Maciej Sobieraj

Lecture 9

Outline

1. **Operators and enumerated types**
   1. **Overloading operators – the basics**
   2. Enumerated types
   3. Overloaded operators in detail

Operators – a glance at the past

- An **operator** is a symbol designed to **operate** on data
- The “C++” language has a wide range of different operators operating on many different types of data.
- Some of the operators are more universal, some are more specific, some of them are written as **single symbols**, some are **di-graphs** or even **tri-graphs**, other are **keywords**.

Operators – a glance at the past

- One of the possible classifications is based on a number of arguments. We know that there are:
  - **unary** operators
  - **binary** operators
  - **ternary** operators
- Another classification relies on the location of the operator. We distinguish:
  - **prefix** operators (placed in front of their argument)
  - **postfix** operators (placed after their argument)
  - **infix** operators (placed in between their arguments)

Operators – a glance at the past

- "C++" allows the programmer **not only to overload functions (to assign a new implementation to the name of an already existing function) but also to overload operators**.
- Fortunately, the programmer isn't allowed to change the existing operator's meaning (e.g. you can't force "+" to subtract *ints* or *floats*) but you can define new domains for it (e.g. strings are a new domain for "+").
- "C++" doesn't allow you to define completely new operators (e.g. you may not define an operator like this "$#$"). You only can redefine any of the existing operators.

What do we want to achieve?

- We want the “<<” to be a synonym of the *push* method invocation and we want the “>>” to play the role of a *pop* member function.

```
Stack stack(100);
int var;

stack << 200;    // push
stack >> var;   // pop
```

What do we want to achieve?

- The “C++” language **treats overloaded operators as very specific functions**. The number of parameters of these functions must correspond to the number of operator arguments, but it isn’t as simple as you may expect (e.g. a function implementing a new role of a binary operator must not have two arguments).

What do we want to achieve?

- The name of such a specific function is also specific: it consists of a keyword “operator” glued to an operator symbol, e.g. a function implementing the “>>” operator will be named:
  - operator>>
- An operator function may be implemented in two ways:
  - **as a member function of a class** – it’s implicitly assumed that an object of that class is one of the required operator’s arguments
  - **as a “standalone” function** – the function must explicitly specify the types of all its arguments

Implementing the << operator

- Definition of operator function for the “<<” operator (the new face of the *push* member function).
- A new method of declaration into the header file

```
void operator<< (int v) throw(stack_overflow);
```

Implementing the << operator

- the operator must accept different forms of its arguments, like:
  - variable, e.g.
    - stack << VAR;
  - expression, e.g.
    - stack << 2 * VAR;
  - literal, e.g.
    - stack << 2;
  - etc
- this means that **the corresponding parameter of the operator function must be passed by value**

Implementing the << operator

- **the object of the class is the first of the operator's arguments** (the left one, to be precise) so we have nothing more to do except invoke the *push* method with a value from the second (right) operator's argument.

```
void Stack::operator<< (int v) throw (stack_overflow) {
    push(v);
}
```

Implementing the << operator

```
#include "mystack_01.h"
#include <iostream>

using namespace std;

int main(void) {
	int i = 2;
	Stack stk;

	stk << 1;
	stk << 2 * i;
	stk << i;
	cout << stk.pop() << endl;
	cout << stk.pop() << endl;
	cout << stk.pop() << endl;
	return 0;
}
```

Implementing the >> operator

- **We're not allowed (for obvious reasons) to store a value popped from the stack inside a literal or an expression**. We have to put it into a variable (or to be more precise, into an *l-value*).
- We declare the function's only argument as passed by reference.

```
void operator>> (int &v) throw(stack_empty);
```

Implementing the >> operator

```
void Stack::operator>> (int &v) throw(stack_empty) {
	v = pop();
}
```

Implementing the >> operator

```
#include "mystack_02.h"
#include <iostream>

using namespace std;

int main(void) {
	int i = 2;
	Stack stk;
	stk << 1;
	stk << 2 * i;
	stk << i;
	stk >> i; cout << i << endl;
	stk >> i; cout << i << endl;
	stk >> i; cout << i << endl;

	return 0;
}
```

Improving the << operator

- There are two different ways of using the “<<” operator: one implemented by us and one we got from the *streams* library.
  - cout << i << endl;
- he line we quoted above is interpreted by the compiler in the following way:
  - (cout << i) << endl;
- **The expression inside the parentheses returns a reference to a stream** (namely: the *cout* stream) so it can be used (reused) as an argument for the next “<<” operator in a chain.

