Programming, numerics and optimization

Lecture A-3: Programming basics III

Łukasz Jankowski
ljank@ippt.pan.pl

Institute of Fundamental Technological Research (IPPT PAN)
Room 4.32, Phone +22.8261281 ext. 428

March 16, 2021[1]

[1] Current version is available at http://info.ippt.pan.pl/~ljank.

Outline

Outline

Pointers

In C++, if you use references to variables and the Standard Template Library (STL), in most of the cases you can avoid using pointers.

However,

1. for numerical computations you need arrays, while the STL containers corresponding to arrays are slow. C-style arrays are lower-level, so much quicker (and 1D arrays are even simpler). But for them you need pointers.
2. pointers seem to be a part of the fundamental skills in programming.

Pointers

Memory areas

static For global (and static) variables

- automatically zeroed at the creation time
- exist till the end of the program

stack For local (automatic) variables and function parameters/results that are passed by value

- contain random data (if not initialized)
- exist till the end of their (local) scope only

heap For *dynamic variables* (created with the operator **new**), referenced by pointers

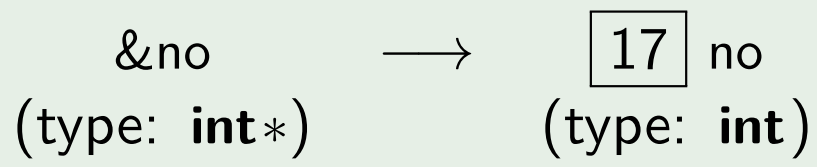
- contain random data (if not initialized)
- exist until destroyed

Reference operator &

Each variable resides somewhere in the memory (static, stack, heap). Its location (which is called its *address*) can be obtained by the *reference operator* &.

Example

```
int no = 17;
cout <<"address of no: " <<&no <<endl;
```



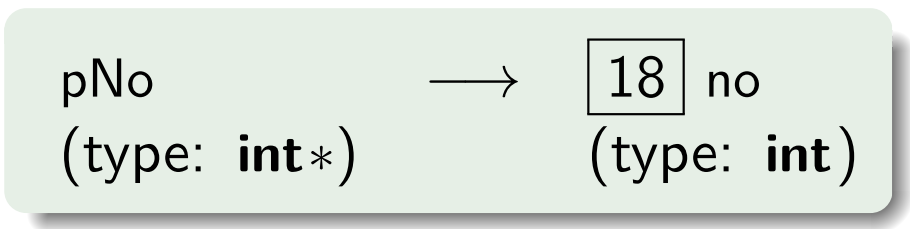
Definition, dereference operator $*$

Pointer variables can be defined in the standard way. For example, the type of a pointer to a variable of type **int** is **int** $*$.

```
int no = 17;
int *pNo = &no;
cout <<"address of no: " <<pNo <<endl;
```

Having a pointer to a variable, the variable itself can be accessed via the *dereference operator* $*$.

```
int no = 17;
int * pNo = &no;
*pNo = 18;
```

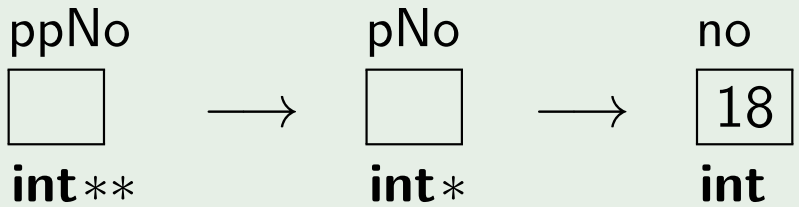


Pointers to pointers

A pointer-type variable also resides somewhere in the memory. Hence its address can be read by the *reference operator* and it can also be accessed by the *dereference operator*.

Example

```
int no = 17;
int *pNo = &no;
int **ppNo = &pNo;
cout <<"address of no: " <<pNo <<endl;
cout <<"address of pNo: " <<ppNo <<endl;
**ppNo = 18;
```



NULL pointer

NULL is a special pointer value indicating that the pointer is not referencing any valid variable.

Example

```
int no = 17;
int * pNo;      // pNo contains garbage

pNo = NULL;     // pNo is null (points to nothing)
pNo = &no;      // pNo contains the address of no
```

All pointer variables, which point to valid data, are guaranteed to compare unequal to NULL.

Operator **new**

- Pointers provide a way to use a *variable* amount of memory, to be decided about during runtime of the program.

```
int * pNo;          // pNo contains garbage
```

pNo [ ] ⟶ ???

This is especially evident in the case of dynamic arrays.

- Additional memory for a *dynamic variable* is requested via the new operator:

```
int *pNo;
pNo = new int;      // memory allocated,  uninitialised
```

pNo [ ] ⟶ [?] *pNo

```
*pNo = 10;
```

Operator **new**

- Initialization can be done also immediately by:

```
int *pNo;
pNo = new int(10);   // allocated & initialized
```

  or, in a single line,

```
int *pNo = new int(10);
```

- In C++, do not use the C-specific malloc function.

Operator **new**

...no more memory?

- If there is no more memory available for allocation, operator **new** returns the NULL pointer. In larger programs using large amounts of memory it should be always tested, for example by

```
int *pNo
if((pNo = new int) == NULL) {
    cout <<"memory overflow" <<endl;
    abort(); // abort the program
}   // if ok, continue execution
```

Operator **delete**

- A dynamic variable allocated with the **new** operator is released (deallocated) with the **delete** operator:

```
int * pNo;                  // pNo contains random data
                                        pNo [ ] ⟶ ???
pNo = new int(10); //  initialised  with 10
                                        pNo [ ] ⟶ [10] *pNo
 ...
delete pNo;
                                        pNo [ ] ⟶ ???
pNo = NULL;
                                        pNo [NULL]
```

- An attempt to delete a non-allocated memory area results in a runtime error (unless the pointer is NULL).

