02393 Programming in C++
Module 4: Data Types (Continued) and Libraries and Interfaces (Introduction)

Sebastian Mödersheim

September 19, 2016

Lecture Plan

#	Date	Topic
1	29.8.	Introduction
2	5.9.	Basic C++
3	12.9.	Data Types, Pointers
4	19.9.	Libraries and Interfaces; Containers
5	26.9.	
6	3.10.	Classes and Objects I
7	10.10.	Classes and Objects II
		Efterårsferie
8	24.10.	Classes and Objects III
9	31.10.	Recursive Programming
10	7.11.	Lists
11	14.11.	Trees
12	21.11.	Novel C++ features
13	28.11.	Summary
	5.12.	Exam

Recap: Enum, structs, and arrays

```
typedef enum {wood,stone} material;

typedef struct {
    int x,y;
    bool isWall;
    material type;
} field;

int main(){
    ...
    field playground[n][m];
    for (int i=0; i<n; i++){
        for (int j=0; j<m; j++){
            playground[i][j].x=i;
            playground[i][j].y=j;
            playground[i][j].isWall=(i==0||i==(n-1)||j==0||j==(m-1));
            if (playground[i][j].isWall)
                playground[i][j].type=stone;
            else
                playground[i][j].type=wood;
        }
    }
  ...
```

Recap

- **enum** is used for enumeration types
- **struct** is one of the core concepts: defining new types of data as record of existing types.
  - ★ Every entry in the record has a name and type.
  - ★ This is the basis for object-oriented programming later: ("just add functions")
- Arrays are also a universal concept. Note however in C++:
  - ★ Arrays range from [0] to $[n-1]$ when the size is n
  - ★ The size of the array is not stored with the array! (It is your responsibility to keep track if it.)
  - ★ If you access outside the boundaries of the array, the compiler will not stop you; this may produce hard-to-find errors!
  - ★ The size of an array cannot be changed.
  - ★ Passing arrays as function arguments can be tricky (more later).
- Next week: C++ offers a data-structure **vector** in the library that overcomes many of the problems with arrays.
  - ★ Usually a vector is preferable over an array!

Recap: Pointers

- A pointer is a variable which contains a memory address
- Access to, and manipulation of, pointers by a program allows some interesting applications:
  - ★ Great way to screw up your code! Use with care!
  - ★ Classic way (pre 90's) to implement call-by-reference
    - ▶ We discuss an example; for most applications use modern C++ call-by-reference.
  - ★ Dynamic memory allocation:
    - ▶ the program asks the system for more memory with **new**.
    - ▶ the system answers with a pointer to the memory block
    - ▶ must be explicitly given back with **delete**—there is no garbage collection.
  - ★ Based on dynamic memory: recursive data structures (in 3rd part of course).

Recap: Pointers

Definition

The address (for example of a variable) is a pointer value, which can be stored in memory and manipulated as data.

Declaring pointer variables:

```
int *p1, *p2, p3;
char *cptr;
```

Pointer operations

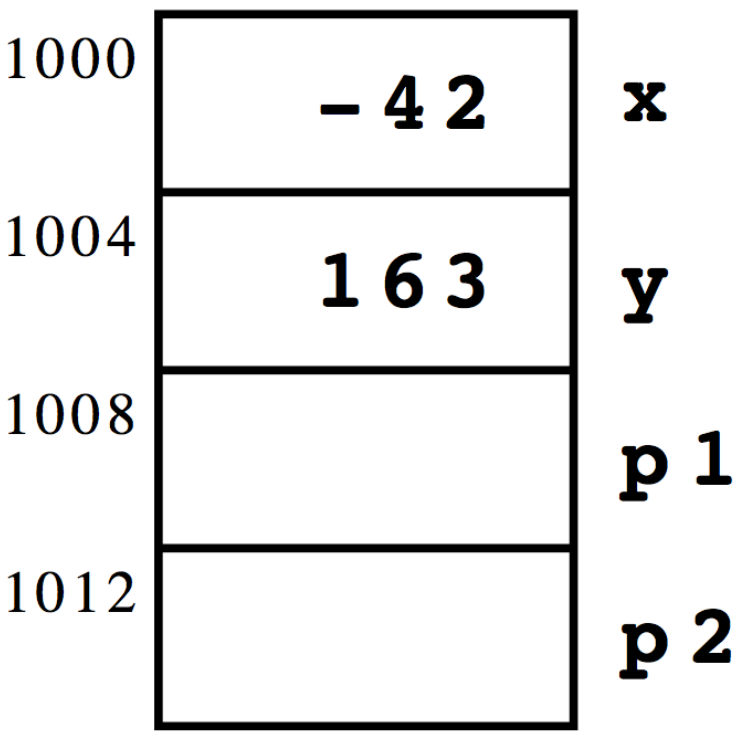
- &: address-of. Takes a variable and returns the corresponding memory address
- *: value-pointed-to, returns the variable, or the pointee , the pointer points to.

