DTU

Before we start:

If you feel ill, go home
Keep your distance to others
Wash or sanitize your hands
Disinfect table and chair
Respect guidelines and restrictions

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

Lecturer:
Alceste Scalas

(Slides based on previous versions by Andrea Vandin, Alberto Lluch Lafuente, Sebastian Mödersheim)

22 September 2020

Lecture Plan

#	Date	Topic	Book chapter *
1	01.09	Introduction	
2	08.09	Basic C++	1
3	15.09	Data Types	2
4	22.09		
		Libraries and Interfaces	3
5	29.09		
6	06.10	Classes and Objects	4.1, 4.2 and 9.1, 9.2
		Autumn break	
7	20.10	Templates	4.1, 11.1
8	27.10	LAB DAY	Old exams
9	03.11	Inheritance	14.3, 14.4, 14.5
10	10.11	Recursive Programming	5
11	17.11	Linked Lists	10.5
12	24.11	Trees	13
13	01.12	Exercises & Summary	
	07.12	Exam	

* Recall that the book uses sometimes ad-hoc libraries that are slightly different with respect to the standard libraries (e.g., strings and vectors).

Recap: Enum, structs, and arrays

```
enum material {wood,stone};

struct field {
    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-
            if (playground[i][j].isWall)
                playground[i][j].type=stone;
            else
                playground[i][j].type=wood;
        }
    }
  ...
```

Recap

- **enum**: enumeration types
- **struct**: new types as records of existing types
  - ★ Every entry in the record has a name and type.
  - ★ The basis for object-oriented programming (later in the course)
- Arrays: collections of n values of a same type
  - ★ In C++, array elements are indexed from [0] to $[n-1]$
  - ★ The size of the array is not stored with the array!
  - ★ If you access outside the boundaries of the array, the compiler will not stop you; this may produce hard-to-find errors!
  - ★ Passing arrays as function arguments can be tricky (more later)

Recap

- **enum**: enumeration types

- **struct**: new types as records of existing types
  - ★ Every entry in the record has a name and type.
  - ★ The basis for object-oriented programming (later in the course)

- Arrays: collections of n values of a same type
  - ★ In C++, array elements are indexed from [0] to $[n-1]$
  - ★ The size of the array is not stored with the array!
  - ★ If you access outside the boundaries of the array, the compiler will not stop you; this may produce hard-to-find errors!
  - ★ Passing arrays as function arguments can be tricky (more later)

- **Next week:** C++ offers a standard data type called **vector** that overcomes many of the problems with arrays.
  - ★ Usually a vector is preferable over an array!

Pointers

- A pointer is a variable which contains a **memory address**

- Accessing and manipulating pointers allows for some interesting applications:
  - ★ Classic way (pre '90s) to implement "call-by-reference"
    - ▶ Don't copy values when calling functions; just pass a pointer
  - ★ 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 deallocated with **delete** — no garbage collection!
  - ★ Recursive data structures (later in the course)

Pointers are a common source of bugs! Use with care!

Pointers

Definition

A pointer is a memory address.

Declaring pointer variables:

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

Pointers

Definition

A pointer is a memory address.

Declaring pointer variables:

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

Pointer operations

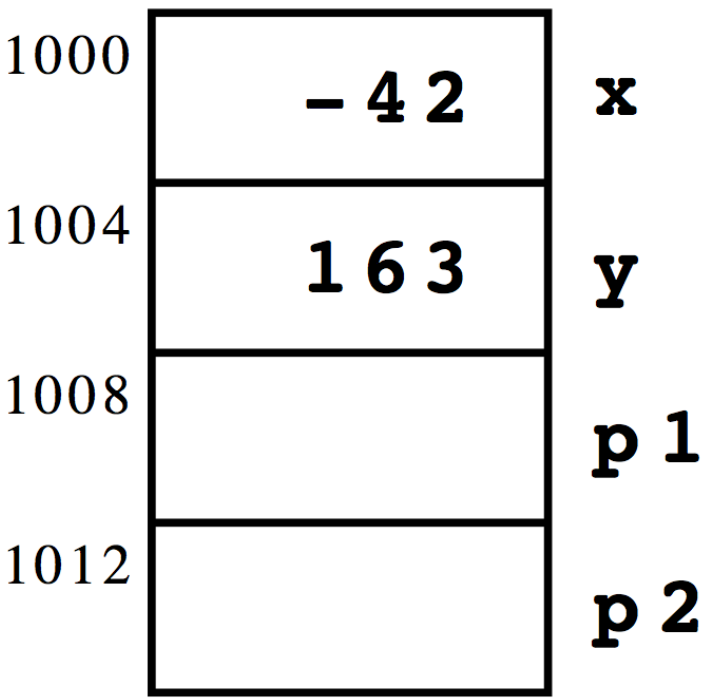
- &: **address-of**. For example: if `x` is a variable, then `&x` is the memory address where the value of `x` is stored
- $*$: **value-pointed-to**. For example: if `p` is a pointer, then `*p` is the value (a.k.a. pointee) stored at the memory address `p`