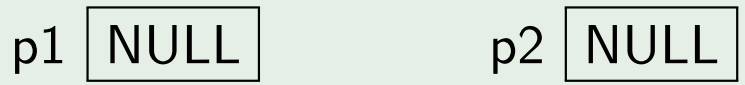
Typical errors and problems

- *Memory leaks* — 'loosing' allocated memory without deallocating it before. Memory leaks are especially dangerous when done iteratively in loops, as then they can lead to...
- *Memory overflow* — no free memory available for a dynamic variable that is being allocated with **new**.
- *Memory corruption*
  - leading to application crash (abnormal termination)
    - assigning a value to an unallocated dynamic variable
    - deallocating an already deallocated dynamic variable
    - deallocating an unallocated dynamic variable
  - harder to detect
    - assigning a value to an already deallocated dynamic variable
    - accessing an array element out-of-range

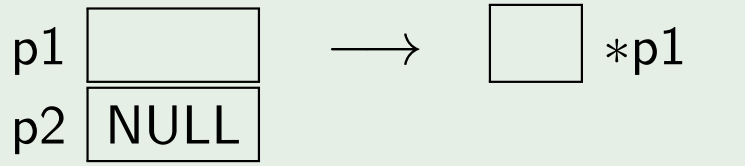
Typical errors and problems

p2 becomes a *dangling pointer*

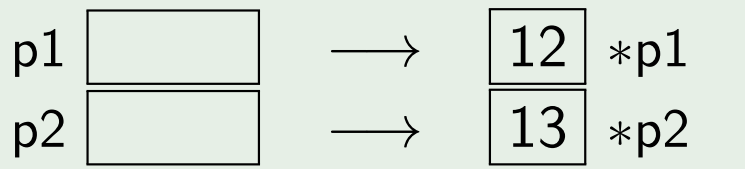
```
int *p1, *p2;
```



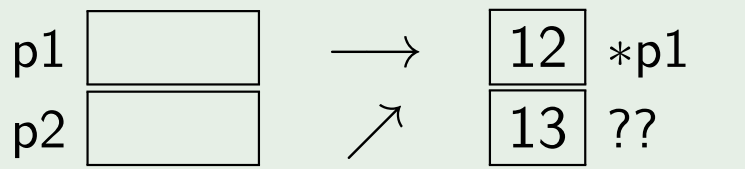

```
p1 = new int;
```



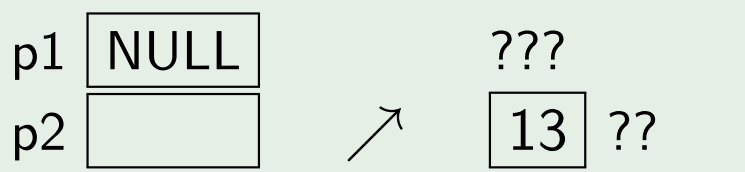

```
*p1 = 12;
p2 = new int(13);
```




```
p2 = p1;
// object *p2 lost!
// use *p2 = *p1;
```




```
delete p1;
p1 = NULL;
```




```
*p2 = 100;   // error!
```

Outline

Arrays in C++

In C++, you can avoid using C-style fixed-size arrays and use the corresponding containers from the Standard Template Library (STL), which are object-oriented and dynamic (variable-length and easily expandable).

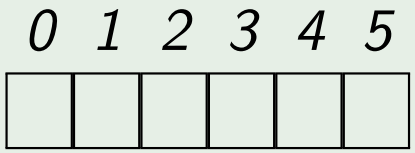
However,

- these containers (vector, valarray) are much slower, which is a serious drawback in numerical applications
- *first things first*: classical fixed-size arrays are more fundamental (and they are anyway the back-end for most STL containers).

Definition — 1D array

1D array of six integers, called tab

```
int tab[6];
```



- The elements are numbered from 0 to n-1.
- In C++, array size must be a constant expression (known exactly at compilation time). In C, local variable-length arrays (VLAs) are allowed.

ok

```
const int size = 30;
double tab[size];
```

wrong in C++, ok in C

```
int size;
cin >> size;
double tab[size];
```

Definition — fixed-size and variable-length arrays

- Sizes of *global* arrays must be constant expressions (known at compilation time) in both C and C++
- In C, *local* variable-length arrays (VLAs) are allowed since 1999 (in the standard "C99"), but
- In C++, VLAs had been expected to be allowed in the standard published in 2011 ("C++11"). However, they have not been included[2] and are not a part of C++ (even if they do compile in many compilers).

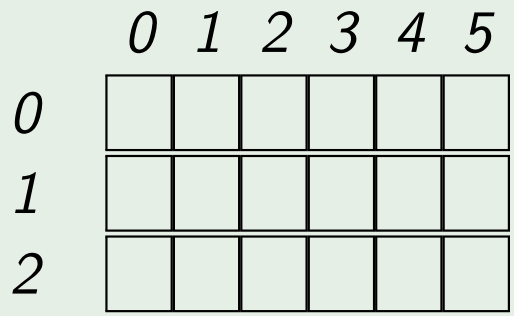
Local VLAs involve performance penalty and they can be dangerous: as local variables they are stored on the stack, which is much smaller than the heap (stack overflow can occur).