Improving the << operator

- the operator functions may not be *void* anymore

```
Stack& operator<< (int v) throw(stack_overflow);
```

- **the function returns a reference to its maternal object**

```
Stack& Stack::operator<< (int v) throw (stack_overflow) {
    push(v);
    return *this;
}
```

Improving the << operator

```
#include <iostream>

using namespace std;

int main(void) {
	int i = 2;
	Stack stk;
	stk << 1 << 2 * i;
	stk >> i; cout << i << endl;
	stk >> i; cout << i << endl;
	return 0;
}
```

Improving the >> operator

- We can improve the “>>” operator in the same way.

```
Stack& operator>> (int &v) throw(stack_empty);
```

```
Stack& Stack::operator>> (int &v) throw(stack_empty) {
	v = pop();
	return *this;
}
```

Improving the >> operator

```
#include "mystack_04.h"
#include <iostream>

using namespace std;

int main(void) {
	int i = 2, j;
	Stack stk;
	stk << 1 << 2 * i;
	stk >> j >> i;
	cout << j << endl << i << endl;
	return 0;
}
```

The same effects in a different way

- **We’re allowed to write operator functions outside any class**

```
#include "mystack.h"
#include <iostream>

using namespace std;

Stack& operator<< (Stack &s, int v) throw(stack_overflow) {
	s.push(v);
	return s;
}

Stack& operator>>(Stack &s, int &v) throw(stack_empty) {
	v = s.pop();
	return s;
}

int main(void) {
	int i = 2, j;
	Stack stk;
	stk << 1 << 2 * i;
	stk >> j >> i;
	cout << j << endl << i << endl;
	return 0;
}
```

An indexing operator for the stack

- We'll redefine the meaning of the **indexing operator**.
- We want the indexing to work in this odd way:
  - Stack[0] returns a reference to the element lying at the top of the stack
  - Stack[-1] returns a reference to the element lying below the top of the stack
    etc, etc.
- An attempt to reach for a non-existent stack element will cause an exception to be thrown

An indexing operator for the stack

```
int& operator[] (int index) throw(std::range_error);
```

- Note:
  - the function returns a value of type *int&* as the stack's element type is *int*
  - the function has one argument – the *index*; we pass it by value as the array index doesn't need to be a variable – it may be an expression too

An indexing operator for the stack

```
int& Stack::operator[] (int index) throw(std::range_error) {
	if(index > 0 || index <= -SP)
		throw std::range_error("out of stack");
	return stackstore[SP + index - 1];
}
```

An indexing operator for the stack

```
#include "mystack_06.h"
#include <iostream>

using namespace std;

int main(void) {
	int i = 2, j;
	Stack stk;
	stk << 1 << 2 * i;
	cout << stk[0] << endl << stk[-1] << endl;
	stk[0] = stk[-1] = 0;
	stk >> i >> j;
	cout << i << endl << j << endl;
	return 0;
}
```

An indexing operator for the stack

- The program produces the following output:
  - 4
  - 1
  - 0
  - 0

Outline

1. **Operators and enumerated types**
    1. Overloading operators – the basics
    2. **Enumerated types**
    3. Overloaded operators in detail

Enumerated types – why do we need them?

- **The “C++” language pre-processor offers a method to create symbols which will be replaced by their values during compilation time.**
- These symbols behave like constants and you can’t change their value during run-time, so this might be the ideal way do represent weekdays in a way very clear to humans and very handy for computers.

Enumerated types – why do we need them?

- The *#define* directive has the following syntax:
  - #define symbol string
  - where:
    - the *symbol* is an arbitrary chosen name built like any variable's or function's name; the unofficial but respected convention says that the symbol should contain upper-case letters only, to be easily distinguished from regular variables (our symbols obey this convention)
    - the *string* is just a series of characters
    - the pre-processor will automatically replace each occurrence of the *symbol* with the *string*, but don't forget that this process occurs during compilation time only and its effects are temporary – your source file remains untouched.

Enumerated types – why do we need them?

- Note that you mustn't treat these symbols as a real constant. **None of the symbols have any value** – they're always treated as strings. Consider that the following snippet will cause the *x* variable to be assigned the value of 0, not 2!
  - #define ALPHA 2-1
  - #define BETA ALPHA*2
  - **int** x = BETA;

Enumerated types – why do we need them?

```
#define SUNDAY 0
#define MONDAY       1
#define TUESDAY      2
#define WEDNESDAY 3
#define THURSDAY     4
#define FRIDAY       5
#define SATURDAY     6
```

Enumerated types – why do we need them?

```
int big_day = MONDAY;
:
:
++big_day;
:
:
if(big_day == SUNDAY) { ... }
:
:
big_day = -1;
```

Enumerated types – how do we use them?

```
enum weekday {SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY};
```

- Let’s explain what we’ve done:
  - the *enum* keyword begins the declaration of the type
  - ‘weekday’ is the name of the new type being created; the name of the type must obey the rules regarding names in general
  - next goes **a list of all of the values creating the new type**, separated by commas and enclosed in curly brackets
  - the compiler will implicitly assign the value of *0* to the first element of the list
  - any symbol except the first one will be assigned **a value greater by one** than the previous element in the list

Enumerated types – how do we use them?

- The enumerated type is treated in a very specific way. **When a value of the type is assigned to any *int* value, everything is O**K and the compiler accepts it without reservations.
- In general, any ***enum* type value is implicitly promoted to the *int* type** when used in a context requiring integral values e.g. when used in conjunction with operators like +, -, etc.

```
int day = SUNDAY;
```

Enumerated types – how do we use them?

- When the enumerated type plays the role of an *l-value*, the situation changes. Assigning an *int* value to it will provoke a compilation warning as the compiler recognizes these assignments as a potential risk to data integrity.

```
weekday day = 0;
```

Enumerated types – how do we use them?

- You may have to modify the assignment in the following way:
  - weekday day = static_cast<weekday>(0);
- or use an alternative way of type-casting like this:
  - weekday f = (weekday)0;
- Both ways are acceptable in this context.

Enumerated types – how do we use them?

- In general, *enum* type values are *ints* and may be used as arguments in any operations accepting *ints*. Internally they're stored just like *ints* too. E.g., the following line
  - `cout << SUNDAY << endl;`

Enumerated types – how do we use them?

- Any of the elements of the *enum* type list may be followed by the ‘=’ sign and an expression resulting in an *int* value.
- In this case, the symbol will be assigned the value specified by the expression (default rules are omitted here).

```
enum Symbols {ALPHA = -1, BETA = 1, GAMMA};
```

Enumerated types – how do we use them?

- More than one symbol of the *enum* type may have been assigned with the same value; in other words, some (or even all) symbols may represent identical values.
- In the following example the A and C symbols represent the same value: 1.

```
enum letters { A = 1, B = 0, C, D };
```

Enumerated types – how do we use them?

- **All symbols in the list must be unique**, even if they're assigned the same value.

```
enum letters { A = 1, B = 0, C, D, A = 1 };
```

- In general, ***enum* type symbols must be unique across a namespace**, i.e. two different *enum* types can't use identical symbols.

```
enum Animals {DOG, CAT, CHUPACABRA};
enum Commands {LS, CD, CAT};
```

Enumerated types – how do we use them?

- You can avoid this conflict by putting one or both of the conflicting *enum* types inside a separate class/classes

```
class Animals {
public:
	enum names {DOG, CAT, CHUPACABRA};
};

class Commands {
public:
	enum names {LS, CD, CAT};
};

int main(void) {
	Animals::names a = Animals::CAT;
	Commands::names c = Commands::CAT;
	return 0;
}
```

Enumerated types – how do we use them?

- Using *enum* types may protect us from many threats, but some of them are still serious.

```
enum weekday {SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY};

int main(void) {
	weekday d = SATURDAY;
	d = weekday(d + 1);
	return 0;
}
```

- The + operator is unaware of weekdays at all and may skip from *SATURDAY* to literally nowhere, leaving the permitted type domain.

Enumerated types – how do we use them?

```
enum weekday {SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY};

weekday operator+(weekday day, int days) {
	return weekday((int(day) + days) % 7);
}

int main(void) {
	weekday d = SATURDAY;
	d = d + 1;
	return 0;
}
```

Enumerated types – how do we use them?

```
#include <iostream>

using namespace std;

enum weekday {SUNDAY,MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY};

weekday operator+(weekday day, int days) {
    return weekday((int(day) + days) % 7);
}

ostream& operator<< (ostream &strm, weekday day) {
    switch(int(day)) {
    case SUNDAY:    strm << "SUNDAY"; break;
    case MONDAY:    strm << "MONDAY"; break;
    case TUESDAY:   strm << "TUESDAY"; break;
    case WEDNESDAY: strm << "WEDNESDAY"; break;
    case THURSDAY:  strm << "THURSDAY"; break;
    case FRIDAY:    strm << "FRIDAY"; break;
    case SATURDAY:  strm << "SATURDAY"; break;
    default:        strm << "Somewhere inside the depths of time..."; break;
    }
    return strm;
}

int main(void) {
    weekday d = SATURDAY;
    d = d + 16;
    cout << d << endl;
    return 0;
}
```

