ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

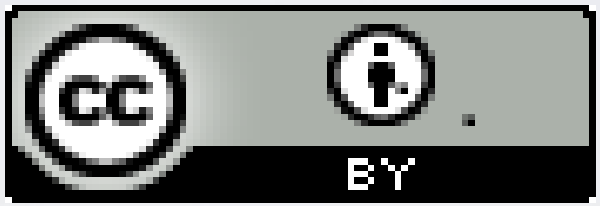
Διάλεξη 2: Αντικείμενα – Τύποι και Τιμές

Γ. Παπαγιαννάκης

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται στην άδεια χρήσης **Creative Commons** και ειδικότερα

Αναφορά Δημιουργού 3.0 - Μη εισαγόμενο
(Attribution 3.0– Unported)



- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Εισαγωγή στον Προγραμματισμό
Introduction to Programming

Lecture 2:
Objects, Types & Values

G. Papagiannakis

Abstract

- Most programming tasks involve manipulating data. Today, we will:
  - describe how to input and output data
  - present the notion of a variable for holding data
  - introduce the central notions of “Type” and “Type Safety”

Overview

- Strings and string I/O
- Integers and integer I/O
- Types and objects
- Type safety

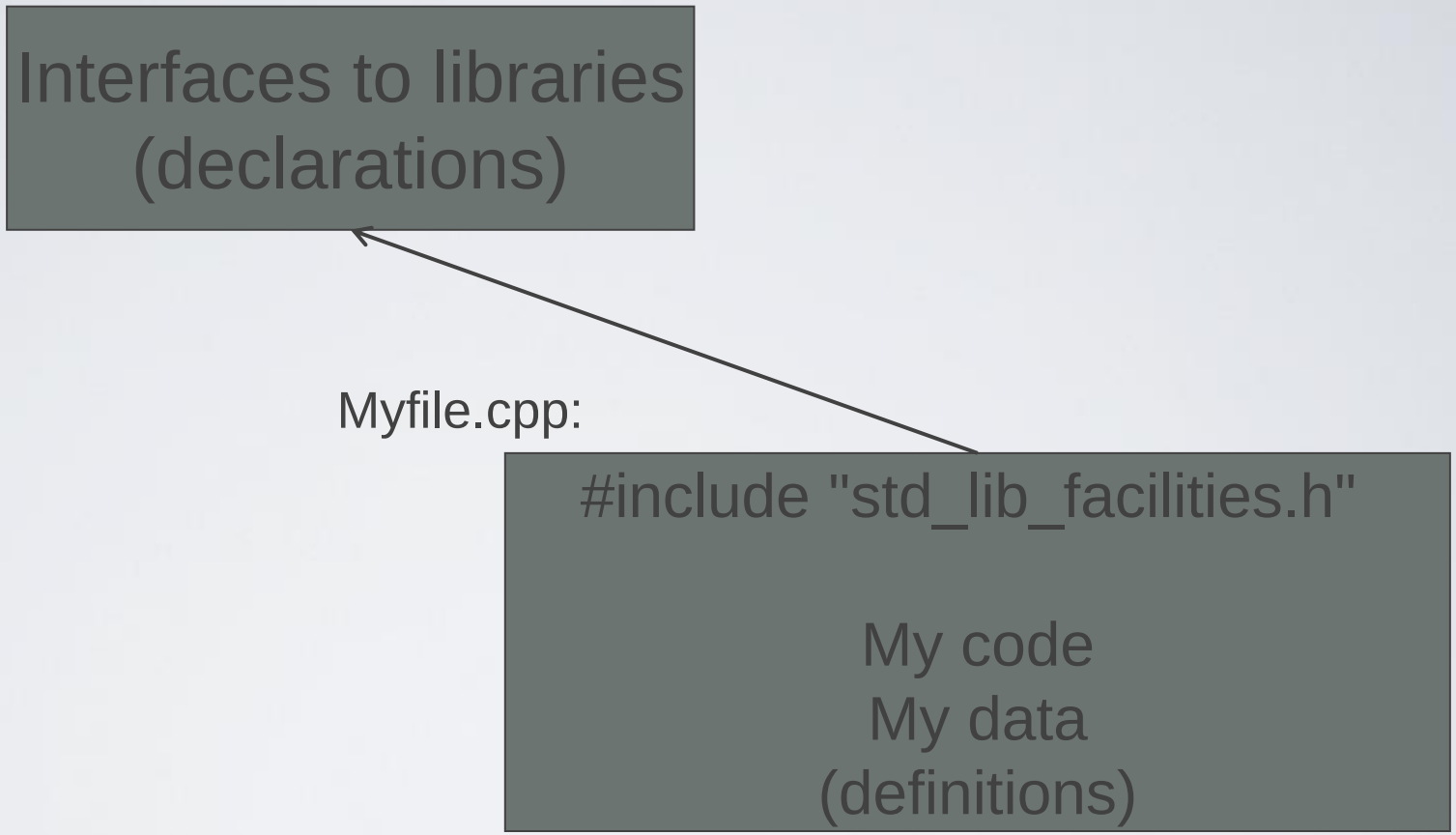
http://www.code.org

Input and output

```
// read first name:
#include "std_lib_facilities.h"                    // our course header

int main()
{
   cout << "Please enter your first name (followed " << "by 'enter'):\n";
   string first_name;
   cin >> first_name;
   cout << "Hello, " << first_name << “\n”;
}

// note how several values can be output by a single statement
// a statement that introduces a variable is called a declaration
// a variable holds a value of a specified type
// the final return 0; is optional in main()
// but you may need to include it to pacify your compiler
```

Source files

std_lib_facilities.h:



- "std_lib_facilities.h" is the header for our course

Input and type

- We read into a variable
  - Here, **first_name**
- A variable has a type
  - Here, **string**
- The type of a variable determines what operations we can do on it
  - Here, **cin>>first_name;** reads characters until a whitespace character is seen
  - White space: space, tab, newline, …

String input

```
// read first and second name:
int main()
{
    cout << "please enter your first and second names\n";
    string first;
    string second;
    cin >> first >> second;              // read two strings
    string name = first + ' ' + second;          // concatenate strings
                                                 // separated by a space
    cout << "Hello, "<< name << '\n';
}

// I left out the #include  "std_lib_facilities.h" to save space and
//  reduce distraction
// Don't forget it in real code
// Similarly, I left out the Windows-specific keep_window_open();
```

Integers

```
// read name and age:


int main()
{
   cout << "please enter your first name and age\n";
   string first_name;              // string variable
   int age;                        // integer variable
   cin >> first_name >> age;       // read
   cout << "Hello, " << first_name << " age " << age << '\n';
}
```

Integers and Strings

- Strings
  - **cin** >> reads (until whitespace)
  - **cout** << writes
  - + concatenates
  - += **s** adds the string **s** at end
  - ++ is an error
  - - is an error
  - …
- Integers and floating point numbers
  - **cin** >> reads a number
  - **cout** << writes
  - + adds
  - += **n** increments by the int **n**
  - ++ increments by **1**
  - - subtracts
  - …

The type of a variable determines which operations are valid and what their meanings are for that type

(that's called "overloading" or "operator overloading")

Names

- A name in a C++ program
  - Starts with a letter, contains letters, digits, and underscores (only)
    - **x, number_of_elements, Fourier_transform, z2**
    - Not names:
      - **12x**
      - **timetomarket**
      - **main line**
    - Do not start names with underscores: **_foo**
      - those are reserved for implementation and systems entities
  - Users can't define names that are taken as keywords
    - E.g.:
      - **int**
      - **if**
      - **while**
      - **double**