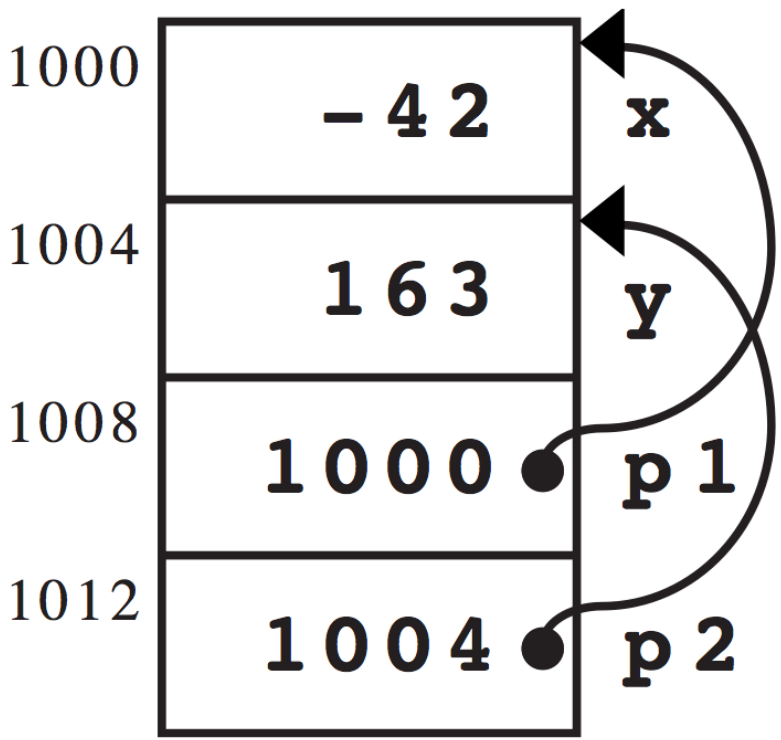
Pointer assignment

```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
```

