Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger
LS 11 Algorithm Engineering

Lecture 8 • Winter 2011/12 • Dec 6

technische universität dortmund
department of computer science

Custom Types

- So far we have used built-in types (**int**, **float**, …) and types defined in the C++ standard library (std::string, std::vector)
- Now we define our own data types
- C++ allows us to define new data types that behave just like built-in and std:: types

Example: A data type for points

- We will implement a custom data type for representing points on the screen
- **Requirements:**
  - a point has an **x**- and **y-coordinate**, both shall be integers
  - possible values for x-coordinates are 0, …, 1919
  - possible values for y-coordinates are 0, …, 1079
  - it shall be possible to **add** two points
  - it shall be possible to **scale** a point by some floating point value
  - it shall be possible to **print** points using the output operator
  - it shall be possible to **read** points using the input operator in a convenient way
- We will start with a basic implementation and add the functionality step-by-step

Data type for points: version 0.1

```
// represents one point on the screen
struct point {
    int x;
    int y;
};

int main()
{
    point p;
    p.x = 2;
    p.y = 45;
    cout << "p = (" << p.x << ',' << p.y << ")\n";

    return 0;
}
```

We define a new data type called point

We declare two data members.
No memory is allocated yet!

We declare a variable **p** of type point.
Now memory (for two ints) is allocated.

We access the data members of **p**.
Note the usage of the .-notation

What about our requirements?

```
point q;
q.x = 3000;   // this shouldn't be possible
q.y = -20;    // and this shouldn't be possible, as well
```

- Solution: Use member functions to ensure integrity of data

Point 0.2: Adding a member function

```
struct point {
    int x;  int y;
    void assign(int new_x, int new_y) {
        if(0 <= new_x && new_x < 1920)
            x = new_x;
        if(0 <= new_y && new_y < 1080)
            y = new_y;
    }
};

int main() {
    point p;
    p.assign(2, 45);
    point q;
    q.assign(3000, 20);
    cout << "q = (" << q.x << ',' << q.y << ")\n";

    return 0;
}
```

We add a member function **`assign`** to **`point`**.
By using **`assign`**, we can make sure that only valid coordinates are assigned

`assign` can only be applied to an object of type **`point`**, and not just on its own.
Again, we have to use the `.`-notation.

Adding a Constructor

- Constructors are invoked when an object is created
  - initializes the object
- They are special member functions:
  - have the same name as their `struct`
  - have no return type
- If we don't implement our own constructors, some constructors are created automatically:
  - Initializing the new object with an object of the same type by memberwise initialization:
    ```
    point p = q;
    ```
  - Default constructor: Initializes each member to its default value:
    ```
    point p;
    ```

Adding a Constructor

- Constructors can be overloaded
- We can define different constructors for a struct, each with different parameter types
- Constructors shall ensure that the instances (variables) of the structure are in a proper state.
- We will add the following constructors to **`point`**:
  - default constructor:
    ```
    point() { } // initializes point to (0,0)
    ```
  -
    ```
    // initializes point to (xc,yc)
    point(int xc, int yc) { }
    ```

Point 1.0: Overloaded Constructors

```
struct point {
    int x;
    int y;

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

    point(int xc, int yc)
      : x(xc), y(yc) {
        truncate();
    }

    void assign(int new_x, int new_y)
    {
        x = new_x;
        y = new_y;
        truncate();
    }

    void truncate() {
        if(x <     0) x = 0;
        if(x >= 1920) x = 1919;
        if(y <     0) y = 0;
        if(y >= 1080) y = 1079;
    }
}; // end of struct point
```

Initialize data members

Using Point 1.0

```
int main()
{
    point p;        Using our default constructor
    cout << "p = (" << p.x << ',' << p.y << ")\n";

    point q(50, 40);        Using our own constructor
    cout << "q = (" << q.x << ',' << q.y << ")\n";

    point r(2000, 78);
    cout << "r = (" << r.x << ',' << r.y << ")\n";

    return 0;
}
```

The story so far…

- What we have done:
  - We defined a data type (structure) for points
  - Using constructors and the assign member function, we can make sure that a point has only valid coordinates
- Not yet possible:
  - Adding two points and scaling a point in a nice way
  - Testing for equality in a nice way
  - Printing and reading points in a convenient way
- Solution: Operator overloading!

Operator Overloading

- C++ allows us to overload the various operators (like ==, <, +) for new data types.
- However, you cannot redefine operators for built-in types
  Would you like to have a new "version" for adding two integers? No you wouldn't!
- We overload an operator by writing a normal function (not a member function) with the operator keyword
  (e.g. `operator==`)
- Example for the equality operator:

```
bool operator==(const point &p, const point &q)
{
    return (p.x == q.x && p.y == q.y);
}
```

Adding overloaded operators for point

```
bool operator==(const point &p, const point &q) {
     return (p.x == q.x && p.y == q.y);
}

bool operator!=(const point &p, const point &q) {
     return !(p == q);
}

point operator+(const point &p, const point &q) {
     return point( p.x+q.x, p.y+q.y );
}

point operator*(float s, const point &p) {
     return point(
        static_cast<int>(s*p.x), static_cast<int>(s*p.y)
     );
}
```

Using our oveloaded operators

```
int main()
{
    point p (35, 5 );
    cout << "p = (" << p.x << ',' << p.y << ")\n";

    point q = p + point(100, 50);
    cout << "q = (" << q.x << ',' << q.y << ")\n";

    point r(3.8f * p);
    cout << "r = (" << r.x << ',' << r.y << ")\n";

    point s(-10.f * p);
    cout << "s = (" << s.x << ',' << s.y << ")\n";

    return 0;
}
```

Overloading Output Operators

- Writing an overloaded output operator is usually easy.
- **But:** compatibility with the various formatting options of the output stream can be a problem
- **Solution:** Use an **`std::ostringstream`**, a string which can be used as output stream.

```
#include <sstream>

ostream &operator<<(ostream &os, const point &p)
{
    ostringstream oss;
    oss << '(' << p.x << ',' << p.y << ')';
    os << oss.str();

    return os;
}
```

Reading points from an input stream

- Reading points is slightly more complicated
  - e.g. we also have to deal with malformed input
- **General strategy for writing input operators**
  - Try to read input, if this is not possible set an error flag
  - The following error flags are available
    - **`goodbit`**: no errors
    - **`eofbit`**: end of file reached
    - **`failbit`**: invalid input (or output)
    - **`badbit`**: unrecoverable error
  - We can set any of the error flags by calling the **`setstate`** member function of an input (or output) stream

Implementing the input operator

```
istream &operator>>(istream &is, point &p)
{
    int x = 0, y = 0;
    char opar = '\0', cpar = '\0', sep = '\0';

    if(!(is >> opar >> x >> sep >> y >> cpar)
       || opar != '(' || sep != ',' || cpar != ')')
       is.setstate(istream::failbit);
    else
       p.assign(x,y);

    return is;
}
```

Using Custom Types

- We can now use our point structure like any built-in type
- E.g. creating a vector of points: **`vector<point>`**

```
vector<point> pv;

// add something to vector pv
for(int i = 0; i < 10; ++i)
   pv.push_back( point(2*i, 2*i+1) );

// iterate over vector and print elements
for(vector<point>::iterator it = pv.begin(); it != pv.end(); ++it)
   cout << *it << endl;
```

- When using iterators, the **`->`** operator is useful:
  - e.g.: `it->x = 10; // is a short-hand for (*it).x = 10;`

Preparations for next week

- Access control:
  - `public` and `private`
  - the `const` modifier for member functions
- Inheritance:
  - classes and derived classes