

1/27

Polymorphism

Code that works for many types. ad-hoc polymorphism (overloading) - this session The version executed is determined statically from the types of the arguments (Savitch 8.1; Stroustrup 7.4,11; Horstmann 13.4) parametric polymorphism (genericity) - next session A single version, parameterized by types, is used (Savitch 16.1-2; Stroustrup 13.2-3; Horstmann 13.5) subtype polymorphism (dynamic binding) - session 7 The version executed is determined dynamically. (Savitch 14,15; Stroustrup 12; Horstmann 14)

Programming in C++

ge's, UoL)

2/27

Overloading

Term symbol is overloaded...

A single symbol has multiple meanings.

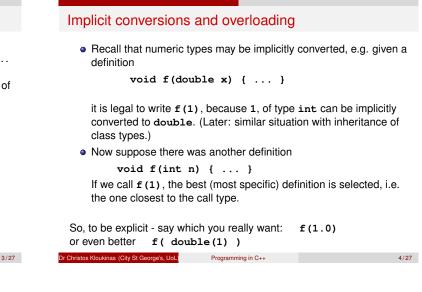
The meaning of a particular use is statically determined by the types of its arguments.

The following may be overloaded in C++:

- constructors (as in Java) often useful.
- member functions (or methods, as in Java) a dubious (and dangerous) feature.
- independent functions ditto.
- operators heavily used in the standard libraries. ۲ Operator overloading makes for concise programs, but overuse may impair readability.

Programming in C++

```
stos Kloukinas (City St George's, UoL)
```



Ambiguity Given the definitions void f(int i, double y) { }
void $f(int i double v)$ {
<pre>void f(double x, int j) { }</pre>
the following is rejected by the the compiler:
f(1, 2); // ambiguous!
We could get around this by also defining
<pre>void f(int i, int j) { }</pre>
Then every application would have a best match.



You're writing programs for PEOPLE first!

So, DOCUMENT THEM!

5/27

f(int(1), double(2));

ristos Kloukinas (City St George's, UoL) Programming in C++

$Over_{Loading}^{Riding}$ – Write fewer if 's with OOP!	
--	--

<pre>void move(person p) { if (p isA driver) { } else if (p isA cyclist) { } else if (p isA pilot) { } else if (p isA pilot) { } else if (p isA pilot) { } else { //*DEFAULT* }</pre>	Overriding – compare:	
<pre>void f(x) { if (x isA double) { void f(double x) {} void f(float x) {} } else if (x isA float) { void f(float x) {} } }</pre>	<pre>void move(person p) { if (p isA driver) { } else if (p isA cyclist) { } else if (p isA pilot) { }</pre>	<pre>void move(){} } class driver :person{ void move(){} } class cyclist :person{ void move(){} } class pilot :person{</pre>
<pre>} else if (x isA int) { } else {assert(0);}//*ERROR* They allow us to write if/then/else's better - the compiler does it! Dr Christos Kloukinas (City St George's, UoL Programming in C++ 6/27</pre>	<pre>void f(x) { if (x isA double) { else if (x isA float) { else if (x isA int) { else if (x isA int) { else {assert(0);}//*ERROR They allow us to write if/then/else's b</pre>	<pre>void f(float x) {} void f(int x) {} //*NO* (runtime) *ERROR*!!! * vetter - the compiler does it!</pre>

Overloaded equality

In C++, we can compare values of built-in types:

int i;
if (i == 3) ... // [*]

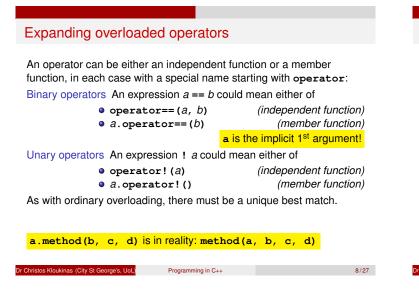
We can also compare objects:

string s1, s2;
if (s1 == s2) ...

And similarly for **vectors**. The **==** operator is **overloaded**:

special definitions have been given for string, vector and many other types.

