Maciej Sobieraj

Lecture 6

Outline

1. **Inheritance**
   1. Defining class hierarchy
   2. Classes, inheritance and type compatibility
   3. Polymorphism and virtual methods
   4. Objects as parameters and dynamic casting
   5. **Various supplements**
   6. The const keyword

More about copying constructors

- The **copying constructor** is a specific form of constructor designed to make a more or less literal copy of an object. You can recognize this constructor by its **distinguishable header**.
- Assuming that a class is called A, its copying constructor will be declared as:
  - A(A &)
- **The implicit constructor simply clones** (bit by bit) **the source object**

More about copying constructors

```
#include <iostream>
using namespace std;
class Class {
	int data;
public:
	Class(int value) : data(value) {}
	void increment(void) {	data++; }
	int value(void) { return data; }
};
int main(void) {
	Class o1(123);
	Class o2 = o1;
	Class o3(o2);

	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;

	return 0;
}
```

More about copying constructors

- The code will produce the following output:
  - 124
  - 123
  - 123

More about copying constructors

```
#include <iostream>
using namespace std;
class Class {
	int *data;
public:
	Class(int value) {
		data = new int;
		*data = value;
	}
	void increment(void) { (*data)++; }
	int value(void) { return *data; }
};
int main(void) {
	Class o1(123);
	Class o2 = o1;
	Class o3(o2);
	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;
	return 0;
}
```

More about copying constructors

- The code we’ve created produces the following output:
  - 124
  - 124
  - 124

More about copying constructors

```
include <iostream>
using namespace std;
class Class {
	int *data;
public:
	Class(int value) {
		 data = new int;
		*data = value;
	}
	void increment(void) {	(*data)++; }
	int value(void) { return *data; }
};

int main(void) {
	Class o1(123);
	Class o2(o1.value());
	Class o3(o2.value());
	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;
	return 0;
}
```

More about copying constructors

- the code produces the following output:
  - 124
  - 123
  - 123

More about copying constructors

```
#include <iostream>
using namespace std;
class Class {
	int *data;
public:
	Class(int value) {
		data = new int;
		*data = value;
	}
	Class(Class &source) {
		data = new int;
		*data = source.value();
	}
	void increment(void) {	(*data)++; }
	int value(void) { return *data; }
};
int main(void) {
	Class o1(123);
	Class o2 = o1;
	Class o3(o2);
	o1.increment();
	cout << o1.value() << endl;
	cout << o2.value() << endl;
	cout << o3.value() << endl;
	return 0;
}
```

More about copying constructors

- The code will output the following lines to the screen:
  - 124
  - 123
  - 123

More about copying constructors

- The mechanism of passing parameters by value assumes that a function operates on the copy of an actual parameter.
- This is clear when we consider parameters of simple types (like *int* or *float*), but it becomes more complex when the parameter is an object.

More about copying constructors

```
#include <iostream>
using namespace std;
class Dummy {
public:
	Dummy(int value) {}
	Dummy(Dummy &source) {
		cout << "Hi from the copy constructor!" << endl;
	}
};
void DoSomething(Dummy ob) {
	cout << "I'm here!" << endl;
}
int main(void) {
	Dummy o1(123);
	DoSomething(o1);
	return 0;
}
```

More about copying constructors

- The output of the program isn't really complex – it says:
  - Hi from the copy constructor!
  - I'm here!

More about copying constructors

- The program will cause at least two compilation errors

```
#include <iostream>
using namespace std;
class Dummy {
private:
	Dummy(Dummy &source) {}
public:
	Dummy(int value) {}
};
void DoSomething(Dummy ob) {
	cout << "I'm here!" << endl;
}
int main(void) {
	Dummy o1(123);
	Dummy o2 = o1;
	DoSomething(o1);
	return 0;
}
```

More about default constructors

- the class will be implicitly equipped with the so-called **implicit default (parameter-less) constructor** but the constructor will do nothing at all.

