Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger
LS 11 Algorithm Engineering

Lecture 10 • Winter 2011/12 • Dec 20

technische universität dortmund
department of computer science

Default Constructors

- A constructor without any parameters is called a default constructor
- If you do not write any constructor, a default constructor is created automatically:
  - calls the default constructors of all base classes and data members (If any base class or data member of custom type has no default constructor, a default constructor cannot be generated!)
  - leaves data members of built-in types (like `int`) uninitialized!
- Caution: If you implement any (other) constructor, no default constructor will be generated automatically.

```
point::point() : x(0), y(0) { }
```

Copy Constructors

- A constructor taking a const reference of its class as parameter is called a copy constructor.
- Copy constructors are created automatically if you do not provide one:
  - call copy constructors of all data members of custom types
  - copy the values of data members of built-in types
- Copy constructors are called in the following situations:
  - initialize an object with an object
  - pass an object using call-by-value
  - return an object

```
point p(q);
point r = t;
```

```
point f(point p){
    return 2*p;
}
r = f(q);
```

```
point::point(const point &p)
    : x(p.x), y(p.y) { }
```

Destructors

- A destructor is called whenever an object of a class is destroyed (e.g. goes out of scope)
- Destructors are used to do some clean-up work like freeing resources
- Destructors that do nothing are created by default
  $\rightarrow$ Write your own destructor if you need to free resources

```
point::~point() { } // does nothing
```

Assignment Operator

- The assignment operator is called whenever an object is assigned to an object.
- Assignment operators are created automatically if you do not provide one:
  - calls the assignment operator for each data member
- If your class requires to write a copy constructor, it will require to write an assignment operator as well.

```
p = q;
```

```
point &point::operator=(const point &p) {
    x = p.x; y = p.y;
    return *this;
}
```

Memory Addresses

- All the values of variables are stored in the memory of the computer
- Every location has a unique address (an integer value)

```
int    x = 10;
double y = 2.1;
int    z = 15;
```

address	*value*
…	
10240 – 10243	**10**
10244 – 10251	**2.1**
10252 – 10255	**15**
…	

Pointers

- We can manipulate these addresses directly using pointers
- Pointers are frequently used in C, but not so much in C++, since it is better to use references in many cases
- Pointers look similar as iterators, but they are not the same
  - pointers are built-in types of C++
  - iterators are part of the C++ standard library
    (you can also define your own containers and iterators)
  - iterators have been designed such that they look like pointers

Notation for Pointers

- **Address operator:**
  If `var` is a variable, then `&var` denotes its address in memory
- **Dereference operator:**
  If `addr` is an address, then `*addr` denotes its content (the value stored there)
- An address is frequently also called a pointer
- Pointers are typed
  - the type denotes the type stored in memory at the address
  - if *`T`* is the type stored, then *`T`* `*` is the corresponding pointer type
  - e.g.: `int *`, `char *`

Example (1)

```
int a = -1, b = 9;
int *p1, *p2; // undefined values
```

variable	value
p1	?
p2	?

pointers

variable	value
a	-1
b	9

`int`-variables

Example (2)

```
int a = -1, b = 9;
int *p1, *p2; // undefined values
p1  = &a;
```

variable	*value*
p1	
p2	?

pointers

variable	*value*
a	-1
b	9

`int`-variables

Example (3)

```
int a = -1, b = 9;
int *p1, *p2; // undefined values
p1  = &a;
*p1 = b;
```

variable	*value*
p1	
p2	?

pointers

variable	*value*
a	9
b	9

`int`-variables

Example (4)

```
int a = -1, b = 9;
int *p1, *p2; // undefined values
p1  = &a;
*p1 = b;
b   = b * b;
p2  = &b;
```

variable	*value*
p1	
p2	

pointers

variable	*value*
a	9
b	81

`int`-variables

Example (5)

```
int a = -1, b = 9;
int *p1, *p2; // undefined values
p1  = &a;
*p1 = b;
b   = b * b;
p2  = &b;
*p1 = *p2;
```

variable	*value*
p1	
p2	

pointers

variable	*value*
a	81
b	81

`int`-variables

0-Pointers

- You cannot assign arbitrary integer values to pointers
- But you can always assign 0 to any pointer:

```
int *ptr = 0;
```

- 0 is never used as address of an object
  $\rightarrow$ 0 is used as a special value to mark an invalid pointer

Address Operator and References

- Beware of the difference between

```
int a;
int &ref = a;
```

& means:
declare a reference type

and

```
int a;
int *ptr = &a;
```

& means:
take address of variable

The -> Operator

- The **->** operator can be used with pointers to structs/classes:

```
struct S {
    int a;
};

int main() {
    S s;
    S *ptr = &s;
    ptr->a = 10;  // short-hand for: (*ptr).a = 10;
    return 0;
}
```

- The **->** operator is simply a short-hand for dereferencing (*****) and selection (**.**)
  - Compare iterators for maps!

