OBJECT PROGRAMMING

- LECTURE 6 -

(1ST YEAR OF STUDY)

Skopje, 2023/24

Contents

6.1. Interface and Implementation

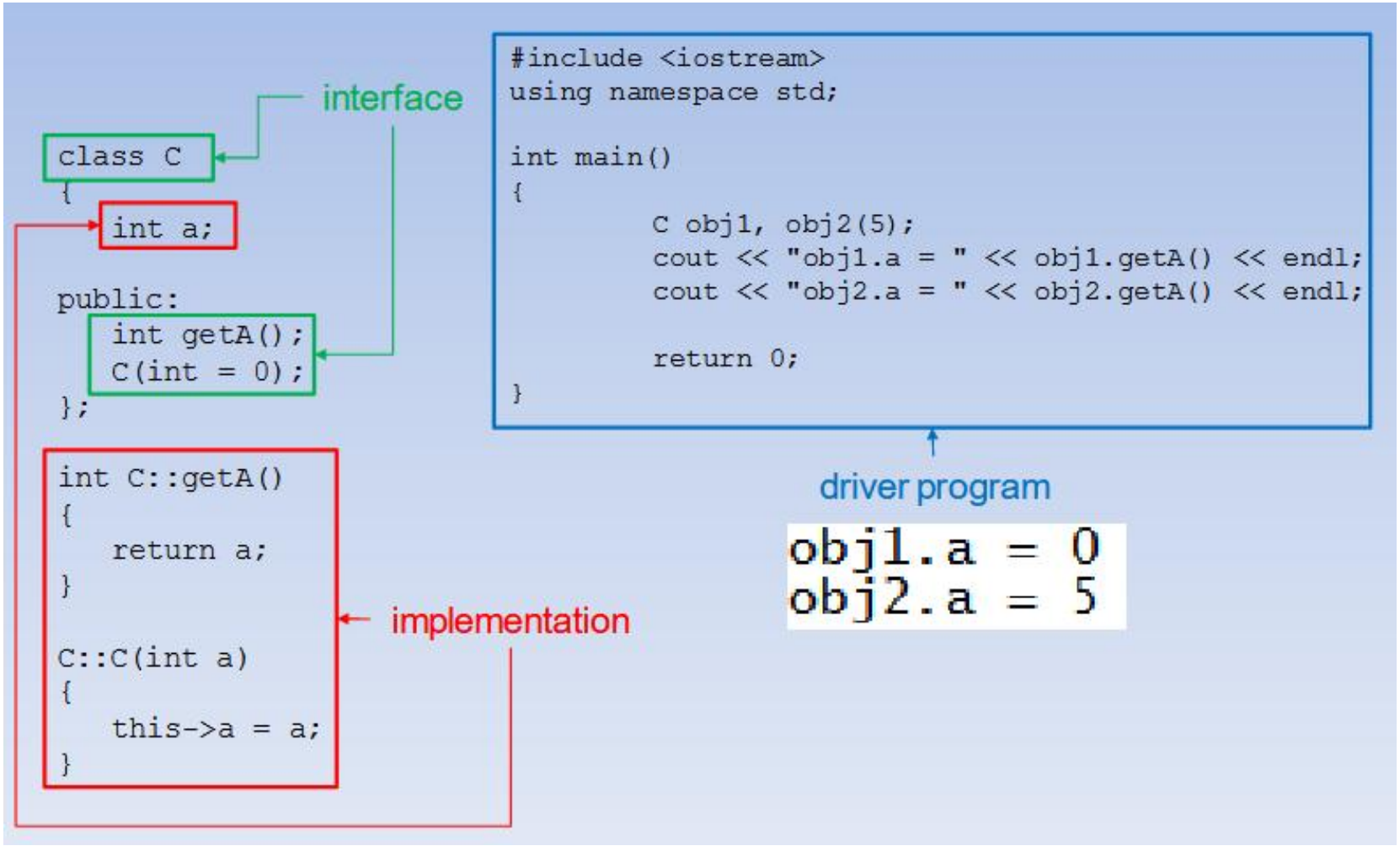
- In general Object Programming (OP) terms:
  - An **interface** is a specification of the members that a user created data type has; and
  - An **implementation** is the code which enables the functionality of that interface.
- Thus, the *interface* lets the compiler know that a user created data type will contain certain features, whereas the *implementation* defines how those features will function.

