ΗΥ352 : ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

5ο ΦΡΟΝΤΙΣΤΗΡΙΟ

ΔΙΔΑΣΚΟΝΤΑΣ
Αντώνιος Σαββίδης

Περιεχόμενα

- C++ type casting
- C++ strings
- Preprocessor directives

C++ type casting – Implicit Conversions (1/2)

- Πραγματοποιούνται αυτόματα από τον compiler
- Τέτοια conversion είναι τα:
  - Integral promotions (e.g. enum to int, int to unsigned int)
  - Floating point conversions (e.g. float to double)
  - Floating-integral conversions (e.g. int to float)
  - Pointer conversions (derived class to base class only)
  - Reference conversions (derived class to base class only)
  - User defined conversions
    - Constructor conversions
    - Operator conversions

C++ type casting – Implicit Conversions (2/2)

- Παραδείγματα user defined conversions

```
struct X {
    int x;
    X(int x) : x(x) {}
};
struct Y {
    int y;
    Y(const X& x) : y(x.x) {}
};
X x(3);  // ok, x can be converted to
Y y = x; // y through Y::Y(const X&)
```

```
struct Y {
    int y;
    Y(int y) : y(y) {}
};
struct X {
    int x;
    X(int x) : x(x) {}
    operator Y() const { return Y(x); }
};
X x(3);  // ok, x can be converted to
Y y = x; // y through X::operator Y
```

```
struct File {
   bool IsOk(void) const;
   operator bool() { return IsOk(); }
   File(const char* path);
};
File f("foo.txt");
if (f) { /*some code here */ }
```

Γενικά, οι operators για conversion είναι ειδικές member συναρτήσεις ***χωρίς ορίσματα***, με όνομα ***operator Type*** και επιστρεφόμενη τιμή *τύπου* ***Type*** (χωρίς όμως αυτό να εμφανίζεται στο πρωτότυπο της συνάρτησης)

C++ type casting – Explicit Conversions (1/2)

- Πραγματοποιούνται από τον προγραμματιστή
- Για να κάνουμε cast την έκφραση ***expr*** σε τύπο ***T***
  - Old C-style casts
    - (T) expr
    - T (expr)
  - Οι νέοι operator για casting της C++
    - *static_cast*<T>(expr)
    - *dynamic_cast*<T>(expr)
    - *const_cast*<T>(expr)
    - *reinterpret_cast*<T>(expr)

C++ type casting – Explicit Conversions (2/2)

- Γιατί να χρησιμοποιήσουμε τους νέους operators για casting;
  - Βελτιωμένο συντακτικό
    - Έχουν πιο ξεκάθαρο συντακτικό, οπότε είναι ευκολότερο να τα καταλάβουμε και να τα βρούμε μέσα στον κώδικα.
  - Βελτιωμένα semantics
    - Είναι πλέον εμφανές ποιος είναι ο στόχος του type cast κι έτσι ο compiler μπορεί να αναγνωρίσει κάποια λανθασμένα casts.
  - Type-safe conversions
    - Κάποια casts μπορούν να πραγματοποιηθούν με ασφάλεια και σε runtime και ο προγραμματιστής μπορεί να ελέγξει αν το cast ήταν επιτυχημένο ή όχι.

C++ type casting – static_cast

- ***static_cast*<T> (expr)**
- Κάνει μετατροπές μεταξύ pointer και σχετιζόμενων κλάσεων (base-to-derived και derived-to-base).
- Δε γίνεται έλεγχος για την ασφάλεια της μετατροπής των τύπων κατά το runtime.
- Ο προγραμματιστής πρέπει να εξασφαλίσει ότι η μετατροπή είναι ασφαλής.

```
struct Base { int x; };
struct Derived: public Base { int y; };
Base* a = new Base;
Base* b = new Derived;
Derived* c = static_cast<Derived*>(a); // ok, but unsafe
Derived* d = static_cast<Derived*>(b); // ok
```