Pointer assignment

```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
```



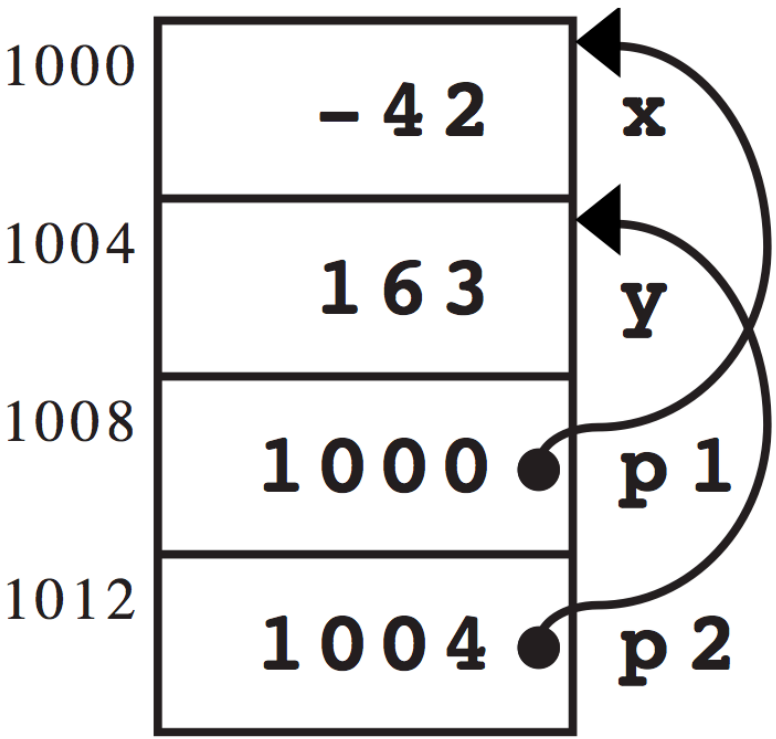


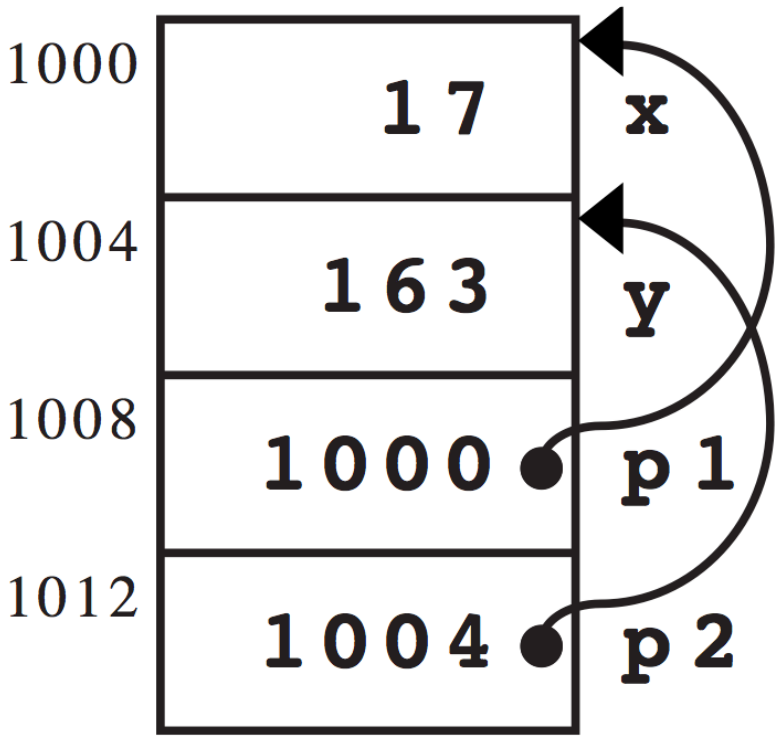
Pointer dereferencing

```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
```

Pointer dereferencing

```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
```