6.1. Interface and Implementation

- The interface and implementation concern user created data types, i.e., classes.
- The *main* function is usually kept separate, i.e., in a separate file.
- A file containing the *main* function is called a driver file, or a **driver program**.
- The interface and implementation of every user created data type are usually stored in separate files themselves, called ***header files*** (or simply ***headers***).
  - E.g., the *string* data type is kept separately from the rest of the C++ language, and needs to be *#included* in the program if it needs to be used.

6.1. Interface and Implementation

- An Example:



6.2. Function Overloading in C++

- **Function Overloading** is a feature of object-oriented programming where two or more functions can have the same name, but different parameters.
- When a function name is overloaded with different jobs, it is called Function Overloading.
- In Function Overloading "Function" name should be the same and the arguments should be different. Function overloading can be considered as an example of a *polymorphism* feature in C++.

6.2. Function Overloading in C++

- If we have to perform only one operation, and having same name of the functions increases the readability of the program.
- Suppose you have to perform addition of the given numbers, but there can be any number of arguments, if you write the function such as **a(int,int)** for two parameters, and **b(int,int,int)** for three parameters, then it may be difficult for you to understand the behavior of the function because its name differs.
- The *parameters* should follow any one, or more than one, of the following conditions for Function Overloading:

6.2. Function Overloading in C++

1) Parameters should have a different type

- add(int a, int b)
- add(double a, double b)

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


void add(int a, int b)
{
  cout << "sum = " << (a + b);
}

void add(double a, double b)
{
    cout << endl << "sum = " << (a + b);
}

// Driver code
int main()
{
    add(10, 2);
    add(5.3, 6.2);

    return 0;
}
```

Output

```
sum = 12
sum = 11.5
```

6.2. Function Overloading in C++

2) Parameters should have a different number

- add(int a, int b)
- add(int a, int b, int c)

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

void add(int a, int b)
{
  cout << "sum = " << (a + b);
}

void add(int a, int b, int c)
{
    cout << endl << "sum = " << (a + b + c);
}

// Driver code
int main()
{
    add(10, 2);
    add(5, 6, 4);

    return 0;
}
```

Output

```
sum = 12
sum = 15
```

6.2. Function Overloading in C++

3) Parameters should have a different sequence of parameters

- add(int a, double b)
- add(double a, int b)

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

void add(int a, double b)
{
    cout<<"sum = "<<(a+b);
}

void  add(double a, int b)
{
    cout<<endl<<"sum = "<<(a+b);
}

// Driver code
int main()
{
    add(10,2.5);
    add(5.5,6);

      return 0;
}
```

Output

```
sum = 12.5
sum = 11.5
```

6.2. Function Overloading in C++

- Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char const *c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output

```
Here is int 10
Here is float 10.1
Here is char* ten
```

6.2. Function Overloading in C++

- In C++, we can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as ***Constructor Overloading*** and is quite similar to *Function Overloading*.
  - Overloaded constructors essentially have the same name (exact name of the class), and differ by number and type of arguments.
  - A constructor is called depending upon the number and type of arguments passed.
  - While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

6.2. Function Overloading in C++


```
// C++ program to illustrate
// Constructor overloading
#include <iostream>
using namespace std;

class construct
{

public:
    float area;

    // Constructor with no parameters
    construct()
    {
        area = 0;
    }

    // Constructor with two parameters
    construct(int a, int b)
    {
        area = a * b;
    }

    void disp()
    {
        cout<< area<< endl;
    }
};
```

```
int main()
{
    // Constructor Overloading
    // with two different constructors
    // of class name
    construct o;
    construct o2( 10, 20);

    o.disp();
    o2.disp();
    return 1;
}
```

Output:

```
0

200
```

6.3. Functions that cannot be overloaded

- In C++, following function declarations **cannot** be overloaded:
- 1) Function declarations that differ only in the return type. For example, the following program fails in compilation.