Pointer assignment

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

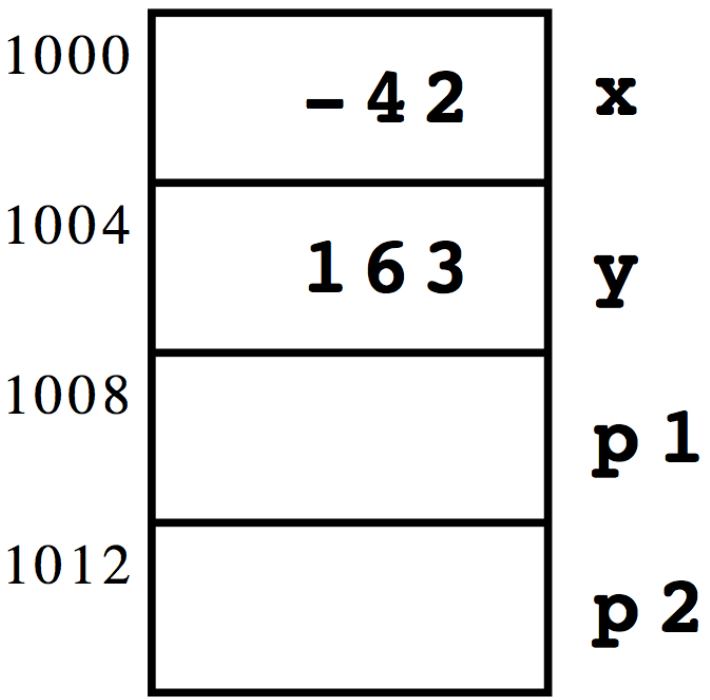
Pointer assignment

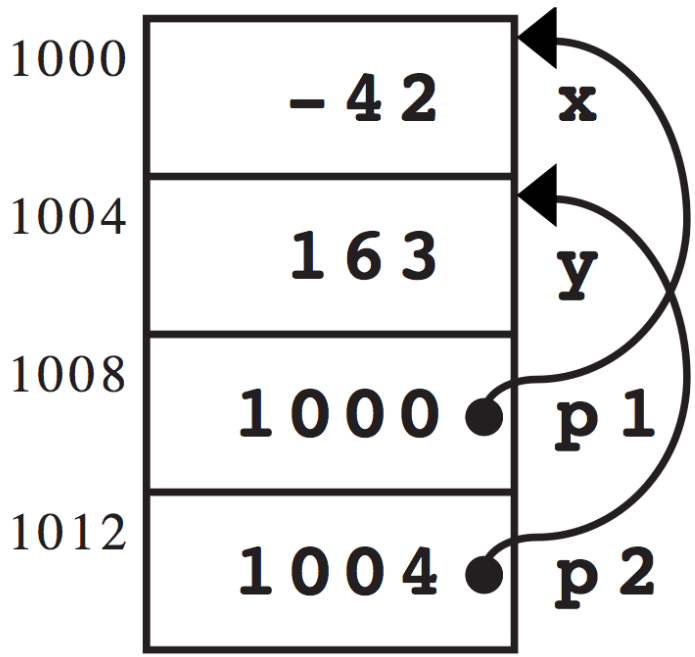
```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
```