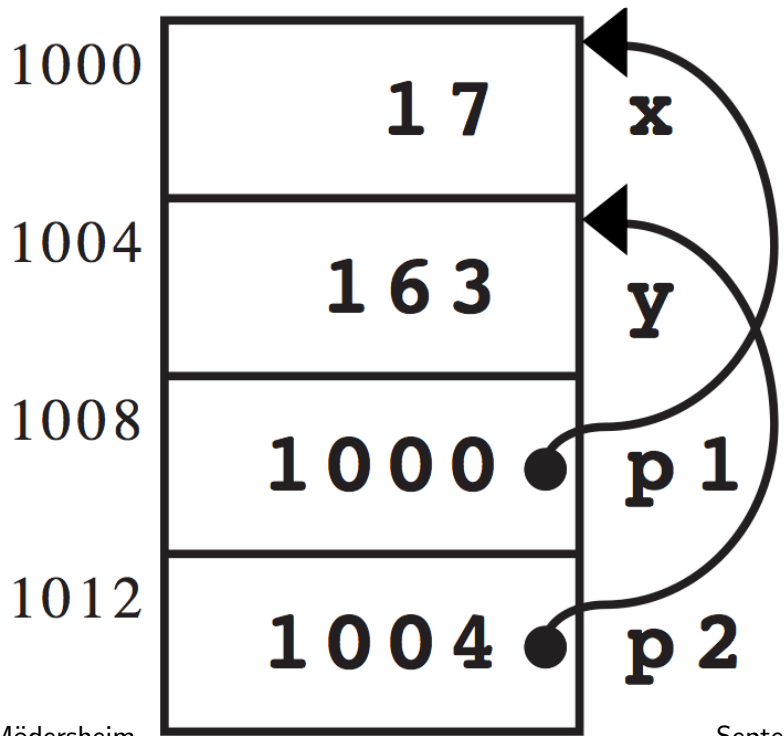


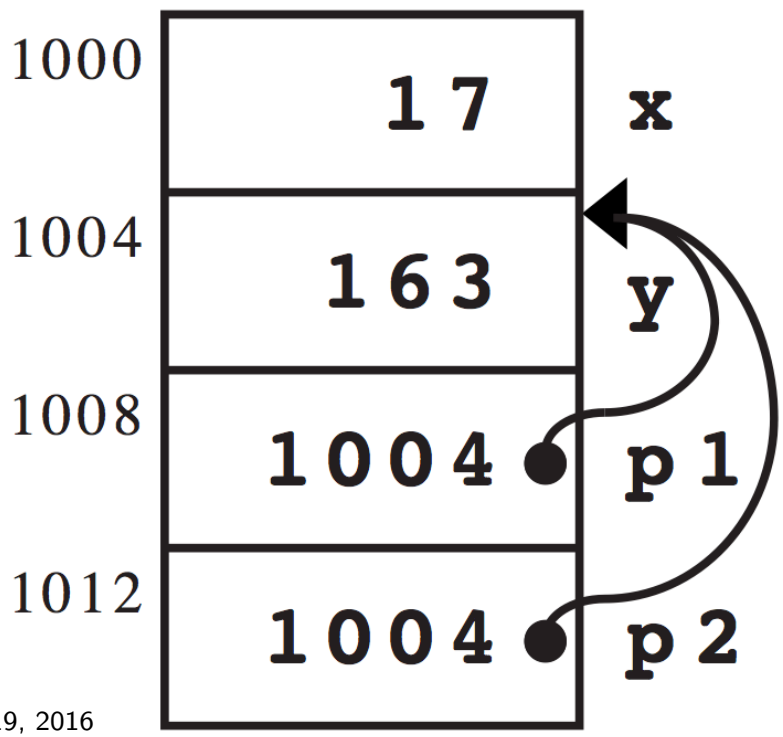
Pointer assignment

```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
p1 = p2;
```

Pointer assignment

```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
p1 = p2;
```





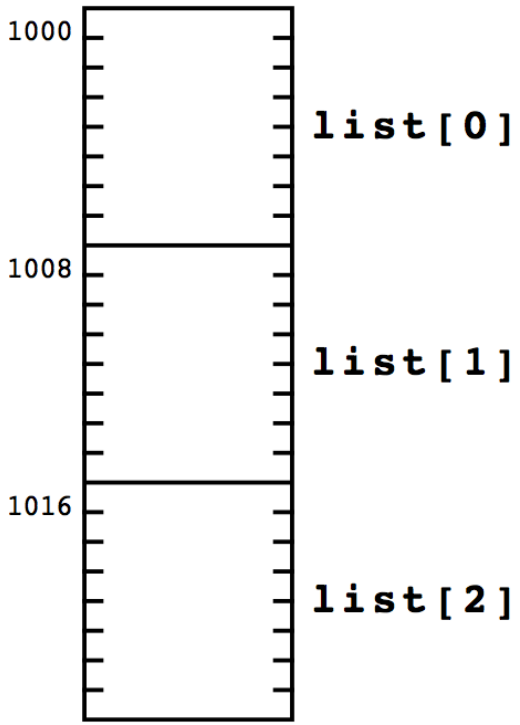
Summary of Pointers

- General hint: often diagrams (like on the previous slides) help to understand what is happening!
- $=$ pointer assignment between pointers: makes the first pointer point to the pointee of the second
- & address-of operator
- $*$ dereference operator, dereference a pointer to access its pointee: only works if there is a pointee
- Special null-pointer `nullptr`. Dereferencing gives null pointer error
- Pointers are distinguished by type of pointee: **int*** is not the same as **double***
- Increment and decrement are defined on pointers, but they are relative to the size of the data-type of the pointee.
- Technically a pointer is similar to a normal integral data type, but conceptually it is very different.
  Avoid type casting from **int*** to **int** and the like.