```
#include<iostream>
int foo() {
  return 10;
}

char foo() {
  return 'a';
}
```

```
int main()
{
    char x = foo();
    getchar();
    return 0;
}
```

6.3. Functions that cannot be overloaded

- 2) Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a ***static*** member function declaration. For example, the following program fails in compilation.

```
#include<iostream>
class Test {
   static void fun(int i) {}
   void fun(int i) {}
};
```

```
int main()
{
   Test t;
   getchar();
   return 0;
}
```

6.3. Functions that cannot be overloaded

- 3) Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is, the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types. For example, following two function declarations are equivalent.

```
int fun(int *ptr);
int fun(int ptr[]); // redeclaration of fun(int *ptr)
```

6.3. Functions that cannot be overloaded

- 4) Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)()); // redeclaration of h(int())
```

6.3. Functions that cannot be overloaded

- 5) Parameter declarations that differ only in the presence or absence of ***const*** are equivalent. That is, the const specifier for each parameter type is ignored when determining which function is being declared, defined, or called. For example, following program fails in compilation with error *"redefinition of `int f(int)' "*

6.3. Functions that cannot be overloaded


```
#include<iostream>
#include<stdio.h>

using namespace std;

int f ( int x) {
    return x+10;
}

int f ( const int x) {
    return x+10;
}

int main() {
  getchar();
  return 0;
}
```

- Only the *const* specifier at the outermost level of the parameter type specification is ignored in this fashion; *const* specifier buried within a parameter type specification is significant and can be used to distinguish overloaded function declarations. In particular, for any type T, “pointer to T,” and “pointer to const T,” are considered distinct parameter types, as are “reference to T,” and “reference to const T.”

6.3. Functions that cannot be overloaded

- 6) Two parameter declarations that differ only in their default arguments are equivalent. For example, following program fails in compilation with error *"redefinition of `int f(int, int)' "*

```
#include<iostream>
#include<stdio.h>

using namespace std;

int f ( int x, int y) {
    return x+10;
}

int f ( int x, int y = 10) {
    return x+y;
}
```

```
int main() {
  getchar();
  return 0;
}
```

6.4. Function Overloading and const keyword

- C++ allows member methods to be overloaded on the basis of ***const*** type.
- Overloading on the basis of const type can be useful when a function returns a reference or pointer. We can make one function *const*, that returns a const reference or const pointer, and another *non-const* function, that returns a non-const reference or pointer.
- **What about parameters?** Rules related to ***const*** parameters are interesting. Let us first take a look at the following two examples. Program 1 fails in compilation, but program 2 compiles and runs fine.

6.4. Function Overloading and const keyword


```
// PROGRAM 1 (Fails in compilation)
#include<iostream>
using namespace std;

void fun(const int i)
{
    cout << "fun(const int) called ";
}
void fun(int i)
{
    cout << "fun(int ) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

Output:

```
Compiler Error: redefinition of 'void fun(int)'
```

6.4. Function Overloading and const keyword


```
// PROGRAM 2 (Compiles and runs fine)
#include<iostream>
using namespace std;

void fun(char *a)
{
cout << "non-const fun() " << a;
}

void fun(const char *a)
{
cout << "const fun() " << a;
}

int main()
{
const char *ptr = "GeeksforGeeks";
fun(ptr);
return 0;
}
```

Output

