[3],  
             &nodes[1]); */
```



データ構造

- グラフ構造
(各ノードは s, t で
他のノードを参照可能)



```
typedef struct node {  
    struct node * s;  
    struct node * t;  
    int id;  
    int visited;  
} node_t, * node_tp;
```

```
initNodes(4);  
link(&nodes[0], &nodes[2], &nodes[1]);  
link(&nodes[1], &nodes[2], &nodes[3]);  
link(&nodes[2], NULL, &nodes[3]);  
link(&nodes[3], &nodes[0], NULL);
```

深さ優先探索

- 今回は、再帰呼び出しで深さ優先探索

```
void dfs(node_tp node) {  
    node->visited++;  
    printNode(node);  
    if(node->visited == 1) {  
        node_tp s = node->s;  
        node_tp t = node->t;  
        if(s != NULL) dfs(s);  
        if(t != NULL) dfs(t);  
    }  
}
```