Pointers and arrays

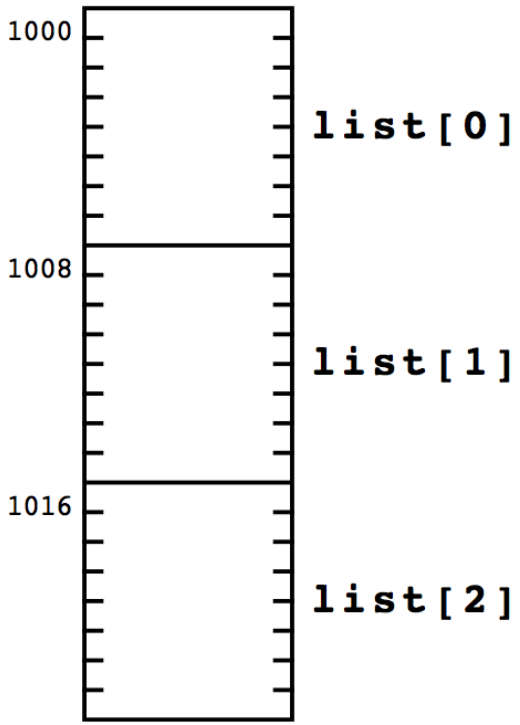
```
double list[3];
```

The allocated memory looks like:



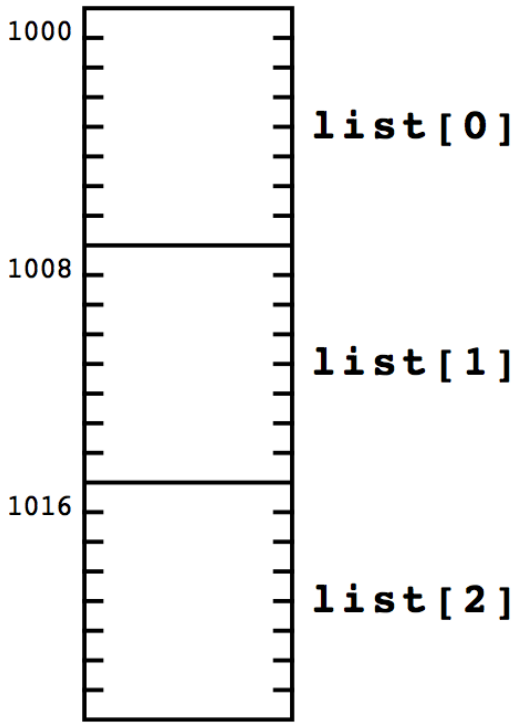
Pointers and arrays

The name of the array is treated as a pointer to the initial element



Pointers and arrays

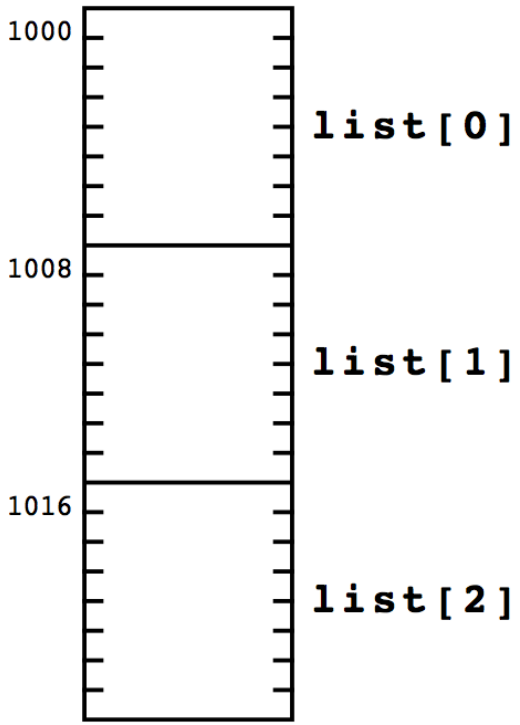
The name of the array is treated as a pointer to the initial element