```
const fun() GeeksforGeeks
```

6.4. Function Overloading and const keyword

- C++ allows functions to be overloaded on the basis of the *const-ness* of parameters, only if the const parameter is a reference or a pointer.
- That is why program 1 failed in compilation, but program 2 worked fine. This rule actually makes sense. In program 1, the parameter 'i' *is passed by value*, so 'i' in fun() is a copy of 'i' in main(). Hence fun() cannot modify 'i' of main().
- Therefore, it doesn't matter whether 'i' is received as a *const* parameter, or a *normal* parameter.

6.4. Function Overloading and const keyword

- When we *pass by reference or pointer*, we can modify the value referred or pointed, so we can have two versions of a function, one which can modify the referred or pointed value, other which can not.
- As an exercise, predict the output of the following program.

6.4. Function Overloading and const keyword


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

void fun(const int &i)
{
    cout << "fun(const int &) called ";
}
void fun(int &i)
{
    cout << "fun(int &) called " ;
}
int main()
{
    const int i = 10;
    fun(i);
    return 0;
}
```

Output

```
fun(const int &) called
```

6.5. Function Overloading and Return Type

- *Function overloading* is possible in C++ and Java but only if the functions must differ from each other by the types and the number of arguments in the argument list. However, functions can not be overloaded if they differ only in the return type. Why is ***Function Overloading*** not possible with different ***return types***?
  - *Function overloading* comes under the compile-time polymorphism. During compilation, the function signature is checked. So, functions can be overloaded, if the signatures are not the same. The return type of a function has no effect on function overloading, therefore the same function signature with different return type will not be overloaded.

6.5. Function Overloading and Return Type

- Example: if there are two functions: **int sum()** and **float sum()**, these two will generate a ***compile-time error***, as function overloading is not possible here.
- Let's understand this further through the following program:

6.5. Function Overloading and Return Type


```cpp
// CPP Program to demonstrate that function overloading
// fails if only return types are different
#include <iostream>
int fun() { return 10; }

char fun() { return 'a'; }
// compiler error as it is a new declaration of fun()

// Driver Code
int main()
{
    char x = fun();
    getchar();
    return 0;
}
```

Output

```
prog.cpp: In function ‘char fun()’:
prog.cpp:6:10: error: ambiguating new declaration of ‘char fun()’
char fun() { return 'a'; }
          ^
prog.cpp:4:5: note: old declaration ‘int fun()’
int fun() { return 10; }
    ^
```

6.6. Function Overloading and float

- Although polymorphism is a widely useful phenomena in C++ yet it can be quite complicated at times. For instance consider the following code snippet:

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s, int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5, 5.6);
    return 0;
}
```

6.6. Function Overloading and float

- It may appear that the call to the function *test* in *main()* will result in output "Function with float called", but the code gives following error:

```
In function 'int main()':
13:13: error: call of overloaded 'test(double, double)' is ambiguous
 test(3.5,5.6);
```

- It's a well known fact in *Function Overloading*, that the compiler decides which function needs to be invoked among the overloaded functions. If the compiler can not choose a function amongst two or more overloaded functions, the situation is - "**Ambiguity** in Function Overloading".

6.6. Function Overloading and float

- The reason behind the ambiguity in above code is that the floating literals **3.5** and **5.6** are actually treated as ***double*** by the compiler.
- As per C++ standard, *floating point literals* (compile-time constants) are treated as *double* unless explicitly specified by a *suffix*.
- Since compiler could not find a function with double argument and got confused if the value should be converted from *double* to *int* or *float*.

6.6. Function Overloading and float

- **Rectifying the error:** We can simply tell the compiler that the literal is a float and NOT double by providing **suffix f.**

  Look at the following code :

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
    cout << "Function with float called ";
}
void test(int s,int t)
{
    cout << "Function with int called ";
}
int main()
{
    test(3.5f, 5.6f); // Added suffix "f" to both values to
                      // tell compiler, it's a float value
    return 0;
}
```

Output:

```
Function with float called
```

6.7. Advantages and disadvantages

- **Function Overloading and Default Arguments**
- All default arguments must be the ***rightmost*** arguments. The following program works fine and produces ***10 as output***.