Pointer assignment

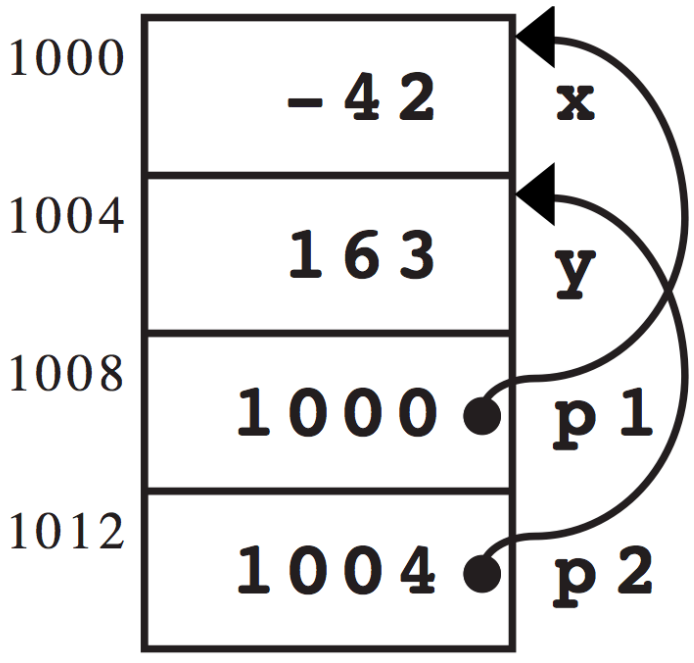
```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
```





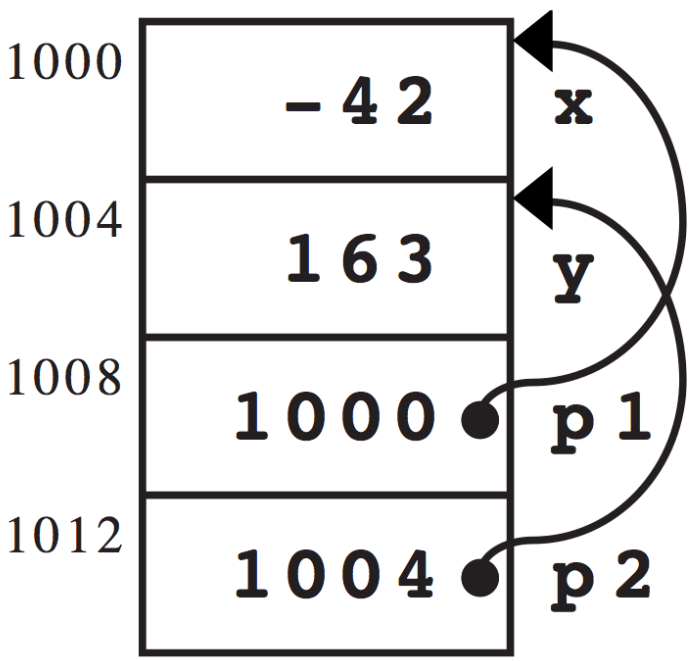
Pointer dereferencing

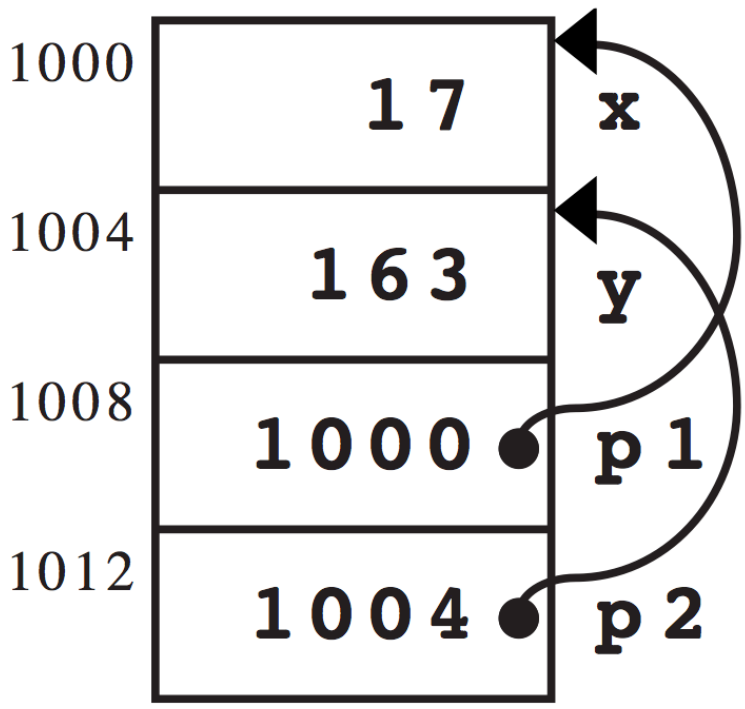
```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
```



Pointer dereferencing

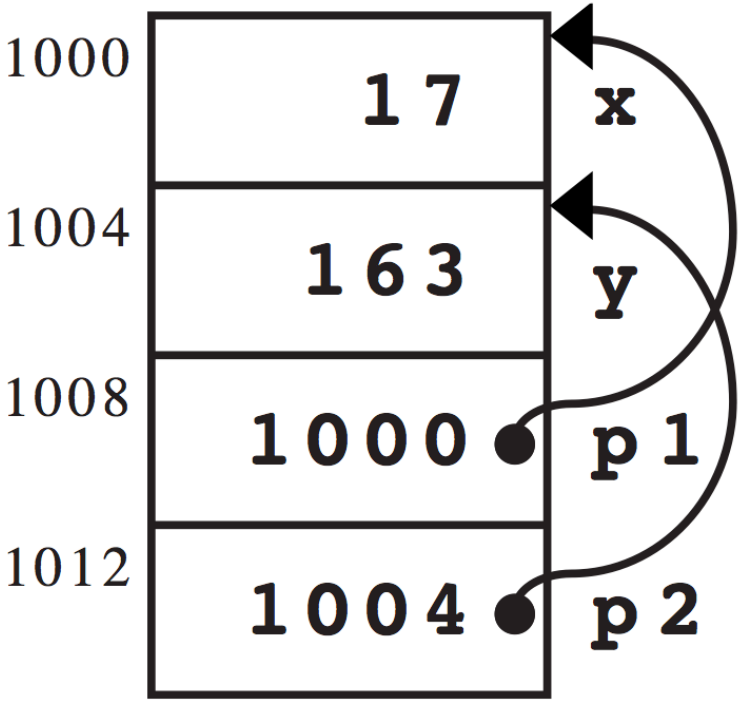
```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
```





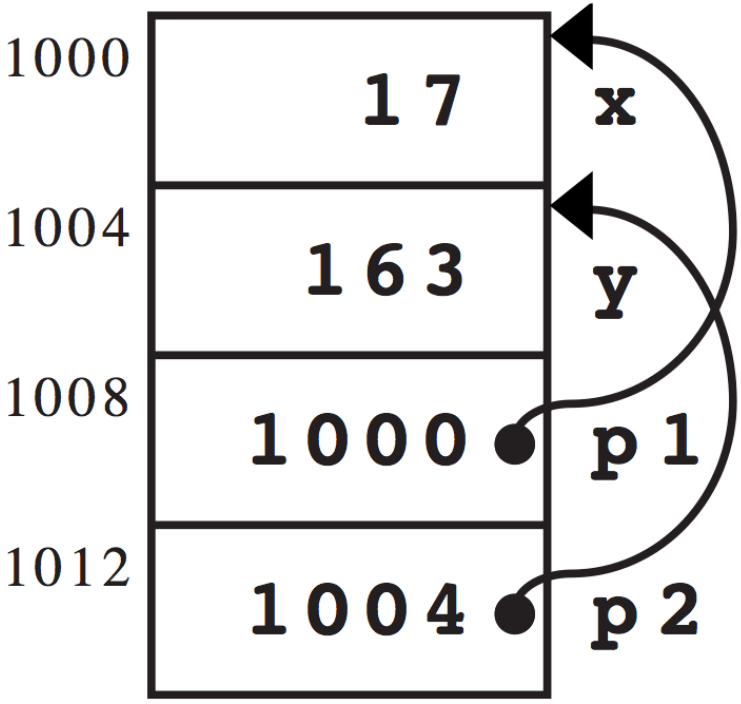
Pointer assignment

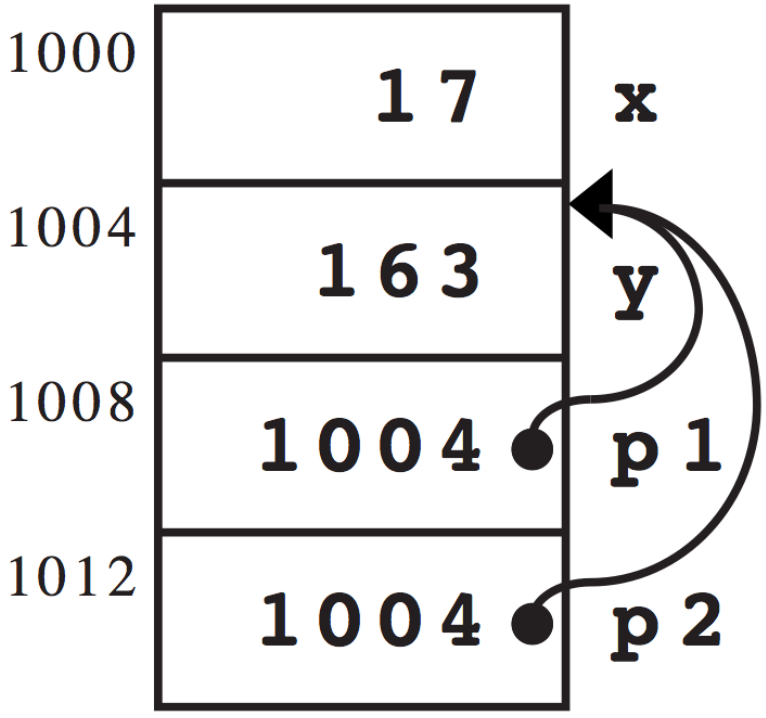
```
int x = -42, y = 163;
int *p1, *p2;
p1 = &x;
p2 = &y;
*p1 = 17;
p1 = p2;
```