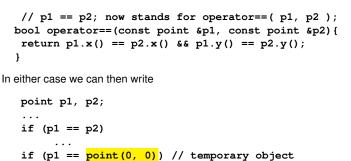
[*] Prefer (3 == i), because "if (i = 3)" is valid C++ (and it's always true...) r Christos Kloukinas (City St George's, UoL Programming in C++ 7/27



Comparing points			
<pre>class point { int _x, _y; public: point(int x, int y) : .</pre>	_x(x), _y	(y) { }	
<pre>int x() const {return { int y() const {return { // p1 == p2; stand bool operator==(const return _x == px }// methods can read p };</pre>	_y;} s for p1.0 point &p) && _y ==]	<pre>perator== const { p.y();</pre>	
 Use const as much as possib Put it in by default, only remove If you need a non-const version const one (for use with constant) 	e it if you (<i>rea</i> on, see if you	• /	vide a
Christos Kloukinas (City St George's, UoL) Programm	ng in C++		9/27

An alternative definition

We could instead have defined an independent function:



```
// (only works if second parameter is *const*)
```

Programming in C++



```
A note on types
```

- The language does not enforce any constraints on the argument types and return type of operator==, or any other operator.
- It is conventional that the arguments have the same type and the result type is **bool**.
- It is also conventional that the == operator should define an equivalence relation.
- Departing from these conventions is permitted by the language, but will be very confusing for anyone trying to understand your code (including a future you).

Equivalence Relation *R*:

10/27

Reflective	$x \mathrel{R} x$	
Symmetric	$x \ R \ y \to y \ R \ x$	
Transitive	$x \mathrel{R} y \land y \mathrel{R} z \to x \mathrel{R} z$	
Christos Kloukinas (Cit	ty St George's, UoL) Programming in C++	

Other comparison operators

The <utility> header file (which is included by <string>, <vector> and other data types) defines

- a != b as ! (a == b)
- a>b as b<a
 a<=b as !(b<a)
 a>=b as !(a<b)

So usually we need only define == and <, but we can also define the others if required.

You need to declare:

using namespace std::rel_ops;

Dr Christos Kloukinas (City St George's, UoL) Programming in C++

12/27

14/27

Dr

Operators available for overloading

Only built-in operators can be overloaded:

a

unary ~	!	+	-	&	*	++		++	
binary +	-	*	/	용	^	æ	I	<<	>>
+=	-=	*=	/=	% =	^=	&=	=	<<=	>>=
==	! =	<	>	<=	>=	88	11		
=	,	->*	->	()	[]				

Their precedence and associativity can't be changed, so the expressions

$$+ b + c * d$$
 (a + b) + (c * d)

are always equivalent, no	matter how the ope	erators are o	verloaded.	
++a; is a.operator+	+();			
a++; is a.operator+	+(int);//dummy	argument	(ignored)	
 Christos Kloukinas (City St George's, UoL)	Programming in C++		13/27	

Output of built-in types

Consider

cout << "Total = " << sum << '\n';

This is equivalent to

((cout << "Total = ") << sum) << '\n';

- The operator << is overloaded in iostream, not in the C++ language.
- It associates to the left.

istos Kloukinas (City St George's, UoL)

• It is defined as a member function of **ostream**, and returns the modified **ostream**.

Programming in C++

The << operator

istos Kloukinas (City St George's, UoL)

- The built-in meaning of << is bitwise left shift of integers, so that the expression 5 << 3 is equal to 40.
- It associates to the left, so 5 << 2 << 1 is also equal to 40.
- It was selected for stream output for its looks. Luckily it associated the right way.
- Different overloadings of the same symbol need not have related meanings, or even related return types.

Bitv	vise left s	hift	
5 << 0	101	=	5
5 << 1	1010	=	10
5 << 2	10100	=	20
5 << 3	101000	=	40
x << y = x >> y =	$x * 2^y \\ x * 2^{-y}$	=	$x/2^y$

Programming in C++

```
The ostream class
class ostream {
public:
         ostream& operator<<(char c);</pre>
         ostream& operator<<(unsigned char c);</pre>
         ostream& operator<<(int n);</pre>
         ostream& operator<<(unsigned int n);</pre>
         ostream& operator<<(long n);</pre>
         ostream& operator<<(float n);</pre>
         ostream& operator<<(double n);</pre>
          . . .
};
In the string header file an independent function:
ostream& operator<<(ostream &out, const string &s);</pre>
```

Why not define it as a member function???

ostream& operator<<(ostream &out);</pre>

Christos Kloukinas (City St George's, UoL) Programming in C++

16/27

Programming in C++ 2024-11-13 —The ostream class Why not define printing string objects as a member function?