Names

- Choose meaningful names
  - Abbreviations and acronyms can confuse people
    - **mtbf, TLA, myw, nbv**
  - Short names can be meaningful
    - when used conventionally:
      - **x** is a local variable
      - **i** is a loop index
  - Don't use overly long names
    - Ok:
      - **partial_sum**
        element_count
        staple_partition
    - Too long:
      - **the_number_of_elements**
        remaining_free_slots_in_the_symbol_table

Simple arithmetic

```
// do a bit of very simple arithmetic:

int main()
{
    cout << "please enter a floating-point number: "; // prompt for a number
    double n;                                         // floating-point variable
    cin >> n;
    cout << "n == " << n
            << "\nn+1 == " << n+1                     // '\n' means “a newline”
            << "\nthree times n == " << 3*n
            << "\ntwice n == " << n+n
            << "\nn squared == " << n*n
            << "\nhalf of n == " << n/2
            << "\nsquare root of n == " << sqrt(n)    // library function
            << endl;                                  // another name for newline
}
```

A simple computation

```
int main()                  // inch to cm conversion
{
   const double cm_per_inch = 2.54;    // number of centimeters per inch
   int length = 1;                     // length in inches
   while (length != 0)                 // length == 0 is used to exit the program
   {                           // a compound statement (a block)
         cout << "Please enter a length in inches: ";
         cin >> length;
         cout << length << "in.  = "
               << cm_per_inch*length << "cm.\n";
   }
}
```

- A while-statement repeatedly executes until its condition becomes false

Types and literals

- Built-in types
  - Boolean type
    - **bool**
  - Character types
    - **char**
  - Integer types
    - **int**
      - **and short** and **long**
  - Floating-point types
    - **double**
      - and **float**
- Standard-library types
  - **string**
  - **complex<Scalar>**

- Boolean literals
  - **true false**
- Character literals
  - **'a', 'x', '4', '\n', '$'**
- Integer literals
  - **0, 1, 123, -6, 0x34, 0xa3**
- Floating point literals
  - **1.2, 13.345, .3, -0.54, 1.2e3, . 3F, .3F**
- String literals **"asdf"**,

  “Howdy, all y‘all!”
- Complex literals
  - **complex<double>(12.3,99)**
  - **complex<float>(1.3F)**

Types

- C++ provides a set of types
  - E.g. **bool**, **char, int, double**
  - **Called "built-in types"**
- C++ programmers can define new types
  - Called "user-defined types"
  - We'll get to that eventually
- The C++ standard library provides a set of types
  - E.g. **string**, **vector**, **complex**
  - Technically, these are user-defined types
    - they are built using only facilities available to every user