Polymorphism

- Given the following declaration

```
class D : public B { … };
```

we may use an object of class **D** wherever a reference or a pointer to an object of class **B** is expected.
→ Polymorphism

- *A **D** object is a (special) **B** object.*

```
void f(B &b) { … }

int main() {
    D d;
    f(d);
    …
}
```

Polymorphism and Redefining

- Recall that we may redefine member functions of the base class in the derived class.
- This is very useful, since derived classes are specialized versions of the base class that might behave differently.

```
class B {
public:
    void id() { cout << "Hi, I'm B\n"; }
};

class D : public B {
public:
    void id() { cout << "Hi, I'm D\n"; }
};
```

Redefining: Problem

```
int main() {
    B b;
    D d;
    b.id();
    d.id();
    B& b2 = b;
    B& d2 = d; // polymorphism
    b2.id();   // ok
    d2.id();   // not what we want...
    return 0;
}
```

Output:

```
Hi, I'm B
Hi, I'm D
Hi, I'm B
Hi, I'm B
```

- Unfortunately, **`d2.id()`** invokes the **`id()`**-method of **B**; although **`d2`** actually refers to **`d`** which is of type **D**.
  - How can we make sure that always the **`id()`** method of the actual object type is invoked?

Virtual Member Functions

- If we want to change the behavior of a member function in a derived class, we declare it as **`virtual`**:

```
class B {
public:
    virtual void id() { cout << "Hi, I'm B\n"; }
};

class D : public B {
public:
    void id() { cout << "Hi, I'm D\n"; }
};
```

Output:

```
Hi, I'm B
Hi, I'm D
Hi, I'm B
Hi, I'm D
```

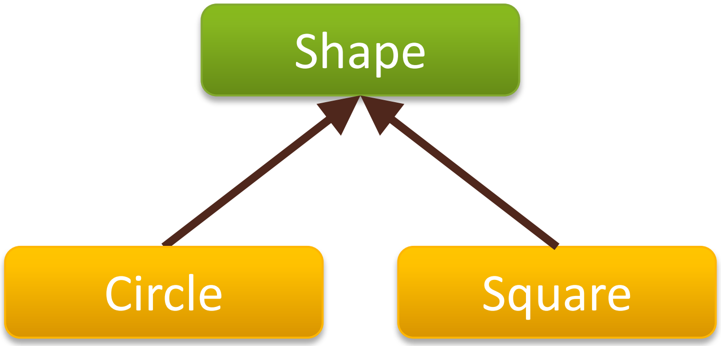
- This enables the compiler to always choose the "right" version of **`id()`**, even when accessed via a reference (or pointer) to a **B** object.

Virtual Member Functions

- Virtual member functions allow us
  - to redefine the behavior of member functions in derived classes **and** (at the same time)
  - to collect objects of derived classes using references or pointers to objects of the base class.
- $\rightarrow$ We can treat objects of (different) derived classes in a uniform way, even without knowing the derived class!
- Examples:
  - `ostream` and its derived classes (`ofstream`, `ostringstream`) in the C++ standard library
  - `draw.cpp` (on the web page)

Abstract Classes

- Suppose we want to write a simple drawing program (for simplicity, we can only draw circles and squares)
- **Class hierarchy:**



- Thus, **Shape** serves as a common interface for drawing, and **Circle** and **Square** shall implement this interface
- Observe that it makes no sense to draw a **Shape** itself

Implementing Shape: First try

```
class Shape {
public:
    virtual void draw() {
        cout << "Error: Cannot draw a Shape!" << endl;
    }
};
```

- We have declared the **`draw()`** method as virtual, since each derived class has to provide its own implementation.
- Its implementation in **`Shape`** simply prints an error message

Pure Virtual Functions

- **Problem:**
  We can still create objects of type **`Shape`** and call their **`draw()`** method, even though this just prints an error message.
- **Better solution:**
  Declare the draw() method of shape to be pure virtual:

```
class Shape {
public:
    virtual void draw() = 0;
};
```

pure virtual function

- Trying to create an object of type **`Shape`** will now cause an error at compile time

Abstract Classes

- A class containing pure virtual functions is called an abstract class
- Abstract classes can only be used as base classes, like:

```
class Circle : public Shape {
    int radius;

public:
    void draw() {
        // ...
    }
    // ...
};
```

- Abstract classes allow us to define an interface to some methods without giving any implementation

Preparations for next week

- Static variables
- Global variables
- Static data members
- Dynamic memory allocation (`new` and `delete`)