C++ type casting – dynamic_cast

- ***dynamic_cast*<T> (expr)**
- Εξασφαλίζει ότι το αποτέλεσμα του cast είναι valid object της κλάσης **T**.
- Μπορεί να χρησιμοποιηθεί μόνο με pointers και references σε objects.
- Αν αποτύχει σε cast pointer επιστρέφει **nullptr** για να δείξει το λάθος.
- Αν αποτύχει σε cast reference πετάει **bad_cast** exception.
- Χρειάζεται να είναι ενεργό το Run-Time Type Information (RTTI) για να λειτουργήσει.

```
class Base { };
class Derived : public Base { };

Base b; Base* pb;
Derived d; Derived* pd;

// ok: derived-to-base
pb = dynamic_cast<Base*>(&d);

// error: base-to-derived with
// base non-polymorphic
pd = dynamic_cast<Derived*>(&b);
```

```
struct Base { virtual void dummy(void) {} };
struct Derived: public Base {};

Base* pbd = new Derived;
Base* pbb = new Base;
Derived* pd;

// ok: derived-to-derived, returns pbd
pd = dynamic_cast<Derived*>(pbd);
if (pd == 0) cout << "Null on first cast" << endl;

// wrong but safe: base-to-derived, returns nullptr
pd = dynamic_cast<Derived*>(pbb);
if (pd == 0) cout << "Null on second cast" << endl;
```

C++ type casting – const_cast

- ***const_cast*<T> (expr)**
- Αφαιρεί ή προσθέτει το "const-ness" ή το "volatile-ness" από ένα τύπο.

```
void f(double& d);
void g(const double& d) { f(d); } // compile error: d is a const ref
void g(const double& d) { f(const_cast<double&>(d)); }   // ok

class X {
  int count;

  void f(void) const { ++count; }  // compile error: const function

  void f(void) const {
    X* instance = const_cast<X*>(this); // ok, cast object isn't const
    ++instance->count;                  // and can change its data
  } // Of course we could have declared count as mutable
};
```

C++ type casting – reinterpret_cast

- ***reinterpret_cast*<T> (expr)**
- Μετατρέπει ένα τύπο pointer σε οποιοδήποτε άλλο, ακόμα και σε ασυσχέτιστες κλάσεις.
- Επίσης μετατρέπει pointers σε integer types.
  - Η μόνη εγγύηση είναι ότι αν ο integer type στον οποίο κάνουμε cast είναι αρκετά μεγάλος για να χωρέσει τον pointer, τότε μπορούμε με ασφάλεια να κάνουμε cast πίσω στον pointer.
- Δε γίνεται καθόλου type checking, ούτε στο περιεχόμενο που δείχνουμε ούτε στον τύπο του pointer.
- Η μόνη ουσιαστική χρήση του είναι για low-level operations σε σημεία όπου η αναπαράσταση είναι system-specific και άρα non-portable.

```
struct A {};
struct B { int x; };
A* a = new A;
B* b = reinterpret_cast<B*>(a); // valid, but unsafe to dereference b
```

C++ type casting – typeid (1/2)

- ***typeid* (expr)**
- Ελέγχει τον τύπο ενός expression.
- Επιστρέφει ένα reference σε const object τύπου **type_info** (που δηλώνεται στο **<typeinfo>**).
- Μπορεί να χρησιμοποιηθεί με τους τελεστές **==** και **!=** για τον έλεγχο μεταξύ δύο τύπων ή να δώσει το όνομα του τύπου σαν αλφαριθμητικό χρησιμοποιώντας τη μέθοδο **name()**.
- Απαιτείται να είναι ενεργό το Run-Time Type Information (RTTI).

```
#include <iostream>
#include <typeinfo>
using namespace std;
int* a = 0, b = 0;
if (typeid(a) != typeid(b)) {
  cout << "a and b are of different types: "
       << "a: " << typeid(a).name() << ", b: " << typeid(b).name() << endl;
  // Prints: a and b are of different types: a: int*, b: int
}
```

C++ type casting – typeid (2/2)

- Όταν χρησιμοποιείται σε pointers δίνει τον τύπο του ίδιου του pointer και όχι του object στο οποίο «δείχνει».
- Όταν χρησιμοποιείται με objects δίνει τον πραγματικό (δυναμικό τύπο), δηλαδή το most derived complete object.