[2] The reason was that there exists a specialized container in C++ STL (std::vector), and so C-style VLAs have been deemed unnecessary.

Definition — multidimensional arrays

2D array of 3×6 integers, called tab2

```
int tab2[3][6];
```



- The rows, columns, etc. are numbered from 0.
- In C++, all the dimensions of a local array must be constant expressions (exactly known at compilation time).

Initialization

- Global arrays (defined outside any function) are automatically initialized with zeros.
- Local arrays (defined within a local scope, e.g., a function) are not initialized.
- Arrays (both global and local) can be initialized manually:

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

0	*1*	*2*	*3*	*4*
1	2	3	4	5

- If last elements are zeros, they can be skipped. However, the size of the array must be then explicitly stated.

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

0	*1*	*2*	*3*	*4*
1	2	3	0	0

Initialization — multidimensional arrays

- Multidimensional arrays are initialized in a similar way:

```
int a[3][5] = {{1,2,3,4,5},{8,6}};
```

	0	*1*	*2*	*3*	*4*
0	1	2	3	4	5
1	8	6	0	0	0
2	0	0	0	0	0

- The first dimension can be skipped:

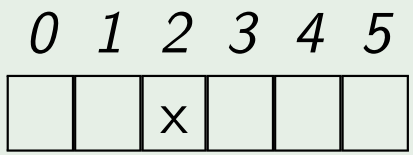
```
int a[][5] = {{1,2,3,4,5},{8,6}};
```

	0	*1*	*2*	*3*	*4*
0	1	2	3	4	5
1	8	6	0	0	0

Accessing an array

Example

```
int tab[6];
tab[2] = 15;      // third element (no. 2)

int x;
x = tab[2];

int n = 2;
cout << tab[n];
```



Do not write past the end of the array:

```
tab[6] = 1;    // compiled ok (!), but a dangerous error
```

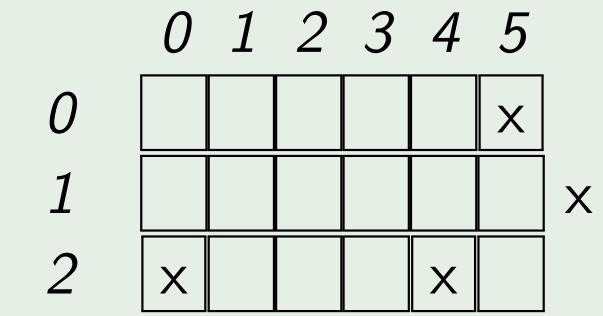
Accessing an array

Example

```
int tab2[3][6];
tab2[2][4] = 15;        // third row (no. 2), fifth column (no. 4)

int x = tab2[2][0];
x = tab2[2][0];         // third row (no. 2), first column (no. 0)

x = 0;
int y = 4;
cout << tab2[x][y+1]; // first row (no. 0), last column (no. 5)
```

Do not write past the end of the array:

```
int x = 1, y = 5;
tab2[x][++y] = 1;
```

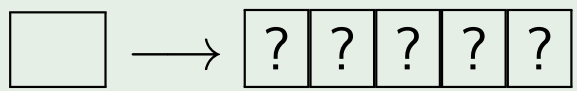


Dynamically-allocated arrays

- Arrays mentioned so far were either local or global.
  - They have to be limited in size
  - They are not easily expandable, and in general, their dimensions should be hardcoded
- Dynamically-allocated arrays
  - can be much larger in size
  - their dimensions are decided in runtime
  - can be reallocated at any time (support for dynamically expandable arrays)

  They are created with the **new**[] operator:

```
int *pNo;              // pNo contains garbage
pNo = new int[5];      // do not confuse with new int(5)!
```



Their elements are referenced in the standard way. For example, use pNo[0] = 1; to set the first element.

Dynamically-allocated arrays

- Elements of standard (global, local) and dynamically-allocated arrays can be referenced in the same way, because the identifier of a standard array is in fact an automatically created (constant) pointer to the first element of the array:

```
int tab[6];
```

tab → | x | ? | ? | x | ? | ? |

- It means that the elements of arrays can be referenced also using *pointer arithmetic*:

```
tab[0] = 1;
tab[3] = 2;
```

or

```
*tab = 1;
*(tab+3) = 2;
```

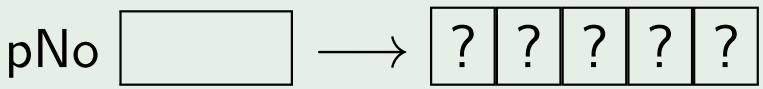
```
for(int i=0; i<5; ++i)
  a[i]=i;
```

```
for(int *p=a; p-a<5; ++p)
  *p=1;
```

Dynamically-allocated arrays

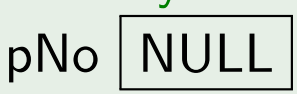
- A dynamically allocated array is deleted with the operator **delete** [] .

```
int *pNo;
pNo = new int[5];
```




```
...
delete [] pNo;
```


```
pNo = NULL; // for safety
```



- An attempt to delete a non-allocated memory area results in a runtime error (unless the pointer is NULL).

