"Ss. Cyril and Methodius" University in Skopje
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Lesson 2

Introduction in C++

Object oriented programming

Contents

1 Introduction

2 Functions as struct members

3 References

4 Namespaces

Introduction

- C++ includes many characteristics of the C programming language enriched with object-oriented flavour (OOP)
- C++ is developed at Bell laboratories and at the begining was called “C with classes”
- The name C++ is increment of C (++) and this means that C++ is extended C
- C++ compiler can be used to compile C programs

Input/Output streams in C++ <iostream>

In C++ to work with input/output streams instead of functions `printf` and `scanf` we use the expressions `cout «` and `cin »`.

Example

```
printf("Enter new value: ");
scanf("%d", &value);
printf("The new value is: %d\n", value);
```

in C++ is:

```
cout << "Enter new value: ";
cin >> value;
cout << "The new value is: " << vred << '\n';
```

Therefore it's recommended to replace '$\backslash n$' with '`endl`':

```
cout << "Novata vrednost e: " << vred << endl;
```

Input/Output streams in C++ <iostream>

To use input/output streams in C++ you should include the header file <iostream>

Example

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

int main() {
    cout << "Enter your age: ";
    int myAge;
    cin >> myAge;
    cout << "Enter your frined's age: ";
    int friendsAge;
    cin >> friendsAge;
    if (myAge > friendsAge)
        cout << "You are older.\n";
    else if (myAge < friendsAge)
        cout << "Your friend is older.\n";
    else
        cout << "You both are the same age.\n";
    return 0;
}
```

Declaration of variables in C++

The variables in C++ can be declared anywhere in the programm until their declaration is before the usage.

Example

```
for(int i = 0;  i < 5; i++)
    cout << i << endl;
```

The scope of the local variables begins with the declaration and ends at the end of the block `}`. The declration can be done in the conditions or in the loop expressions `while`, `do/while`, `for` or `if`.

inline functions

Each standard call of function consumes additional time in the process of calling the function. In C++ while defining small and simple function can be used the keyword `inline`, so each call of this function can be replaced with the body of the function, so the additional overhead of calling the function will be lost.

Example function for computing volume of cube

```
#include <iostream>
using namespace std;
inline float cube(const float s) {
    return s * s * s;
}
int main() {
    float a;
    cin >> a;
    cout << "Volume of the cube with side " << a << " is: " << cube(a)
         << endl;
    return 0;
}
```

Additional keywords in C++

`asm`	`explicit`	`operator`
`catch`	`friend`	`private`
`class`	`inline`	`protected`
`const_cast`	`mutable`	`public`
`delete`	`new`	`reinterpret_cast`
`dynamic_cast`	`namespace`	`static_cast`
`template`	`throw`	`using`
`this`	`try`	`virtual`

typedef

Using typedef is premitted, but not nessesery in defining structs, unions or enums (enum) in C++.

Example

```
#include <iostream>
using namespace std;
struct Person {
    char name[80], address[90];
    double sallary;
};
int main() {
    Person employee[50]; // array of Persons
    Person manager;
    manager.name = "Alexander";
    cout << "Name of the manager is " << manager.name << endl;
    return 0;
}
```

Contents

1 Introduction

2 Functions as struct members

3 References

4 Namespaces

Functions as struct members
1/3

C++ allows defining functions as part of some struct.

Example

```
struct Person {
    char name[80], address[80];
    // declaration of function for printing the name and the address
    void print();
};
```

Functions as struct members
2/3

1. The function `print()` is only delcared; the body is in another part of the programm.
2. The size of the struct (sizeof) is defined **only** from the size of its member size. The declared function does not affect its size.
3. The access of the function declared as part of some struct is identical as the access of data member, i.e. by using the operator `(.)`. When we have pointer to struct than we use `->`. This syntax allows usage of same function name in many structs.

Functions as struct members
3/3

Usage of function as struct member

Example

```
#include <iostream>
using namespace std;
struct Person {
    char name[80], address[90];
    double sallary;
    void print();
};
void Person::print() {
    cout << "Name of the employee is:" << name << " and his address is:"
            << address << endl;
}
int main() {
    Person manager;
    manager.name = "Michael";
    manager.address = "St. 5th avenue";
    manager.print();
    return 0;
}
```

Contents

1 Introduction

2 Functions as struct members

3 References

4 Namespaces

References

Definition and declaration

Reference is new data type in C++ similar to the pointer type in C but more safe for usage.

Declaration

```
<Type> & <Name>
```

Example

```
int A = 5;
int& rA = &A;
```

References

Differences and similarities with pointers

References in C++ have these differences with pointers:

1. Direct access to the reference after its declaration is not possible, each access is actualy access to the variable/object it is referencing.
2. Once initialized it can not be dereferenced or refere to other variable/object.
3. It can not be NULL (referencing nothing).
4. Once declared they must be initialized.

References

Usage

One of the most important usage is passing function arguments.

Example