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

struct Base { virtual void f(){} };
struct Derived : public Base {};

int main () {
  Base* a = new Base; Base* b = new Derived;
  cout << "a is: " << typeid(a).name() << ", b is: " << typeid(b).name() << endl;
  //prints a is: class Base*, b is: class Base*
  cout << "*a is:" << typeid(*a).name()<< ", *b is:" << typeid(*b).name() << endl;
  //prints *a is: class Base, *b is: class Derived
  return 0;
}
```

C++ Strings (1/3)

- Τα C strings (**char***) είναι δύσκολο να τα χειριστεί κανείς σωστά, γιατί είναι επιρρεπή σε λάθη και σε buffer overrun.
- Τα C++ strings μας παρέχουν ασφάλεια και επιπλέον ευκολία με τις πολλές συναρτήσεις και μεθόδους που προσφέρουν.
- Ορίζονται στο standard header ***<string>*** *και* ανήκουν στο **namespace std**.
  - `using namespace std; string s;`
  - `std::string str("this is a string");`
  - `std::string x = "this is another string";`

C++ Strings (2/3)

```
#include <string>
#include <iostream>
using namespace std;
int main () {
    string str = "initial value", str2 = "str2 value";
    cin >> str >> ws;  // read from keyboard until first whitespace
    getline(cin, str); // read until the end of line
    printf("%s\n", str.c_str()); //c_str() gives a null-terminated string

    str = 'a';                  // assignment operators: str is now "a"
    str = str2;                 // str is now "str2 value"
    str[3] = '1';               // index operator: str is now "str1 value"
    cout << str + str2;         // concatenate using overloaded operator +

    if (str == str2)            // value equality check (also with !=)
        str.append(str2);       // str2 appended at end of str
    else if (str > str2)        // lexicographic ordering (also >=, <, <=)
        str.clear();            // str is now empty

    return 0;
}
```

C++ Strings (3/3)

```
#include <string>
#include <iostream>
#include <cassert>

using namespace std;

int main () {
    string str = "blabla";
    string s = str.substr(1, 3);     // substring from pos 1 with length 3
    cout << s.size() << s.length(); // s is "lab" and has size (length) 3
    assert(!s.empty());              // s isn't "" so it's not empty

    int i = s.compare(str);          // i = strcmp(s.c_str(), str.c_str());
    string::size_type pos;           // unsigned integer type (size_t)
    pos = str.find('b', 1);          // first occurance of 'b' from 1: 2
    assert(pos != string::npos);     // npos is unsigned -1 (invalid pos)