Dynamically-allocated arrays

The differences can be summarized as follows:

local array

- In C++, the length should be known in advance. In C, small VLAs are allowed.
- deleted automatically
- stored on stack (must be small)

dynamically-allocated array

- the length can be decided in runtime
- must be explicitly deleted
- stored on heap (can be large)

Dynamically-allocated multidimensional arrays

If all its dimensions except the first are known at the compilation time, a multidimensional array can be allocated in a single line by

A no $\times$ 10 $\times$ 12 array of int

```
int no;
...  // for example cin <<no;
int (*tab)[10][12] = new int[no][10][12];
...
delete[] tab;
```

Dynamically-allocated multidimensional arrays

An array with more unknown dimensions (or a non-rectangular array) is more tricky and error-prone:

A no1 × no2 array of **int**

```
int no1, no2;
...
int **tab = new int*[no1];
for(int i=0; i<no1; ++i)
    tab[i] = new int[no2];
...
for(int i=0; i<no1; ++i)
    delete[] tab[i];
delete[] tab;
```

Often, it is easier to use the STL containers.

Arrays as function arguments

Arrays can be passed as parameters to functions

```
void printMe(int tab[], int length);
```

In the case of multidimensional arrays, all dimensions except the first must be explicitly specified

```
void printMe(int tab[][5], int length);
```

Arrays as function arguments — example

Referencing by index

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

void reversePrint(int tab[], int no) {
    while (no--> 0)   // no is passed by value
        cout <<tab[no] <<'\t';
    cout <<endl;
}

int main(void) {
    int a[5] = {1,2,3,4,5};
    reversePrint(a,5);
    // Dev C++ users might add: system("pause");
    return 0;
}
```

Arrays as function arguments — example

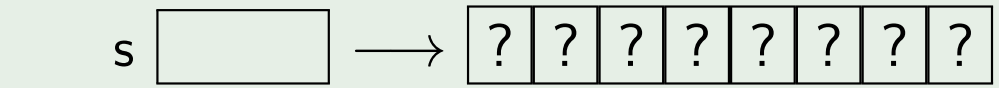
Referencing by pointer arithmetics

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

void reversePrint(int *p1, int *p2) {
    while (p1<=p2)
        cout <<*(p2--) <<'\t';
    cout <<endl;
}

int main(void) {
    int a[5] = {1,2,3,4,5};
    reversePrint(a,a+4);
    // Dev C++ users might add: system("pause");
    return 0;
}
```

Character arrays

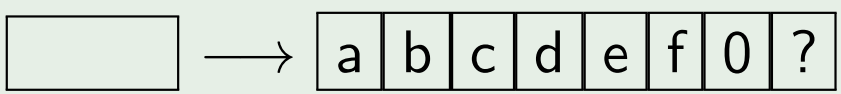
- Strings in C++ can be handled by the string class.
- Strings are in fact sequences of characters, and the standard C way to handle strings are character arrays.

```
char s[8];
```



- The character '\0' is used to mark the end of the string.
- Initialization of strings

```
char s[8] = {'a','b','c','d','e','f','\0'};
                        // space needed also for '\0'
char s[8] = "abcdef"; // char s[]="abcdef";
```



- Character arrays can be also allocated dynamically.

Outline

Structures

- A group of variables of different types can be collected into a single variable (a *structure*) of a new user-defined type.
- Example of a declaration of the type:

```
struct circle {
    float x, y, radius;
    int colorNo;
};                  // notice the semicolon here
```

- Definition of two variables of the type circle

```
circle c1, c2;
```

Structures

```
circle c1, c2;
```

- Members of a structure can be accessed separately

```
c1.x = c1.y = 0;
c2.x = c1.x + 1;
```

- The structure can be accessed as a whole, too

```
c2 = c1;      // field by field copy
```

Structures

Example, **struct** circle already defined

```
void drawCircle(const circle &c) {  // by reference
    /* function body */
}

int main(void) {
    circle c1, c2;

    c1.x = c1.y = 100;
    c1.radius = 200;
    c1.colorNo = 17;
    c2 = c1;
    c2.radius = 10;
    drawCircle(c1);
    drawCircle(c2);
    return 0;
}
```

Arrays of structures

Arrays can be built of structures (as of variables of any other type)

```
circle tab[10];    // global array - zeroed

int main() {
    // ...

    for(int i=0; i<10; ++i)
        tab[i].radius = 10+10*i;

    // ...
}
```

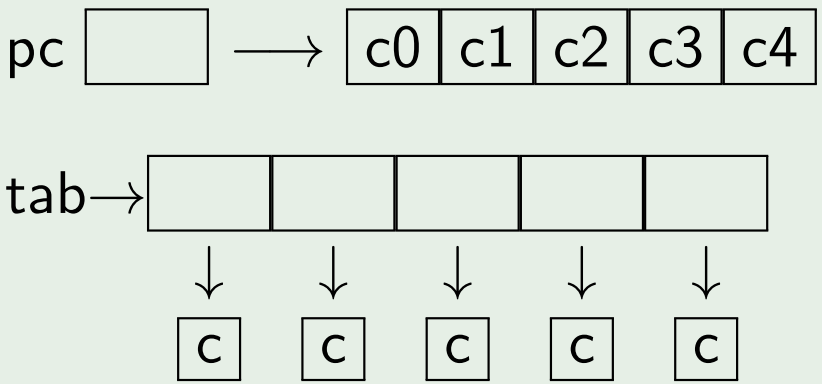
Structures and pointers

(Arrays of) structures can be created and destroyed dynamically

```
circle *pc, *tab[5];
            // single NULL & an array of NULLs

int main() {
    pc = new circle[5];
            // array of circles, contains garbage
            // pc is a pointer to the 1st element

    for(int i=0; i<5; ++i)
        tab[i] = new circle;
            // array of pointers to circles

    // do not forget to deallocate the memory
}
```