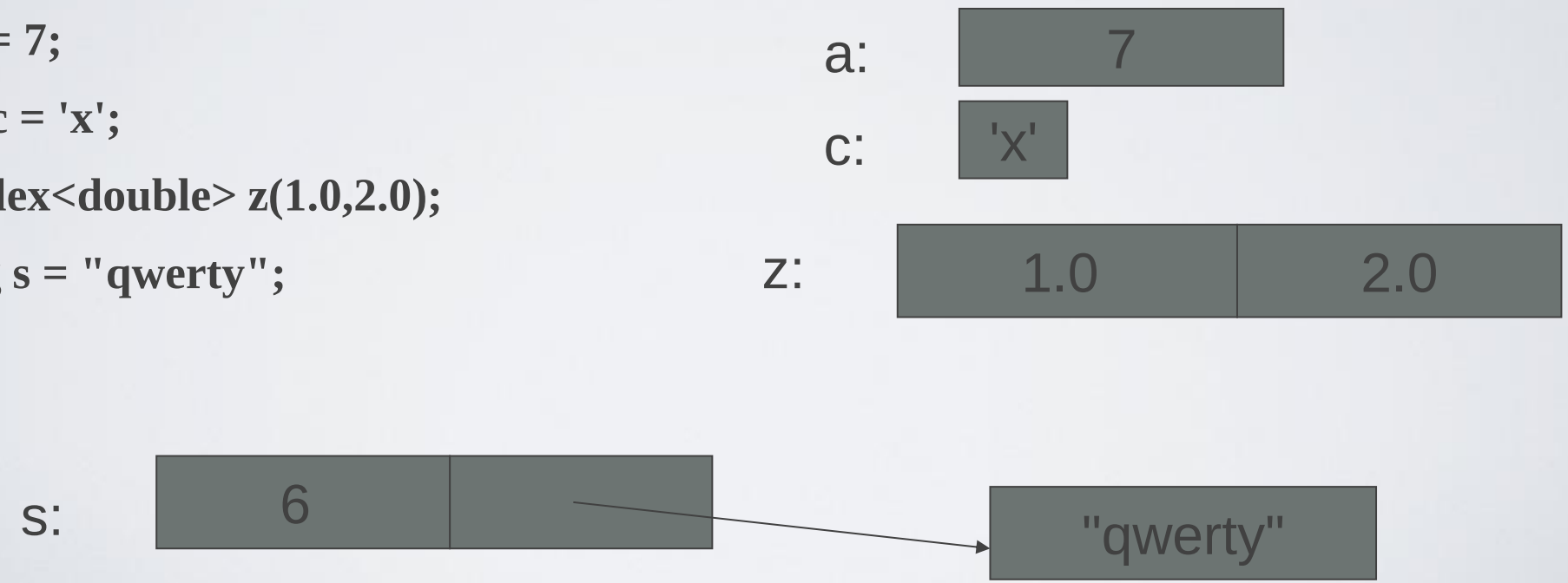
Declaration and initialization

```
int a = 7;

int b = 9;

char c = 'a';

double x = 1.2;

string s1 = "Hello, world";

string s2 = "1.2";
```

Objects

- An object is some memory that can hold a value of a given type
- A variable is a named object
- A declaration names an object

```
int a = 7;
char c = 'x';
complex<double> z(1.0,2.0);
string s = "qwerty";
```



Type safety

- Language rule: type safety
  - Every object will be used only according to its type
    - A variable will be used only after it has been initialized
    - Only operations defined for the variable's declared type will be applied
    - Every operation defined for a variable leaves the variable with a valid value
- Ideal: static type safety
  - A program that violates type safety will not compile
    - The compiler reports every violation (in an ideal system)
- Ideal: dynamic type safety
  - If you write a program that violates type safety it will be detected at run time
    - Some code (typically "the run-time system") detects every violation not found by the compiler (in an ideal system)

Type safety

- Type safety is a very big deal
  - Try very hard not to violate it
  - “when you program, the compiler is your best friend”
    - But it won’t feel like that when it rejects code you’re sure is correct
- C++ is not (completely) statically type safe
  - No widely-used language is (completely) statically type safe
  - Being completely statically type safe may interfere with your ability to express ideas
- C++ is not (completely) dynamically type safe
  - Many languages are dynamically type safe
  - Being completely dynamically type safe may interfere with the ability to express ideas and often makes generated code bigger and/or slower
- Most of what you’ll be taught here is type safe
  - We’ll specifically mention anything that is not

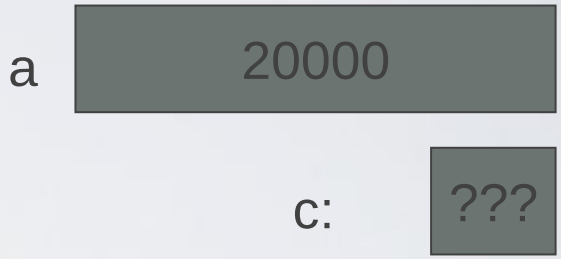
Assignment and increment

```
// changing the value of a variable
int a = 7;          // a variable of type int called a
                    // initialized to the integer value 7
a = 9;              // assignment: now change a's value to 9

a = a+a;            // assignment: now double a's value

a += 2;             // increment a's value by 2

++a;                // increment a's value (by 1)
```

a:

A type-safety violation
(“implicit narrowing”)

```
// Beware: C++ does not prevent you from trying to put a large value
// into a small variable (though a compiler may warn)

int main()
{
    int a = 20000;
    char c = a;
    int b = c;
    if (a != b)                    //  != means “not equal”
        cout << "oops!: " << a << "!=" << b << '\n';
    else
        cout << "Wow! We have large characters\n";
}
```



- Try it to see what value **b** gets on your machine

A type-safety violation (Uninitialized variables)

```
// Beware: C++ does not prevent you from trying to use a variable
// before you have initialized it  (though a compiler typically warns)

int main()
{
   int x;          // x gets a "random" initial value
   char c;         // c gets a "random" initial value
   double d;       // d gets a "random" initial value
                   //    – not every bit pattern is a valid floating-point value
   double dd = d;          // potential error: some implementations
                           // can't copy invalid floating-point values
   cout << " x: " << x << " c: " << c << " d: " << d << '\n';
}
```

- Always initialize your variables – beware: "debug mode" may initialize (valid exception to this rule: input variable)

A technical detail

- In memory, everything is just bits; type is what gives meaning to the bits

  (bits/binary) **01100001** is the int **97** is the char **'a'**

  (bits/binary) **01000001** is the int **65** is the char **'A'**

  (bits/binary) **00110000** is the int **48** is the char **'0'**

```
char c = 'a';
cout << c;      // print the value of character c, which is a
int i = c;
cout << i;      // print the integer value of the character c, which is 97
```

- This is just as in "the real world":
  - What does "42" mean?
  - You don't know until you know the unit used
    - Meters? Feet? Degrees Celsius? $s? a street number? Height in inches? …

About Efficiency

- For now, don't worry about "efficiency"
  - Concentrate on correctness and simplicity of code
- C++ is derived from C, which is a systems programming language
  - C++'s built-in types map directly to computer main memory
    - a **char** is stored in a byte
    - An **int** is stored in a word
    - A **double** fits in a floating-point register
  - C++'s built-in operations map directly to machine instructions
    - An integer + is implemented by an integer add operation
    - An integer = is implemented by a simple copy operation
  - C++ provides direct access to most of the facilities provided by modern hardware
- C++ help users build safer, more elegant, and efficient new types and operations using built-in types and operations.
  - E.g., **string**
  - Eventually, we'll show some of how that's done

A bit of philosophy

- One of the ways that programming resembles other kinds of engineering is that it involves tradeoffs.
- You must have ideals, but they often conflict, so you must decide what really matters for a given program.
  - Type safety
  - Run-time performance
  - Ability to run on a given platform
  - Ability to run on multiple platforms with same results
  - Compatibility with other code and systems
  - Ease of construction
  - Ease of maintenance
- Don't skimp on correctness or testing
- By default, aim for type safety and portability

Another simple computation

```
// inch to cm and cm to inch conversion:

int main()
{
    const double cm_per_inch = 2.54;
    int val;
    char unit;
    while (cin >> val >> unit) {        // keep reading
        if (unit == 'i')                // 'i' for inch
            cout << val << "in == " << val*cm_per_inch << "cm\n";
        else if (unit == 'c')           // 'c' for cm
            cout << val << "cm == " << val/cm_per_inch << "in\n";
        else
            return 0;                   // terminate on a "bad unit", e.g. 'q'
    }
}
```

Things to remember

- Input and output in C++
  - cout, cin
- Integers and strings
- Types and literals
- Declaration and initialization
- Objects
- Type safety

The next lecture

- Will talk about expressions, statements, debugging, simple error handling, and simple rules for program construction
- Read Chapter 3

Acknowledgements

Bjarne Stroustrup

Programming -- Principles and Practice Using C++

http://www.stroustrup.com/Programming/

Thank you!

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης