Object-oriented Programming
for Automation & Robotics

Carsten Gutwenger
LS 11 Algorithm Engineering

Lecture 4 • Winter 2011/12 • Nov 8

technische universität dortmund
department of computer science

How to receive news about the lecture

- I’m posting news (like changes to the web page) in my Google+ stream
- To get these news:
  - (You must have a Google+ profile)
  - First, add me to one of your circles
    (there’s a link to my profile on the OOP web page)
  - Then, I need to add you to a special circle of mine
  - When I get notified and recognize you as one of my students, I will add you automatically to this circle
    (I will not get notified when you add me to the “just follow” circle)
  - Send me an email (with a link to your profile) if I didn’t add you yet

Floating Point Numbers

- There are two data-types for floating point numbers:
  - `float` (single precision, 32-bit)
  - `double` (double precision, 64-bit)
- Support the usual arithmetic operators (`+`, `-`, `*`, `/`)
- Floating point literals are written using a decimal point (`float` is marked by an `f` or `F` at the end):
  - `3.14, 1.0, 25., 3e-10` (type `double`)
  - `3.14f, 1.0f, 25.f, 3e-10f` (type `float`)
- Scientific notation (with exponent `e` or `E`):
  - `3e-10, 5.67e5` (type `double`)
  - `3e-10f, 5.67e5f` (type `float`)
  - Example: `5.67e5` $\triangleq 5.67 \cdot 10^5$
- Caution: The literal `1` is of type `int`!

Printing and reading floating point numbers

- Similar as integers:
  - Use **`cout`** and the output operator **<<** for printing.
  - Use **`cin`** in the input operator **>>** for reading.
- Special manipulators
  - **`fixed`**: prints floating point numbers always in fixed-point notation
  - **`scientific`**: prints floating point numbers always in scientific notation
  - switch back to default behavior:
    `resetiosflags(ios_base::fixed)` or
    `resetiosflags(ios_base::scientific)`
- Precisions of output: **`setprecision(n)`**
  - **default:** n specifies maximum number of meaningful digits to display (before and after decimal point)
  - **fixed or scientific:** display exactly n digits after decimal point (adds trailing zeros if necessary)

Example: Printing floating point numbers

```cpp
double a = 3.1415926534;
double b = 2011.;
double c = 1.0e-10;

cout << right << setprecision(5);
cout << setw(11) << "default:";
cout << setw(15) << a <<
        setw(15) << b <<
        setw(15) << c << endl;
```

```cpp
cout << setw(11) << "fixed:";
cout << fixed <<
        setw(15) << a <<
        setw(15) << b <<
        setw(15) << c << endl;
cout << setw(11) << "scientific:";
cout << scientific <<
        setw(15) << a <<
        setw(15) << b <<
        setw(15) << c << endl;
```

Output:

```
   default:         3.1416           2011         1e-010
     fixed:        3.14159     2011.00000        0.00000
scientific:   3.14159e+000   2.01100e+003   1.00000e-010
```

Increment and Decrement

- Let **`a`** and **`b`** be two **`int`** variables.
  The following statements are equivalent:

```
++a;
--b;
```

and

```
a = a+1;
b = b-1;
```

- These are the pre-increment and pre-decrement operators

Pre- vs. Post-

- Why pre- ?
- There are also post-increment and -decrement operators:

```
a++;
b--;
```

- What is the difference?
- These statements also return a value:
  - pre: returns the value after incrementing/decrementing
  - post: returns the old value before incrementing/decrementing

Example: Pre vs. Post

```
int a = 5, b = 9;

cout << "pre:  ";
cout << ++a << " " << --b << " | ";
cout << a << " " << b << endl;

a = 5; b = 9;

cout << "post: ";
cout << a++ << " " << b-- << " | ";
cout << a << " " << b << endl;
```

Output:

```
pre:  6 8 | 6 8
post: 5 9 | 6 8
```

- Prefer pre-variants (might be slightly faster)
- Use post-variants only if required

Compound Assignment Operators

- We often apply an operator to a variable and then reassign the value to this variable
- In this case we can use compound assignment operators:

  variable op= *expression*;

  where op ∈ { `+`, `-`, `*`, `/`, `%` }
- Examples:

```
a += 2;
b *= 10;
c /= 3 - b;
```

⇔

```
a = a + 2;
b = b * 10;
c = c / (3 - b);
```

Vectors

- Often we need a large supply of variables of the same type
- Suppose we have to read 4 integers and then print their sum:

```
int a, b, c, d;
cin >> a >> b >> c >> d;
cout << a + b + c + d << endl;
```

- This quickly becomes cumbersome: imagine dozens of variables …
- And we need to know the number of variables when we write the program!
- The solution: Use a vector (`std::vector`), which groups a number of variables of the same type together

std::vector

- The data type `std::vector` is a container
- It holds a number of variables of the same type
- These variables are stored sequentially

Example: Working with vectors

```
int main()
{
    int n; cout << "n = "; cin >> n;

    vector<int> v;
    for(int i = 0; i < n; ++i) {
        int x; cin >> x;
        v.push_back(x);
    }

    for(vector<int>::size_type i = 0; i < v.size(); ++i)
        v.at(i) *= 2;

    for(vector<int>::size_type i = 0; i < v.size(); ++i)
        cout << v[i] << endl;

    return 0;
}
```

Example: Step-by-Step

```
vector<int> v;
```

- We create a variable **`v`** of type **`vector<int>`**
- Initially v is empty
- Vectors are typed: all elements are of the same type (**`int`** in our example)

```
v.push_back(x);
```

- We add a new element with value x at the end of the vector
- Vectors can grow automatically (no elements are overwritten)

Example: Step-by-Step

```
for(vector<int>::size_type i = 0; i < v.size(); ++i)
    v.at(i) *= 2;
```

- **`vector<int>::size_type`**
  is a special type for indices of vectors
- **`v.size()`**
  gives the current size of the vector (i.e. number of elements)
- **`v.at(i)`**
  gives us access to the element stored in the vector at position **`i`**
- We can use **`v.at(i)`** like any variable (assign value, use in expressions,…)
- Valid positions are indices between **`0`** and **`v.size()-1`**; any other position results in a runtime error

Example: Step-by-Step

```
for(vector<int>::size_type i = 0; i < v.size(); ++i)
    cout << v[i] << endl;
```

- We can also access an element with the array-operator: **`v[i]`**
- Similar as **`v.at(i)`**, but does not check if we access a legal position
- **Warning:** Trying to access illegal positions in a vector is a very common cause of errors!

Example: Fibonacci numbers with vectors

```
int main()
{
    cout << "n = ";
    vector<int>::size_type n; cin >> n;

    if(n >= 2) {
        vector<int> fib(n+1);
        fib.at(0) = 0;
        fib.at(1) = 1;

        for(vector<int>::size_type i = 2; i <= n; ++i)
            fib.at(i) = fib.at(i-1) + fib.at(i-2);

        for(vector<int>::size_type i = 0; i <= n; ++i)
            cout << "F_" << i << " = " << fib.at(i) << endl;
    }
    return 0;
}
```

Containers and Iterators

- The standard C++ library contains
  - different container classes (e.g. `std::vector` and `std::list`), each with its own advantages and disadvantages
  - algorithms working on containers, e.g. sorting and searching
- Link between containers and algorithms: iterators
  - an iterator points to an element in a container
  - allow us to iterate over the elements in the container
  - every container class has its own iterator type
- Important operations on iterators
  - `++it` advance iterator to the next element
  - `*it` obtain the element to which iterator `it` points
  - comparison of iterators with `==` and `!=`

Example: Sorting a vector

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    vector<int> v(25);

    for(int i = 0; i < 25; ++i)
        v[i] = rand() % 1000;

    sort(v.begin(), v.end());

    for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        cout << setw(3) << *it << endl;

    return 0;
}
```

Returns a random number

Example: Step-by-Step

```
#include <algorithm>
```

- Gives us access to (all) the algorithms in the C++ standard library
- See: http://www.cplusplus.com/reference/algorithm/

```
sort(v.begin(), v.end());
```

- Sorts the range between `v.begin()` and `v.end()` in ascending order
- `begin()` returns an iterator pointing to the first element
- `end()` returns an iterator pointing to one-past-the-last element

Example: Step-by-Step

```
for(vector<int>::iterator it = v.begin(); it != v.end(); ++it)
    cout << setw(3) << *it << endl;
```

- `vector<int>::iterator`
  is the type of an iterator for vectors
- `it = v.begin()`
  We start with the first element
- `it != v.end()`
  We continue until we have visited all elements
- `++it` advances the iterator by one (goes to the next element)
- `*it` returns the value (`int`) of the element to which `it` points

Scope and Lifetime of Variables

- Recall:
  - After `if`, `while`, `for`, only one statement is executed conditionally.
  - If we want to execute more statements conditionally, we need to form a compound statement using `{` and `}`.
  - Everything between a `{` and a matching `}` is called a block.
- The scope of a variable is the block in which it is declared.
- A variable exists (in particular memory is allocated for the variable) only in its scope.

Example for blocks and scope

```
int main()
{
    {
        int a = 1;
    }

    std::cout << a;

    return 0;
}
```

Scope of variable **a**

- This code is wrong!
- When we want to output **a**, the variable does not exist anymore!

Nested Scope and Hidden Variables

- When we declare a variable inside a block using the same name as a variable declared outside this block, the new variable hides the old one.

```
int main()
{
    int a = 40;
    {
        int a = 10;
        cout << a << endl;
    }

    cout << a << endl;

    return 0;
}
```

Output:

```
10
40
```

Nested Scope and Hidden Variables

```
int main()
{
    int a = 40;
    {
        int a = 10;
        cout << a << endl;
    }

    cout << a << endl;

    return 0;
}
```

- variable a is defined in the scope of the main()-function
- variable a is defined in a local scope
- variable a hides variable a
- variable a still exists and has a value

Scope and for-loops

- Recall the translation of **for**-loops to **while**-loops.
- Every **for**-loop statement implicitly creates a block around it
- Therefore, any variable declared in a **for**-statement cannot be used outside the loop!

```
for(int i = 0; i < 10; ++i)
   cout << i << endl;

cout << 2*i << endl;
```

Error: variable i is not declared!

Example with vectors

- Why doesn't the compiler complain about multiple definitions of variable **`i`** here?

```
for(int i = 0; i < n; ++i) {
    int x; cin >> x;
    v.push_back(x);
}
```

```
for(vector<int>::size_type i = 0; i < v.size(); ++i)
    v.at(i) *= 2;
```

```
for(vector<int>::size_type i = 0; i < v.size(); ++i)
    cout << v[i] << endl;
```

Preparations for next week

- File I/O and characters
- Maps (data type std::map)
- Type definitions (typedef)
- Constants
- Types of integers and the sizeof operator