Structures and pointers

Data structure




```
circle *pc, *tab[5];

int main() {
  pc = new circle[5];
  for(int i=0; i<5; ++i)
    tab[i] = new circle;
```

```
  delete [] pc;
  pc = NULL;
  for(int i=0; i<5; ++i){
    delete tab[i];
    tab[i] = NULL;
  }
// ...
```

Structures and pointers

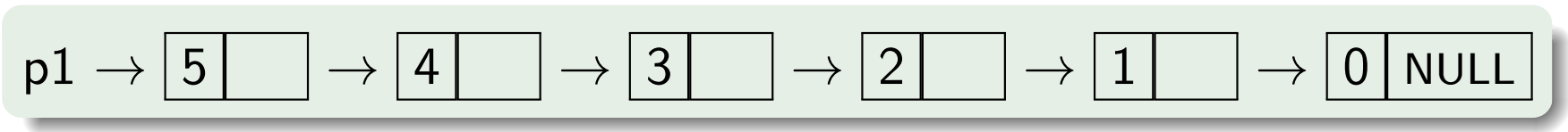
Both structures and their members can be pointed to by pointers of the respective types

```
circle *pc;
float *pR;              // NULL pointers

int main() {
    pc = new circle;

    (*pc).x = 10;       // *pc.x = 10; would be an error
                        // ('pc' is not a structure)
    pc->y = 0;          // note the simpler referencing
    pR = &pc->r;        // a pointer to member
                        // (the same as 'p = &(pc->r);')
    // ...
    delete pc;
    pc = NULL;
//...
```

Lists

A member of a structure can be a pointer to a structure of the same type.

```
struct data {
    int n;
    data *p;
};
```



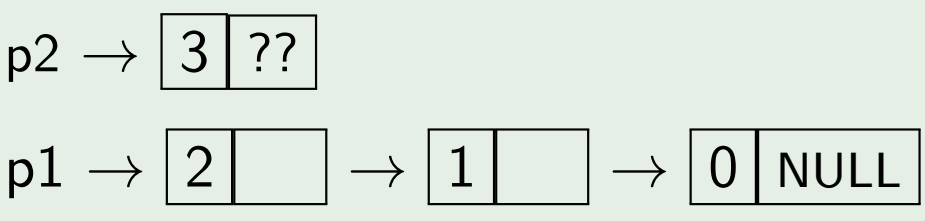
Lists

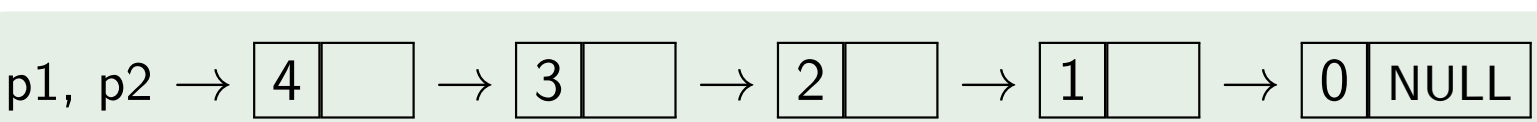
Building a list

```
data *p1=NULL, *p2;
for(int i=0; i<6; ++i) {
    p2 = new data;
    p2->n = i;    // data
    p2->p = p1;
    p1 = p2;
}
```

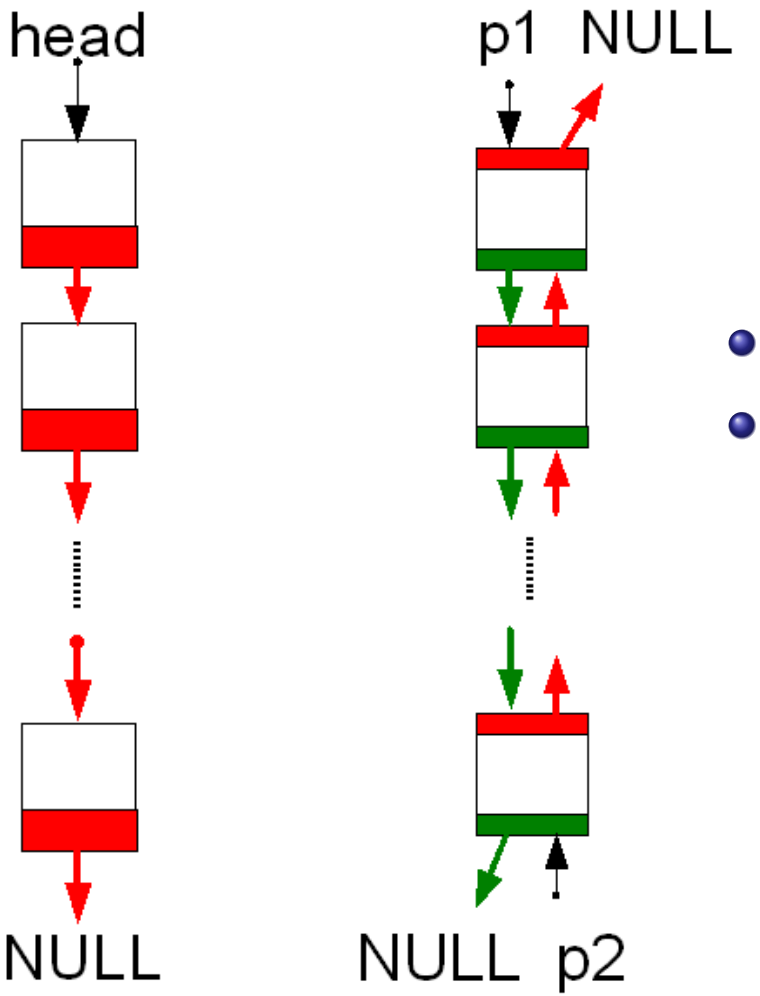
Destroying a list

```
while(p2) {
    p1 = p2->p;
    delete p2;
    p2 = p1;
}
```