More about default constructors

```
#include <iostream>
using namespace std;
class NoConstructorsAtAll {
public:
	int i;
	float f;
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	NoConstructorsAtAll  o1;
	NoConstructorsAtAll *o2;
	o2 = new NoConstructorsAtAll;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

More about default constructors

- The class has no constructor. In effect their fields will not be initialized in any way. The values outputted by the *display* method are completely random. The number we've seen won't be repeated when you run the program on your computer.
- One of our outputs is as follows:
  - i=2147344384,f=1.54143e-044
  - i=5641768,f=7.89812e-039

More about default constructors

- The default constructor has to be implicitly invoked when a new object is created (twice in our example). We get error (twice)

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
	int i;
	float f;
	WithConstructor(int a, float b) : i(a), f(b) { }
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	WithConstructor  o1;
	WithConstructor *o2;
	o2 = new WithConstructor;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

More about default constructors

- We’ve changed the header of the existing constructor by adding default values to both parameters.

```
#include <iostream>
using namespace std;
class WithConstructor {
public:
	int i;
	float f;
	WithConstructor(int a = 0, float b = 0) : i(a), f(b) { }
	void Display(void) { cout << "i=" << i << ",f=" << f << endl; }
};
int main(void) {
	WithConstructor  o1;
	WithConstructor *o2;
	o2 = new WithConstructor;
	o1.Display();
	o2 -> Display();
	return 0;
}
```

More about default constructors

- The program produces the following output:
  - i=0,f=0
  - i=0,f=0

Compositions vs. constructors

```
#include <iostream>
using namespace std;
class A {
public:
	void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
	void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
	A f1;
	B f2;
};
int main(void) {
	Compo  co;
	co.f1.Do();
	co.f2.Do();
	return 0;
}
```

Compositions vs. constructors

- The program produces the following output:
  - A is doing something
  - B is doing something

Compositions vs. constructors

```
#include <iostream>
using namespace std;
class A {
public:
	A(A &src) { cout << "copying A..." << endl; }
	A(void) { }
	void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
	B(B &src) { cout << "copying B..." << endl; }
	B(void){ }
	void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
	Compo(void) { } ;
	A f1;
	B f2;
};
int main(void) {
	Compo  co1;
	Compo  co2 = co1;
	co2.f1.Do();
	co2.f2.Do();
	return 0;
}
```

Compositions vs. constructors

- We’ve compiled the code and run it. It’s produced the following output:
  - copying A...
  - copying B...
  - A is doing something
  - B is doing something
-

Compositions vs. constructors

```
#include <iostream>
using namespace std;
class A {
public:
    A(A &src) { cout << "copying A..." << endl; }
    A(void) { }
    void Do(void) { cout << "A is doing something" << endl; }
};
class B {
public:
    B(B &src) { cout << "copying B..." << endl; }
    B(void){ }
    void Do(void) { cout << "B is doing something" << endl; }
};
class Compo {
public:
    Compo(Compo &src) { cout << "Copying Compo..." << endl; }
    Compo(void) { } ;
    A f1;
    B f2;
};
int main(void) {
    Compo  co1;
    Compo  co2 = co1;
    co2.f1.Do();
    co2.f2.Do();
    return 0;
}
```

Compositions vs. constructors

- The program produces the following output:
  - Copying Compo...
  - A is doing something
  - B is doing something
- The explicit copying constructor (written by us) has invoked none of the component's copying constructors.

Compositions vs. constructors

- One way to do this is to add a line like this one:
  - Compo(Compo &src) : f1(src.f1), f2(src.f2) { cout << "Copying Compo..." << endl; }
- instead of
  - Compo(Compo &src) { cout << "Copying Compo..." << endl; }
- The solution is correct despite how it looks. The modified program behaves the way we want, producing the following output:
  - copying A...
  - copying B...
  - Copying Compo...
  - A is doing something
  - B is doing something

Outline

1. **Inheritance**
    1. Defining class hierarchy
    2. Classes, inheritance and type compatibility
    3. Polymorphism and virtual methods
    4. Objects as parameters and dynamic casting
    5. Various supplements
    6. **The const keyword**
    7. Friendship in the “C++” world

The const keyword