    pos = str.rfind('b');            // index of last occurance of 'b': 2
    pos = str.find("bla");           // first occurance of "bla": 0
    return 0;
}
```

Preprocessor directives (1/6)

- Οι γραμμές που ξεκινούν με # δεν είναι μέρος των εντολών του προγράμματος, αλλά αποτελούν οδηγίες προς τον preprocessor.
- Σε κάθε γραμμή μπορούμε να έχουμε μια το πολύ τέτοια οδηγία.
  - Αν θέλουμε να συνεχίσουμε στην επόμενη γραμμή, βάζουμε στο τέλος της γραμμής ένα backslash **('\')**
  - Δε βάζουμε semicolon στο τέλος των οδηγιών.
- Οι οδηγίες αυτές εκτελούνται από τον preprocessor ***πριν*** το compilation.

Preprocessor directives (2/6)

- Macro definitions
  - **`#define identifier replacement`**
    - **`#define SIZE 100`**
      `int table[SIZE];`
  - **`#define macro(arg1, …, argn) f(arg1, …, argn)`**
    - **`#define max(a, b)  a > b ? a : b`**
      `max(4, 5)`
- Ο preprocessor αντικαθιστά κάθε εμφάνιση του **identifier** με το **replacement**, π.χ. του **max(a, b)** με το **a > b ? a : b**.
- Ο preprocessor δεν καταλαβαίνει C++, κάνει απλά «τυφλή» λεξικογραφική αντικατάσταση.
  - Θέλει μεγάλη **προσοχή** καθώς αυτό μπορεί να προκαλέσει **λάθη**.
    - **`int i = 2; max(++i, 2)`** → **`++i > 2 ? ++i : 2`** → **`4`**
    - **`max(3, 2) * 2`** → **`3 > 2 ? 3 : 2 * 2`** → **`3`**

Preprocessor directives (3/6)

- Στα macros μπορούμε να κάνουμε χρήση και δύο ειδικών τελεστών, των # (*stringify*) και ## (*paste*).
  - **`#define str(x) #x`**
    - **`std::cout << str(lala)`** → **`std::cout << "lala"`**
  - **`#define concat(x, y) x##y`**
    - **`concat(c, out) << "test"`** → **`cout << "test"`**
- Ό,τι κάνουμε **#define** ισχύει από τη γραμμή της δήλωσης μέχρι το τέλος του αρχείου στο οποίο δηλώνεται ή γίνεται **#include**, ανεξάρτητα από τη δομή των blocks του κώδικα μας.
- Μπορούμε να αναιρέσουμε μία δήλωση με το **#undef**:
  - **`#define SIZE 100`**
  - **`#undef SIZE`**

Preprocessor directives (4/6)

- Μπορούμε να έχουμε και *conditional logic* στα directives του preprocessor:
  - **#if - #elif - #else**
  - Αντί για **#if** condition μπορούμε να έχουμε επίσης και:
    - **#ifdef id == #if defined id == #if defined(id)**
    - **#ifndef id == #if !defined id == #if !defined(id)**

```
#ifndef SIZE          // if SIZE is not defined
#   define SIZE 100  // define it with the given value
#elif SIZE > 100      // if it is defined with a value > 100
#   undef SIZE       // first undefine it
#   define SIZE 100  // then redefine it with the given value
#elif !defined SZ     // if SIZE is defined but SZ is not
#   define SZ SIZE   // define SZ as SIZE (SZ will get the same value)
#endif                // any if directive ends with an endif directive
```

Preprocessor directives (5/6)

- Μπορούμε να ενσωματώσουμε ένα ολόκληρο αρχείο:
  - **`#include <file>, #include "file"`**, π.χ.:
    - **`#include <iostream>, #include "SymbolTable.h"`**
- Μπορούμε να αλλάξουμε το όνομα του αρχείου, καθώς και τη γραμμή στην οποία βρισκόμαστε:
  - **`#line number "file"`**, π.χ.: **`#line 30 "test.cpp"`**
- Μπορούμε να προκαλέσουμε ένα compilation error:
  - **`#error message`**
    - **`#ifndef __cplusplus`**
    - **`#  error A C++ compiler is required!`**
    - **`#endif`**
- Τέλος, υπάρχει η εντολή **#pragma**, που παρέχει κάποιες επιλογές του εκάστοτε compiler (compiler-specific), π.χ. **#pragma once**, η οποία λειτουργεί ως *#include guard*, δηλαδή φροντίζει ένα header file να μην ενσωματωθεί περισσότερες από μία φορές, αποτρέποντας τη δημιουργία duplicate definitions.

Preprocessor directives (6/6)

macro	value
__LINE__	Integer value representing the current line in the source code file being compiled.
__FILE__	A string literal containing the presumed name of the source file being compiled.
__DATE__	A string literal in the form "Mmm dd yyyy" containing the date in which the compilation process began.
__TIME__	A string literal in the form "hh:mm:ss" containing the time at which the compilation process began.
__cplusplus	An integer value. All C++ compilers have this constant defined to some value. If the compiler is fully compliant with the C++ standard its value is equal or greater than 199711L depending on the version of the standard they comply.

Μερικά macros που ορίζονται σχεδόν από όλους τους compilers.