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 1: Data Types

Lecturer:
Alceste Scalas

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

15 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).

Programming Assignments on CodeJudge

Some general remarks and suggestions:

- Use the **sample test data** to check your input/output
- If CodeJudge seems to refuse correct solutions...
  - ★ Check thoroughly the input and output of the tests
  - ★ Beware of imprecisions in the input/output (e.g. blank spaces)
  - ★ Beware of portability issues: try to write rock-solid code
    - e.g., **initialise variables before using them**
    - do not assume variables to be initialised to 0
  - ★ Use Piazza!

Last week's exercises (to hand in today at 17:00):

- See model solutions online
- Questions? Ask us later

Outline

Recap: Last Programming Session

- Bounded numerical types, e.g.:
  - ★ `int` $= [$`INT_MIN`$, \ldots,$ INT_MAX$] \subset \mathbb{N}$
  - ★ `unsigned int` $= [0, \ldots,$ UINT_MAX$] \subset \mathbb{Z}$
- Stack limits, e.g., recursion may crash
- Side effects vs. arithmetic axioms
- C++ functions = procedures $\neq$ pure mathematical functions
- First taste of functions, if/then, loops, variables, etc.

Outline

The hierarchy of data types

Atomic/Fundamental types

- booleans: **bool**
- characters: **char**
- integer numbers: [**unsigned**] [**long**] **int**
- floating point numbers: **float**, **double**, **long double**
- define your own: **enum**

⇒ `http://en.cppreference.com/w/cpp/language/types`

The hierarchy of data types

Atomic/Fundamental types

- booleans: **bool**
- characters: **char**
- integer numbers: [**unsigned**] [**long**] **int**
- floating point numbers: **float**, **double**, **long double**
- define your own: **enum**

⇒ `http://en.cppreference.com/w/cpp/language/types`

New types composed from the existing type

1. **struct** (or record): a collection of data values
2. **array**: sequence of data values of the same type
3. **pointer**: stores a memory address

Mixed data types, casting

What's the type of

- 9/6?
- 9.0/6?
- 9/6.0?
- 9/`int`(6.0), and `float`(9/6)?

Mixed data types, casting

What's the type of

- 9/6?
- 9.0/6?
- 9/6.0?
- 9/`int`(6.0), and `float`(9/6)?

If an operator is applied to operands of different types, the compiler converts the operands to a common type (if possible)

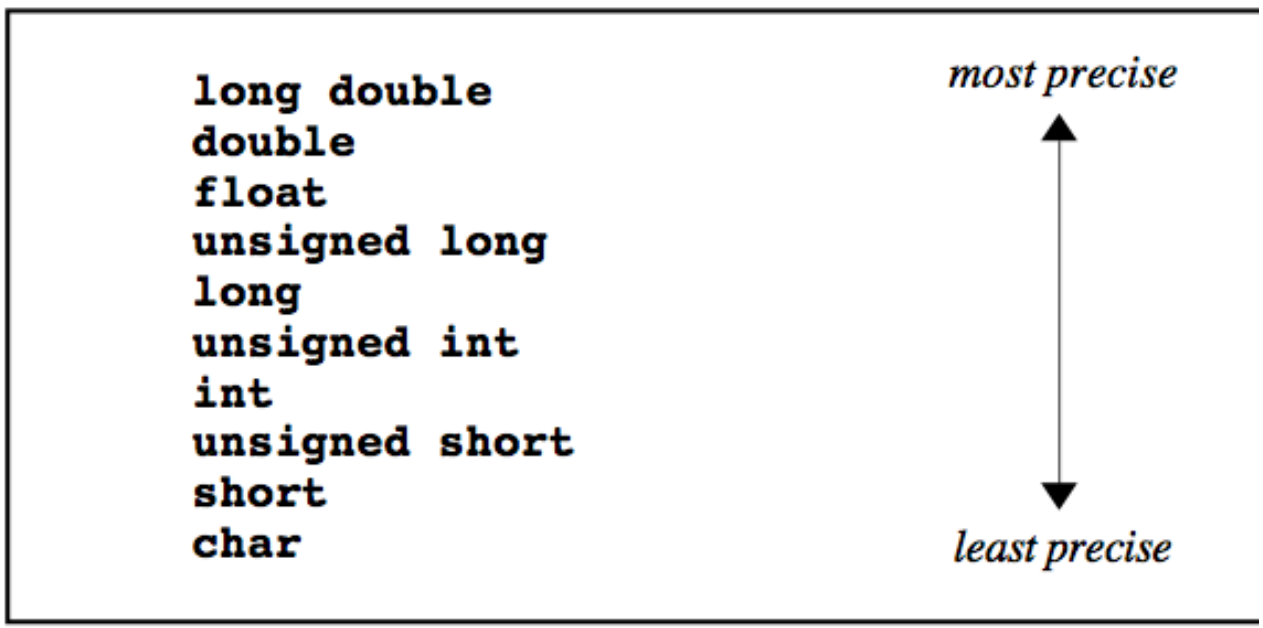
- The type that is more precise will be chosen
- The result type is always that of the arguments, after any conversions are applied

Mixed data types, casting

What's the type of

- 9/6?
- 9.0/6?
- 9/6.0?
- 9/`int`(6.0), and `float`(9/6)?

Table 1-5 Type conversion hierarchy for numeric types



Values of more precise types require more memory

Live Programming

Enum, structs, and arrays in a maze

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

Enum, structs, and arrays in a maze

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

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

Enum, structs, and arrays in a maze

```
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;
        }
    }
    ...
```

Remarks on these Data Types

- **enum** describes a type with user-specified values
  - ★ each allowed value has a descriptive name. . .
  - ★ . . . and corresponds to an integer

Remarks on these Data Types

- **enum** describes a type with user-specified values
  - ★ each allowed value has a descriptive name. . .
  - ★ . . . and corresponds to an integer
- **struct** defines a new type of data as a record of existing types
  - ★ Every entry in the record has a name and type
  - ★ Gives the basis for object-oriented programming (lecture 6)

Remarks on these Data Types

- **enum** describes a type with user-specified values
  - ★ each allowed value has a descriptive name...
  - ★ ...and corresponds to an integer
- **struct** defines a new type of data as a record of existing types
  - ★ Every entry in the record has a name and type
  - ★ Gives the basis for object-oriented programming (lecture 6)
- Arrays are collections of n values of a same type
  - ★ Arrays range from [0] to $[n-1]$
  - ★ **The size of the array is not stored with the array!**
    - ▶ It is your responsibility to keep track of it!
  - ★ The compiler lets you access **outside the array boundaries**
    - ▶ This may produce hard-to-find errors!
  - ★ Passing arrays as function arguments can be tricky (more later)

Remarks on these Data Types

- **enum** describes a type with user-specified values
  - ★ each allowed value has a descriptive name...
  - ★ ...and corresponds to an integer
- **struct** defines a new type of data as a record of existing types
  - ★ Every entry in the record has a name and type
  - ★ Gives the basis for object-oriented programming (lecture 6)
- Arrays are collections of n values of a same type
  - ★ Arrays range from [0] to $[n-1]$
  - ★ **The size of the array is not stored with the array!**
    - ▶ It is your responsibility to keep track of it!
  - ★ The compiler lets you access **outside the array boundaries**
    - ▶ This may produce hard-to-find errors!
  - ★ Passing arrays as function arguments can be tricky (more later)
- **Note:** C++ offers a type **vector** that is "better" than arrays
  - ★ We will talk about it in a few lectures

Outline

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 memory address (e.g., of a variable) is a pointer value, which can be stored in memory like "normal" data

Pointers

Definition

A memory address (e.g., of a variable) is a pointer value, which can be stored in memory like "normal" data

Declaring pointer variables:

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

Pointers

Definition

A memory address (e.g., of a variable) is a pointer value, which can be stored in memory like "normal" 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.

Live Programming

DTU

Before we leave:

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