Outline

1. **Operators and enumerated types**
   1. Overloading operators – the basics
   2. Enumerated types
   3. **Overloaded operators in detail**

Number of arguments

- The **number of arguments of the overloaded operator function is strictly restricted** and it's precisely defined by the context in which the function exists.
- Two aspects are decisive:
  - the **location** in which the operator function is defined
  - the **operator** it overloads

Number of arguments

		The number of arguments of the operator you want to overload	
		1	2
The way the operator is implemented	as a global (standalone) function	1	2
	as a class member function	0	1

What you mustn't do

- Don't forget that you're not allowed to:
  - **define new operators** (those that are not known in the "C++" language)
  - **change the priority** of the redefined operators
  - **overload operators working with standard data types**

Arithmetic operators

Operators	+ - * / %
May be implemented as a global function?	YES
May be implemented as a member function?	YES
Type of return value	Depending on context

Arithmetic operators

```
#include <iostream>
using namespace std;
class V {
public:
	float vec[2];
	V(float a0, float a1) { vec[0]=a0; vec[1]=a1; }
	V operator+(V &arg) {
		V res(0.0f,0.0f);
		for(int i = 0; i < 2; i++)
			res.vec[i] = vec[i] + arg.vec[i];
		return res;
	}
};
float operator*(V &left, V &right) {
	float res = 0.0;
	for(int i = 0; i < 2; i++)
		res += left.vec[i] * right.vec[i];
	return res;
}
int main(void) {
	V v1(0.0f, 1.0f), v2(1.0f, 0.0f), v3(0.0f, 0.0f);
	float x;
	v3 = v1 + v2;
	x = v1 * v2;
	cout << "(" << v3.vec[0] << ", " << v3.vec[1] << ")" << endl;
	cout << x << endl;
	return 0;
}
```

Arithmetic operators

- Note that the first function returns a vector, while the second returns a scalar. The first of the functions is a member function, and the second is global.
- The program produces the following output to the screen:
  - (1, 1)
  - 0

Arithmetic operators

- Note that the first of the newly defined operators **may be chained**, e.g. in the following way:
  - v3 = v1 + v2 + v3;
- while the second one may not be treated in the same way – why? Can you explain it?

Bitwise operators

Operators	^ | & ~ << >>
May be implemented as a global function?	YES
May be implemented as a member function?	YES
Type of return value	Depending on context

Bitwise operators

```
#include <iostream>
using namespace std;
class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	V operator>>(int arg) {
		V res(vec[0],vec[1]);
		for(int i = 0; i < 2; i++)
			res.vec[i] >>= arg;
		return res;
	}
};
int operator~(V &arg) {
	int res = 1;
	for(int i = 0; i < 2; i++)
		res *= arg.vec[i];
	return res;
}
int main(void) {
	V v(15, 7);
	v = v >> 1;
	cout << "(" << v.vec[0] << ", " << v.vec[1] << ")" << endl;
	cout << ~v << endl;
	return 0;
}
```

Bitwise operators

- Now the *V* class is able to bitwise right shift each of its elements and to evaluate their product (it's a rather unusual use of the ~ operator).
- The program produces the following output:
  - (7, 3)
  - 21

Assignment operator

Operators	=
May be implemented as a global function?	NO
May be implemented as a member function?	YES
Type of return value	A reference to an object or an l-value in general

Assignment operator

```
#include <iostream>
using namespace std;
class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	V(void) { vec[0]=vec[1]=0; }
	V& operator=(V &arg) {
		for(int i = 0; i < 2; i++)
			vec[i] = arg.vec[1 - i];
		return *this;
	}
};
int main(void) {
	V v1(4, 8), v2;
	v2 = v1;
	cout << "(" << v2.vec[0] << ", " << v2.vec[1] << ")" << endl;
	return 0;
}
```

Assignment operator

- The program produces the following output:
  - (8, 4)
- Try to guess the output of the program when the main function takes the following form:
  - **int** main(**void**) {
  - V v1(4, 8), v2, v3;
  - v2 = v3 = v1;
  - cout << "(" << v2.vec[0] << ", " << v2.vec[1] << ")" << endl;
  - **return** 0;
  - }