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

int fun(int x = 0, int y = 0, int z = 0)
{ return (x + y + z); }

int main()
{
cout << fun(10);
return 0;
}
```

6.7. Advantages and disadvantages

- **Can main() be overloaded in C++?**
- Predict the output of following C++ program.

```
#include <iostream>
using namespace std;
int main(int a)
{
    cout << a << "\n";
    return 0;
}
int main(char *a)
{
    cout << a << endl;
    return 0;
}
```

```
int main(int a, int b)
{
    cout << a << " " << b;
    return 0;
}
int main()
{
    main(3);
    main("C++");
    main(9, 6);
    return 0;
}
```

6.7. Advantages and disadvantages

- The above program fails in compilation and produces warnings and errors. You may get different errors on different compilers.
- To overload *main() function* in C++, it is necessary to use ***class*** and declare the *main* as ***member function***.
- Note that *main* is not reserved word in programming languages like C, C++, Java and C#.

6.7. Advantages and disadvantages

- For example, we can declare a variable whose name is main, try below example:

```
#include <iostream>
int main()
{
    int main = 10;
    std::cout << main;
    return 0;
}
```

Output:

```
10
```

6.7. Advantages and disadvantages

- The following program shows overloading of main() function in a class.

```
#include <iostream>
using namespace std;
class Test
{
public:
    int main(int s)
    {
        cout << s << "\n";
        return 0;
    }
    int main(char *s)
    {
        cout << s << endl;
        return 0;
    }
    int main(int s ,int m)
    {
        cout << s << " " << m;
        return 0;
    }
};
int main()
{
    Test obj;
    obj.main(3);
    obj.main("I love C++");
    obj.main(9, 6);
    return 0;
}
```

The outcome of program is:

```
3
I love C++
9 6
```

6.7. Advantages and disadvantages

- ***Function overloading*** is one of the important features of object-oriented programming.
- It allows users to have more than one function having the same name, but different properties.
- Overloaded functions enable users to supply different semantics for a function, depending on the signature of functions.

6.7. Advantages and disadvantages

- **Advantages of function overloading are:**
  - The main advantage of function overloading is that it improves code readability and allows code reusability.
  - The use of function overloading is to save memory space, consistency, and readability.
  - It speeds up the execution of the program.
  - Code maintenance also becomes easy.
  - Function overloading brings flexibility to code.
  - The function can perform different operations and hence it eliminates the use of different function names for the same kind of operations.

6.7. Advantages and disadvantages

- **Disadvantages of function overloading are:**
  - Function declarations that differ only in the return type cannot be overloaded; illustration:

```
int fun();
float fun();
```

    - It gives an error because the function cannot be overloaded by return type only.
  - Member function declarations with the same name and the same parameter types cannot be overloaded if any of them is a *static* member function declaration.
  - The main disadvantage is that it requires the compiler to perform name mangling on the function name to include information about the argument types.

6.7. Advantages and disadvantages


```
// Importing input output stream files
#include <iostream>

using namespace std;

// Methods to print

// Method 1
void print(int i)
{

    // Print and display statement whenever
    // method 1 is called
    cout << " Here is int " << i << endl;
}

// Method 2
void print(double f)
{

    // Print and display statement whenever
    // method 2 is called
    cout << " Here is float " << f << endl;
}
```

```
// Method 3
void print(char const* c)
{

    // Print and display statement whenever
    // method 3 is called
    cout << " Here is char* " << c << endl;
}

// Method 4
// Main driver method
int main()
{

    // Calling method 1
    print(10);
    // Calling method 2
    print(10.10);
    // Calling method 3
    print("ten");

    return 0;
}
```

Output

```
Here is int 10
Here is float 10.1
Here is char* ten
```

6.7. Advantages and disadvantages

- **Output Explanation:**
- In the above example all functions that were named the 3 same but printing different to which we can perceive there were different calls been made.
- This can be possible because of the arguments been passed according to which the function call is executed no matter be the functions are sharing a common name.
- Also, remember we can overload the function by changing their signature in return type. For example, here *bool print()* function can be called for all three of them.