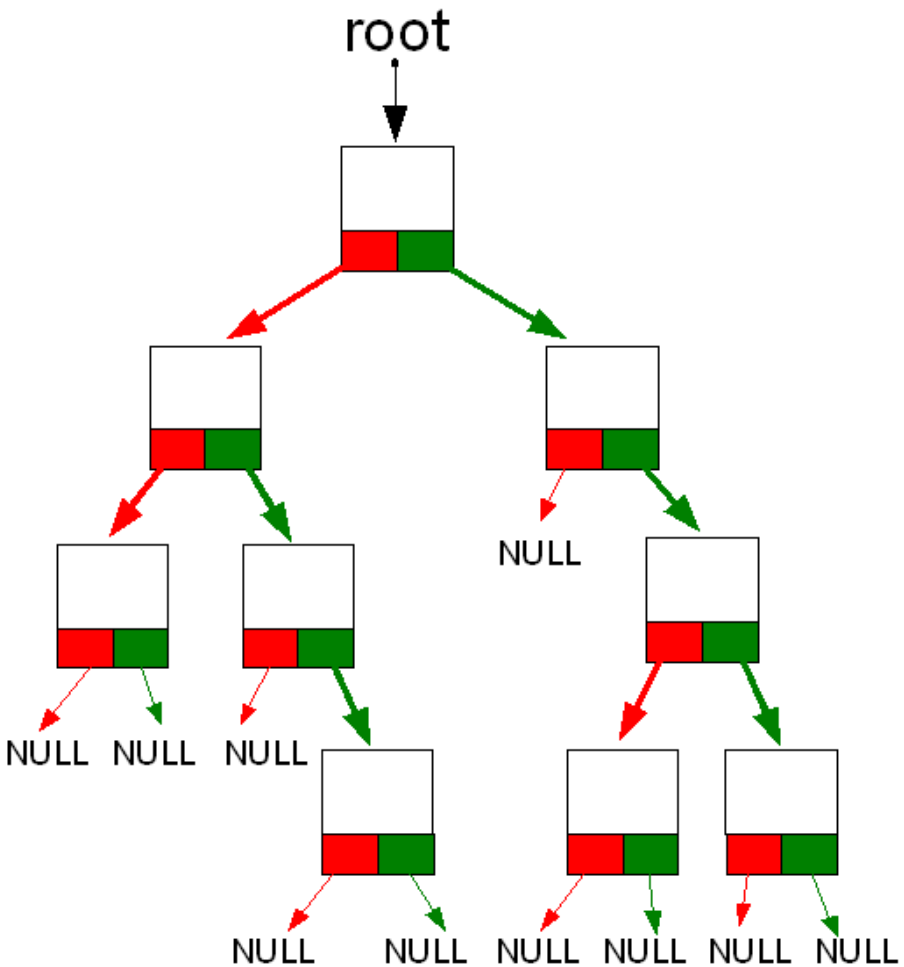


Lists



- Lists can be uni- and bidirectional
- Compared to arrays:
  - \+ insertion and removal of the middle elements is easy and quick
  - – no direct access to elements (time-consuming)