Pointer assignment

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





Summary of Pointers

Suggestion: often diagrams (like the previous slides) help to understand what is happening!

- `=` pointer assignment: `p1 = p2` makes `p1` point to the same pointee of `p2`
- `&` address-of operator: gets the address of a variable
- `*` dereference operator: `*p` gets the pointee of `p`
- `nullptr` Special null-pointer. Dereferencing gives an error!
- Pointers are distinguished by type of pointee: **int*** is not the same as **double***

Live demo - pointers

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?

Live demo - Swap

Libraries

There are many programming tasks that have already been solved a thousand times! Do not re-invent the wheel!

The **standard template library (STL)** of C++ has lot to offer:

- **vector**: an easier alternative to arrays
- other **container** types: **map**, **set**, ...
- **string**: an alternative to arrays of `chars`
- file I/O: similar to `cin`/`cout` but with files
- mathematical functions
- ...

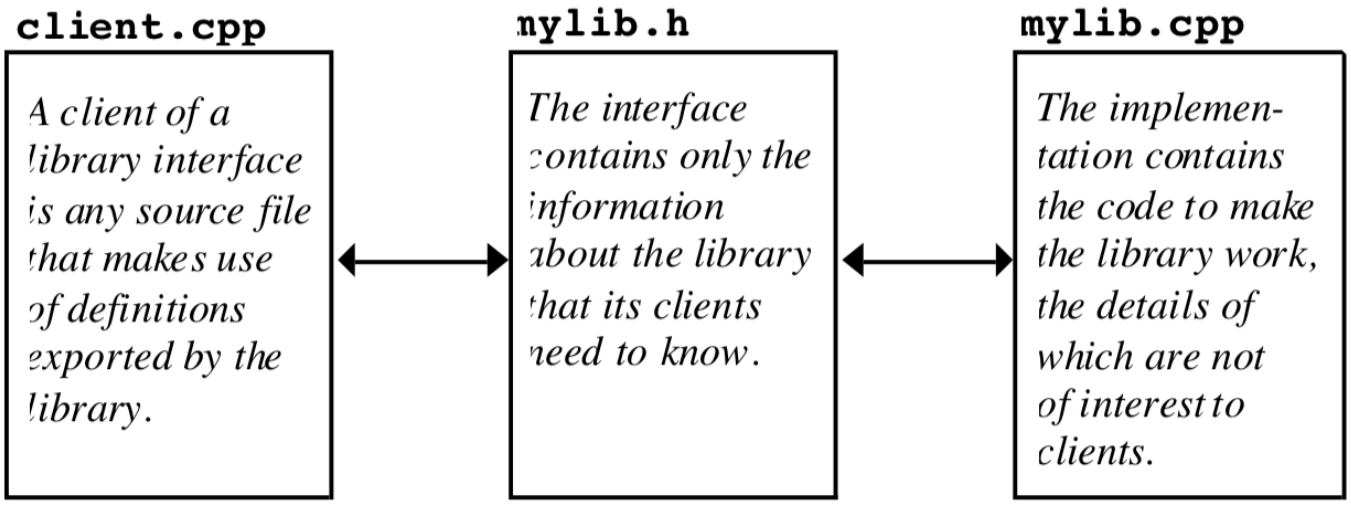
Interfaces

An interface is a "contract" between a library and its clients (users)

- Provides information for using the library
- Hides many the implementation details

Interfaces in C++

- Usually in a header file, with file extension `.h`
- Header files are just C++ files, but they only contain:
  - ★ Function prototypes (not implementations!)
  - ★ Type definitions (including class declarations, later in the course)
- The implementation of the functions (and classes) is in a corresponding implementation file, with extension `.cpp`



Live demo

DTU

Before we leave:

Disinfect table and chair
Maintain your distance to others
Wash or sanitize your hands
Respect guidelines and restrictions outside