```
int swap(int &a, int &b) {
    a += b;
    b = a - b;
    a -= b;
}
int main() {
    int x = 10;
    int y = 15;
    swap(x, y);
    // x = 15, y = 10
    return 0;
}
```

References

Example arguments struct reference

```
// declaration of struct
struct Person {
    char name[80], address[90];
    double sallary;
};
Person employee[50]; // array of Person
// print accepts reference to Person struct
void print(Person const &p) {
    cout << "Name of the employee is: " << p.name << " and his address is: "
            << p.address << endl;
}
// return data of the employee by his index
Person const &getEmployee(int index) {
... return (person[index]); // returns reference
}
int main() {
    Person employee;
    print(employee); // passing is same as is there was no reference
    return (0);
}
```

Usage of const instead of #define

By using const we specify that the value of the variable can not be changed. For instance in the following code segment, the expression of changing the value of the variable ival = 4 will result in error.

```
int main() {
    int const ival = 3;
    // int const initialized with value 3
    ival = 4; // reports error message
    return 0;
}
```

In contrary to C, in C++ the variables that are defined as const can be used in array declarations.

```
int const size = 20;
char buffer[size]; // array of 20 chars
```

Usage of const with pointers

```
char const *buf;
```

buf is pointer to chars that are defined as const, i.e. can not be changed, while the pointer buf can be changed. Therefore, buf++ is allowed, but *buf = 'a' is not allowed.
Then the expression

```
char *const buf;
```

defines constant pointer that can not be changed, but the data of type char that the pointers buf points can be changed.
And finally,

```
char const *const buf;
```

means that neither the pointer nor the data can not be changed.

Usage of const with pointers

Definiions and declarations where the keywork const is used are read beginning with the name of the variable (the function) to the type of the variable (function). Therefore, the last expression is read as "buf is const pointer to const char".

```
#include <iostream>
using namespace std;
int main() {
    char const *buf = "hello";
    buf++; // accepted by compiler
    *buf = 'u'; // not accepted, error
    return 0;
}
```

Contents

1 Introduction

2 Functions as struct members

3 References

4 Namespaces

Namespaces - namespace
1/3

namespace are parts in the code where you can define and use (without conflicts or name collisions) entities (variables, functions, structs, etc.) with same names as other entities defined in other namespaces.

Defining namespace

```
namespace name_of_namespace {
    // (region for declaration of entities)
    // variables, functions, structs, classes
    // other nested namespaces
}
```

Namespaces - namespace
2/3

Definition of namespace can not be in block (ex. function), but same namespaces can be declared on many places (`multiple namespace`). Then we say, the namespace is open.
For example, if we have declaration of `namespace My_space` во `prog1.cpp` and `prog2.cpp`, then they will be united in one namespace which contains all the entities from the two "physical" spaces.

```
// vo prog1.cpp
namespace My_space {
    double cos(double argInDegrees) {
    ... function body
    }
}
// vo prog2.cpp
namespace My_space {
    double sin(double argInDegrees) {
    ...function body
    }
}
```

Namespaces - namespace
3/3

The namespace `My_space` contains the definitions of both functions, `sin()` and `cos()`. Instead of defining, entities can only be declared as in the following example:

```
namespace My_space {
    double cos(double degrees);
    double sin(double degrees);
}
```

Closed namespaces

Namespace can be defined without name; then it is anonimous (closed) i.e. the entities of this namespace can only be used in the program where it is defined.
Accessing entities from namespace

```
// namespace My_space is defined in the following header file:
#include <My_space>
int main() {
    cout << "Cosinus od 60 stepeni e: " <<
    My_space::cos(60) << endl;
    return 0;
}
```

Closed namespaces

Definition of entity (ex. function) with same name as a standard function implicates that the standard function can not be used, but should be used with the scope operator as in the following example: Accessing functions with same name as functions from the standard name space in `#include <iostream>`

```
#include <cmath>
namespace My_space {
double cos(double argInDegrees) {
    ...body function
}
using namespace std;
int main() {
    using Moj_prostor::cos;
    ...
    cout << cos(60) // using function from My_space::cos()
    << ::cos(1.5) // using standard cos() function
    << endl;
    return (0);
}
```

New casting syntax

In C we used the following casting syntax

```
(typename)expression
(float) nominator;
```

In C++ it's available new notation:

```
typename(expression)
float(nominator);
```

Also there are 4 new ways of casting in C++:

1. `static_cast<type>(expression)` - standard static casting
2. `const_cast<type>(expression)` - used for modifying the type of constants
3. `reintrepret_cast<type>(expression)` - used to reinterpret the information
4. `dynamic_cast<type>(expression)` - polymorphism casting

Materials and Questions

Lectures, exsercises and announcements
courses.finki.ukim.mk

Source code of all examples and problems
https://github.com/tdelev/SP/tree/master/latex/src

Questions and discussion
forum.finki.ukim.mk