Trees

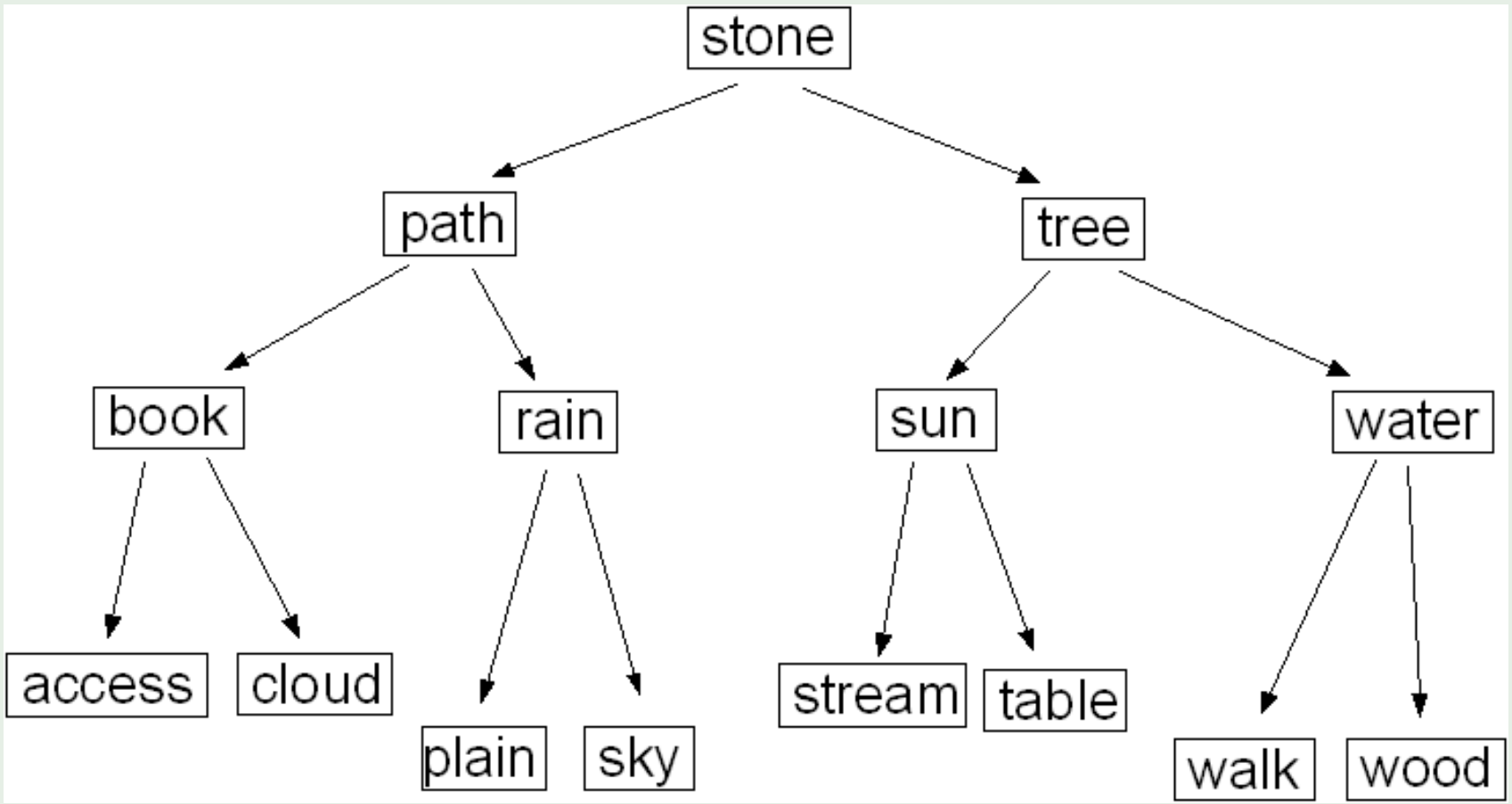


- Tree elements can be ordered in the natural way (all elements to the left are smaller, all elements to the right are larger)
- If the tree is kept balanced
  - \+ quick access
  - \+ quick insertion and removal

  Otherwise the tree can have properties similar to an unidirectional list.

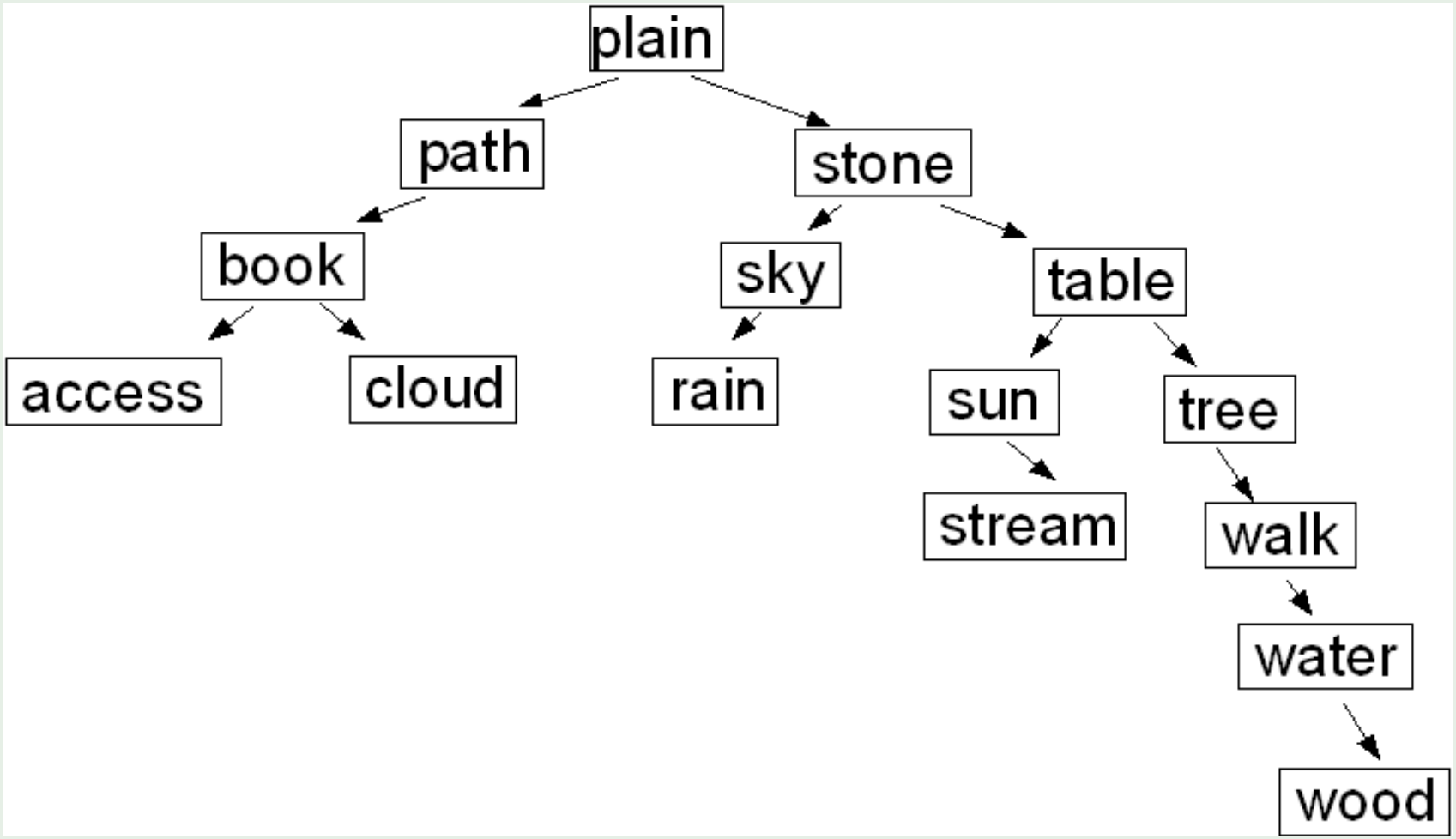
Trees

Word tree (ordered and balanced)