```
const int size1 = 100;
int const size2 = 100;
```

- size1 and size2 are variables of type int and have a value of 100.
- Both entities behave like constants (more precisely, as read-only variables).
- Note that the const keyword is located in different places in each line. Both forms are acceptable.

The const keyword

- The compiler will **protect both variables from being modified**. The following two lines will cause compilation errors:
  - size1++;
  - size2 = size1;
- You can use both symbols (names of const) anywhere you can use a literal or an expression consisting of literals, as in this example:
  - **const int** size = 100;
  - **int** buffer[**size**];

The const keyword

- Note that **you mustn't declare a *const* without initialization** (think about this for a moment and you'll find it obvious). The following line will cause a compilation error:
  - **const int** size;

Constant aggregates

- Aggregates (structures and arrays as well as arrays of structures and structures of arrays *et cetera*) may be declared as *const* too, although the effects are somewhat different.

```
const int points[5] = {1, 2, 4, 8, 16};
const struct { int key; } data = { 10 };
```

Constant aggregates

- ***points* and *data* are read-only variables and you mustn't modify them**. Both of the following lines are wrong:
  - --points[2];
  - data.key = 0;
- Some of the "C++" compilers may consider the following line as incorrect:
  - **int** array[points[2] + data.key];
- as the compiler may not be able to determine the number of the array's elements during the compile time.

Constant pointers

- Pointers are allowed to be declared as const as well.

```
int arr[5] = {1, 2, 4, 8, 16};
int * const iptr = arr + 2;
char * const cptr = "Why?";
```

- Both iptr and cptr mustn't be modified. This means that the following lines will cause compilation errors:
  - --iptr;
  - ++cptr;

Constant pointers

- The entities pointed to by the *const* pointers may be modified with no restrictions. The following two lines will be accepted and successfully performed:
  - *iptr = 0;
  - *cptr = 'T';

Pointers to constants

- **Constant pointers aren't equivalents for pointers to constants.**

```
int arr[5] = {1, 2, 4, 8, 16};
const int *iptr = arr + 2;
const char *cptr = "Why?";
```

- The const keywords have changed their locations and now they're placed at the beginning of the declarations. Note that the following form is correct too:
  - int const *iptr = arr + 2;
  - char const *cptr = "Why?";

Pointers to constants

- **Both *iptr* and *cptr* may be modified**. The following lines are correct:
  - --iptr;
  - ++cptr;
- In contrast, **the entities pointed to by these pointers cannot be modified any more**. The following two lines will not be accepted:
  - *iptr = 0;
  - *cptr = 'T';

Constant pointers to constants

- **Both of the above variants can be mixed together giving a *const* pointer to a *const* value.**

```
int arr[5] = {1, 2, 4, 8, 16};
const int * const iptr = arr + 2;
const char * const cptr = "Why?";
```

- **None of the following lines are correct in the scope of this declaration:**
  - --iptr;
  - ++cptr;
  - *iptr = 0;
  - *cptr = 'T';

Constant function parameters

- **Any of the function parameters passed by value may be declared as *const*.**

```
int fun(const int n) {
	return n * n;
}
```

- Note that the effects of these declarations are only observable inside the function and have no impact on the outside world.
- Function returns n*n

Constant function parameters

- **Any of the function parameters passed by reference may be declared as *const*.**
- We can say that this is **a stronger form of the previous declaration**. We can understand it as a solemn **promise** made by the function: *I'm not going to modify your actual parameter*.

```
int fun(const int &n) {
    return n++;
}
```

- The snippet is incorrect.

Constant function results

- **Any function may declare its result as *const*.**

```
const char *fun(void) {
    return "Caution!";
}
```

- This line will be rejected by the compiler:
  - **char** *p = fun();
- This one will be accepted:
  - **const char** *str = fun();

Constant class variables

- **Any class may declare its field as *const*.**

```
class Class {
private:
    const int field;
public:
    Class(int n) : field(n) { };
    Class(Class &c) : field(0) { };
    Class(void) : field(1) { };
};
```

- **A *const* class field must be initialized inside an initialization list within any of the class constructors**. Any other assignment will be rejected.

Constant class variables

- All of the constructors initialize the *const field* with a different value. All the initializations are valid.
- The following snippets, inserted inside the public part of the *Class*, will be recognized as invalid:
  - Class(**double** f) { field = f; }
  - **void** fun(**int** n) { field += n; }

Constant objects

- **An object of any class may be declared as *const*.**