list is the address of the first element, i.e. **&list[0]**

Pointers and arrays

The name of the array is treated as a pointer to the initial element



Thus, **list[i]** is just nice notation for ***(list+i)**

Passing arrays as parameters

```
double Mean(double array[], int n) {
  double total = 0;
  for (int i = 0; i < n; i++) {
    total += array[i];
  }
  return total / n;
}
```

- it is common to omit the maximum bound in the parameter declaration
- it will get a pointer to that array

```
mean = Mean(list, 5);
```

- thus the array argument will not be copied, and changes affect the value of `list`

Swap function: call-by-value, call-by-reference and pointers

```
void swap_classic(int *x, int *y){
  int tmp= *x;
  *x=*y;
  *y=tmp;
}
void swap_modern(...){
...



}
int main(){
  int x=5;
  int y=7;
  swap_classic(&x,&y);
  swap_modern(...);
}
```

Swap function: call-by-value, call-by-reference and pointers

```
void swap_classic(int *x, int *y){
  int tmp= *x;
  *x=*y;
  *y=tmp;
}
void swap_modern(int & x, int & y){
  int tmp= x;
  x=y;
  y=tmp;
}
int main(){
  int x=5;
  int y=7;
  swap_classic(&x,&y);
  swap_modern(x,y);
}
```

swap using call by value

```
void swap(int *xp, int *yp){
    int* z = xp;
    xp = yp;
    yp = z;
}
```

does not work! Why?

swap using pointers

How about:

```
void swap(int **xp, int **yp){
    int* z = *xp;
    *xp = *yp;
    *yp = z;
}
```

swap using pointers

How about:

```
void swap(int **xp, int **yp){
    int* z = *xp;
    *xp = *yp;
    *yp = z;
}
```

Is that useful?

swap using pointers

How about:

```
void swap(int **xp, int **yp){
    int* z = *xp;
    *xp = *yp;
    *yp = z;
}
```

Is that useful?

- Not for **int**, but for large data-structures:
  just swapping pointers, instead of the data-structure itself.

swap using pointers

How about:

```
void swap(int **xp, int **yp){
    int* z = *xp;
    *xp = *yp;
    *yp = z;
}
```

Is that useful?

- Not for **int**, but for large data-structures:
  just swapping pointers, instead of the data-structure itself.
- But it can be written more nicely:

```
void swap(int* &xp, int* &yp){
    int* z = xp;
    xp = yp;
    yp = z;
}
```

Libraries

There are many programming tasks have already been solved a thousand times! Do not re-invent the wheel!
For example, the standard template library (STL) of C++ has lot to offer:

- Vector: an alternative to arrays that can make live much easier!
- Other "containers": Map, Set, ...
- String: an alternative to char-arrays
- File I/O: can be used like cin/cout but with files
- Mathematical functions
- ...

Interfaces

- An interface is a conceptual boundary between the library and its clients
- Provides information for using the library
- Hides many the implementation details

Principles of good interface design

- unified: consistent abstraction with a unifying scheme
- simple: hide the complexity of the real implementation
- sufficient: clients do not need to read the implementations
- general: flexible to meet the needs of many different clients
- stable: the implementation could be changed, but not the interface

Interfaces in C++

- Usually in a header file, with file extension `.h`
- Header files are in C++ syntax, containing typically
  - ★ Function prototypes (not implementation!)
  - ★ Data type definitions
  - ★ (Later in OOP): class declarations
- The implementation of the functions (and classes) is in a corresponding implementation file, with extension `.cpp`