Trees

Word tree (ordered and mildly unbalanced)



Outline

Program arguments

After the command line (Dev C++: menu Execute→Parameters)

```
executable  0  1  text  [ENTER]
```

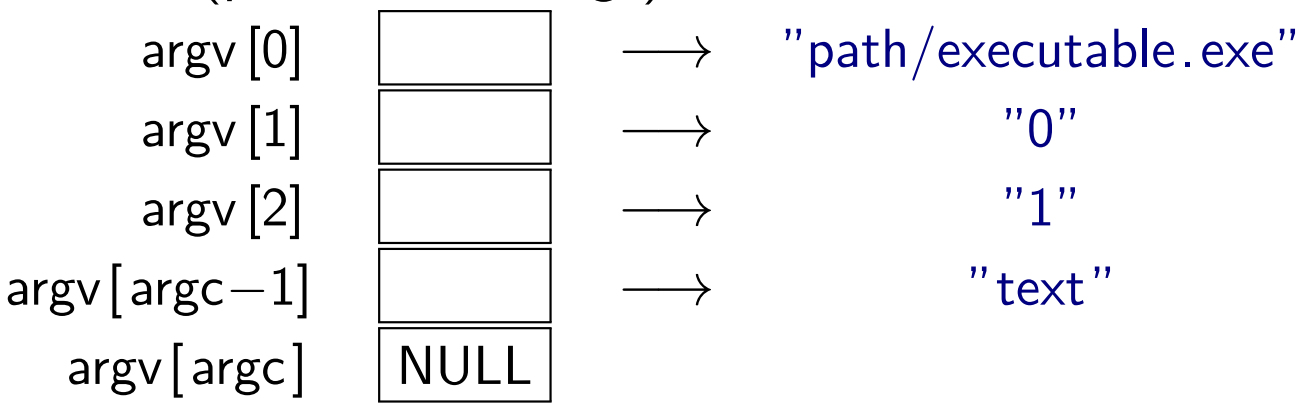
the arguments (0, 1, text) are passed via the arguments of the main() function:

```
int main(int argc, char *argv[]) {...}
```

argc equals to the total number of the typed arguments +1

```
argc == 4;
```

argv is a NULL-terminated array of pointers to the actual parameters (passed as strings)



Program arguments

```
int main(int argc, char *argv[]) {
    for(int i=0; argv[i]; ++i)
        cout <<"argument " <<i <<":\t" <<argv[i] <<endl;
    // Dev C++ users might add: system("pause");
    return 0;
}
```

Instead of the **for** loop, a **while** loop can be used:

```
    while(*argv)
        cout <<"argument:\t" <<*(argv++) <<endl;
```

Note that all arguments (numbers also) are passed alphanumeric (as strings, of the type **char** $*$). To get back the integers you can use the atoi(argv[i]) function (requires **#include**<cstdlib>).

Output redirection

- cout directs the output to the console (screen). It can be easily redirected to a text file by typing in the command line

  ```
  programName >file.txt    [ENTER]
  ```

  Your code stays exactly the same (no need for file streams, etc.).
- This is a feature of the operating system (shell) and has (almost) nothing to do with C/C++.
- The >file.txt is not passed to the main(...) function. Hence, redirecting can be only done from the command line[3].

[3] In Dev C++, menu Execute→Parameters will not work.

Outline

Homework 3 (15 points)

Arrays and pointers (Fibonacci numbers)

1. (4 points) Rework the code you obtained in HW2.2:
   - Define in the main() function a local array of a constant length.
   - Rewrite the fibo(**int** x0, **int** x1, **int** n) function so that it
     1. accepts the array as an additional parameter and
     2. fills the passed array with the first n Fibonacci numbers starting with x0, x1.
   - Pass the array you defined in the main() function to the fibo() function.
2. (3 points) Rework the code you produced in HW3.1:
   - Add a function printIntArray() so that it
     1. accepts an **int** array as a parameter and
     2. prints its contents.
   - Remove the printing code from the fibo() function.
   - Modify the main() function to use both fibo() and printIntArray().

Homework 3 (15 points)

Arrays and pointers (Fibonacci numbers)

3. (4 points) Rework the code you obtained in HW3.2 to use a dynamically-allocated array of a variable length (entered by the user) instead of a constant size local array.
4. (4 points) Modify the code you obtained in HW3.3 to accept three command line arguments `n`, `x0` and `x1`, instead of asking the user to enter them manually. For example, if the executable file is called `fibo`, the command line

   ```
   fibo 40 0 1 [ENTER]
   ```

   should print the first 40 Fibonacci numbers starting with 0, 1 and exit[4].

E-mail the answer and the source code to `ljank@ippt.pan.pl`.

[4] In Dev C++ you can use the menu 'Execute → Parameters' to specify the arguments instead of running the program from the command line.