```
class Class {
public:
    int field;
    Class(int n) : field(n) { };
    Class(Class &c) : field(0) { };
    Class(void) : field(1) { };
    void set(int n) { field = n; }
    int get(void) { return field; }
};
```

Constant objects

- Let's assume that we have the following declarations (all valid):
  - Class o1(1);
  - **const** Class o2(2);
  - **int** i;
- The following three lines will be rejected:
  - o2.field = 3;
  - o2.set(1);
  - i = o2.get();
- They'll be considered valid if you replace '*o2*' with '*o1*'.

Constant member functions

- **Any of the class's member functions may declare themselves as *const*.**
- The syntax of the declaration may be surprising as the *const* keyword is placed after the parameter list, like this:
  - type name(parameters) const; in declarations
  - type name(parameters) const { … } in definitions

Constant member functions

```
class Class {
public:
	int field;
	Class(int n) : field(n) { };
	Class(Class &c) : field(0) { };
	Class(void) : field(1) { };
	void set(int n) { field = n; }
	int get(void) const { return field; }
};
```

- In effect, the following line will be considered valid:
  - i = o2.get();

Outline

1. **Inheritance**
   1. Defining class hierarchy
   2. Classes, inheritance and type compatibility
   3. Polymorphism and virtual methods
   4. Objects as parameters and dynamic casting
   5. Various supplements
   6. The const keyword
   7. **Friendship in the “C++” world**

Friend or foe?

- A friend of a class may be:
  - a **class** (it's called the **friend class**)
  - a **function** (it's called the **friend function**)
- A friend (class of function) can access those components hidden from others. **Friends are allowed to access or to use private and protected components of the class**.

Friend or foe?

```
#include <iostream>
using namespace std;
class Class {
friend class Friend;
private:
	int field;
	void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
class Friend {
public:
	void DoIt(Class &c) { c.field = 100; c.print(); }
};
int main(void) {
	Class o;
	Friend f;

	f.DoIt(o);
	return 0;
}
```

Friend or foe?

- Note that **it doesn’t matter where you add the friendship declaration**, i.e. the line starting with the phrase:
  - friend class … ;
- may exist inside any of the class parts (public, private or protected), but must be placed outside any function or aggregate.
- This program outputs:
  - It's a secret, that field = 100

The rules

- There are some additional rules that must be taken into account:
  - a class may **be a friend of many classes**
  - a class may **have many friends**
  - a friend’s friend **isn’t my friend**
  - friendship **isn’t inherited** – the subclass has to define its own friendships

The rules

```
#include <iostream>
using namespace std;
class A {
friend class B;
friend class C;
private:
	int field;
protected:
	void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
class C {
public:
	void DoIt(A &a) { a.print(); }
};
class B {
public:
	void DoIt(A &a, C &c) { a.field = 111; c.DoIt(a); }
};
int main(void) {
	A a; B b; C c;

	b.DoIt(a,c);
	return 0;
}
```

The rules

- It's a secret, that field = 111

Friend functions

- **A function may be a class's friend too**.
- The rules are a bit different from before:
  - a friendship declaration must contain **a complete prototype of the friend function** (including return type); a function with the same name, but incompatible in the sense of the parameters' conformance, will not be recognized as a friend
  - a class **may have many friend functions**
  - a function **may be a friend of many classes**
  - a class may recognize as friends **both global and member functions**

Friend functions

```
#include <iostream>
using namespace std;
class A;
class C {
public:
	void dec(A &a);
};
class A {
friend class B;
friend void C::dec(A&);
friend void DoIt(A&);
private:
	int field;
protected:
	void print(void) { cout << "It's a secret, that field = " << field << endl; }
};
void C::dec(A &a) { a.field--; }
class B {
public:
	void DoIt(A &a) { a.print(); }
};
void DoIt(A &a) {
	a.field = 99;
}
int main(void) {
	A a; B b; C c;

	DoIt(a);
	b.DoIt(a);
	return 0;
}
```

Friend functions

- The example program writes:
  - It's a secret, that field = 99