Relational operators

Operators	== != > >= < <=
May be implemented as a global function?	YES
May be implemented as a member function?	YES
Type of return value	boolean

Relational operators

```
#include <iostream>
using namespace std;
class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	bool operator==(V &arg) {
		for(int i = 0; i < 2; i++)
			if(vec[i] != arg.vec[i])
				return false;
		return true;
	}
};
bool operator>(V &l, V &r) {
	return l.vec[0]+l.vec[1] > r.vec[0]+r.vec[1];
}
int main(void) {
	V v1(4, 8), v2(3, 7);
	cout << (v1 == v2 ? "true" : "false") << endl;
	cout << (v1 > v2 ? "true" : "false") << endl;
	return 0;
}
```

Relational operators

- The program emits the following text:
  - false
  - true

Logical operators

Operators	! && ||
May be implemented as a global function?	YES
May be implemented as a member function?	YES
Type of return value	Boolean

Logical operators

```
#include <iostream>
#include <cmath>
using namespace std;
class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	bool operator&&(V &arg) {
		return abs(vec[0]) + abs(vec[1]) > 0 &&
			abs(arg.vec[0]) + abs(arg.vec[1]) > 0;
	}
};
bool operator!(V &v) {
	return v.vec[0] * v.vec[1] != 0;
}
int main(void) {
	V v1(4, 8), v2(3, 7);
	cout << (v1 && v2 ? "true" : "false") << endl;
	cout << (!v1 ? "true" : "false") << endl;
	return 0;
}
```

Logical operators

- The program will emit the following two lines:
  - true
  - true

Compound assignment operators

Operators	+= -= *= %= /= &= |= ^= >>= <<=
May be implemented as a global function?	NO
May be implemented as a member function?	YES
Type of return value	A reference to an object or an l-value in general

Compound assignment operators

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

class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	V& operator+=(V &arg) {
		for(int i = 0; i < 2; i++)
			vec[i] += arg.vec[i];
		return *this;
	}
};
V& operator+(V &left, V &right) {
	V *res = new V(0, 0);
	for(int i = 0; i < 2; i++)
		res->vec[i] = left.vec[i] + right.vec[i];
	return *res;
}
int main(void) {
	V v1(0, 0), v2(1, 2), v3(3, 4);
	v1 = v2 + v3;
	v1 += v1;
	cout << "(" << v1.vec[0] << ", " << v1.vec[1] << ")" << endl;
	return 0;
}
```

Compound assignment operators

- The program outputs:
  - (8, 12)

Prefix increment and decrement operators

Operators	++--
May be implemented as a global function?	NO
May be implemented as a member function?	YES
Type of return value	A reference to an object or an l-value in general

Prefix increment and decrement operators

```
#include <iostream>
using namespace std;
class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	V& operator++(void) {
		for(int i = 0; i < 2; i++)
			vec[i]++;
		return *this;
	}
};
int main(void) {
	V v1(1, 2);
	++v1;
	cout << "(" << v1.vec[0] << ", " << v1.vec[1] << ")" << endl;
	return 0;
}
```

Prefix increment and decrement operators

- The example program shows you an overloaded prefix ++ which affects all vector elements. The program outputs:
  - (2, 3)

Postfix increment and decrement operators

Operators	++--
May be implemented as a global function?	NO
May be implemented as a member function?	YES
Type of return value	A reference to an object or an l-value in general

Postfix increment and decrement operators

```
#include <iostream>
using namespace std;
class V {
public:
	int vec[2];
	V(int a0, int a1) { vec[0]=a0; vec[1]=a1; }
	V operator++(int none) {
		V v(vec[0],vec[1]);
		for(int i = 0; i < 2; i++)
			++vec[i]
		return v;
	}
};
int main(void) {
	V v1(2, 3);
	v1++;
	cout << "(" << v1.vec[0] << ", " << v1.vec[1] << ")" << endl;
	return 0;
}
```

Postfix increment and decrement operators

- The postfix form of the ++/-- has to be implemented as **a one-parameter operator function** (sic! note that the parameter of type *int* is a complete dummy and you mustn't use it within the function) and since it serves the object before it's affected by the modification, it should return **a copy of the unmodified object**.
- The presence of the dummy *int* parameter is **the only trait that allows the compiler to distinguish between prefix and postfix overloaded operators**.

Postfix increment and decrement operators

- The example program here → outputs the following text:
  - (3, 4)