The writer of the string class cannot modify the ostream class. So if they want to declare it as a member function they can only do so within the class string.

But then the meaning of the operator changes - instead of writing cout << s; we would have to write s << cout; - not what we want!</pre>

Do you understand why we'd have to write s << cout; to print a string s on cout if we'd have defined operator << as a member function of class string?

If you do not, start reading again from slide "Expanding overloaded operators" (slide 8) - repeat until it's clear.

Output of a user-defined type

```
class point { int _x, _y;
 public:
  point(x, y) : _x(x), _y(y) { }
int x() const { return _x; }
  int y() const { return _y; }
};
```

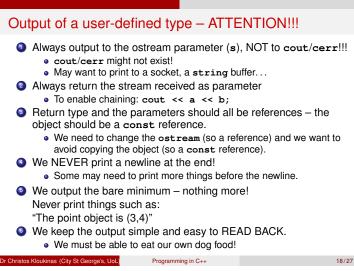
The output operator for **points** is defined as a non-member function:

```
ostream& operator<<(ostream &s, const point &p) {</pre>
  return s << '(' << p.x() << ',' << p.y() << ')';
}
```

Again - why as a non-member function ???

r Christos Kloukinas (City St George's, UoL) Programming in C++

17/27



Using various versions of the << operator

Suppose we have an expression $a \ll b$, where a has type **A**, and b has type **B**. Then the relevant definition of \ll could be either

• a method of class A taking one argument of type B:

ReturnType A::operator<<(B x)

• or an independent function (not a method in a class) taking two arguments of types **A** and **B**:

ReturnType operator<<(A x, B y)

For example the following uses a mixture of these:

os Kloukinas (City St George's, UoL) Programming in C++

point p(2,3); cout << "The point is " << p << '\n';</pre>

Can you identify which occurrences of the << operator are independent functions and which are member functions? (Hint: Think which types were already known to whomever wrote the ostream class.)

19/27

On accessing private state: Friend (or NOT)

An accidental consequence of the way operators are defined in C++:

- An operator defined as a member function has access to the private and protected fields of its first argument, but not its second (when the second is an object of a different class).
- Sometimes this is not what we want (e.g. for << and >> of user-defined types).
- One work-around is to declare the operator as a friend of the second class.
- Even *better* to use a helper member function:

class point {
 public:
 ostream& print_on(ostream &s) const {//*CONST* !!!
 return s << '(' << _x << ',' << _y << ')'; }
};
ostream& operator<<(ostream &s, const point &p) {
 return p.print_on(s); }
stos Kloukinas (Cily St George's, Uol Programming in C++ 20/27</pre>

Input of built-in types

Input is almost the mirror image of output:

int x, y, z; cout << "Please type three numbers: "; cin >> x >> y >> z;

- Again >> is overloaded: it knows what to look for based on the type of its argument.
- It also associates to the left, and returns an istream.
- By default, >> will skip white space before the item; in this mode you will not see a space, newline, etc.

Programming in C++

```
Christos Kloukinas (City St George's, UoL)
```

21/27

The istream class

```
class istream : virtual public ios {
  public:
        istream& operator>>(char &c);
        istream& operator>>(unsigned char &c);
        istream& operator>>(int &n);
        istream& operator>>(unsigned int &n);
        istream& operator>>(long &n);
        istream& operator>>(float &n);
        istream& operator>>(double &n);
        ...
```

};

In the string header file, as an *independent* function:

istos Kloukinas (City St George's, UoL) Programming in C++

istream& operator>>(istream &in, string &s);

The state of an istream The following methods of istream test its state: bool eof() the end of the input has been seen. bool fail() the last operation failed. bool good() the next operation might succeed. (Equivalent to ! eof() && ! fail().) bool bad() the stream has been corrupted: data has been lost (data was read but not stored in an argument). (Implies fail(), but not vice-versa.) A test "if (s)" is equivalent to "if (! s.fail())"

Programming in C++

Kloukinas (City St George's, UoL)

23/27

25/27

Input of a user-defined type
<pre>istream& operator>>(istream &s, point &p) { int x, y; char lpar, comma, rpar;</pre>
<pre>if (s >> lpar) { //not met EOF (End Of File)</pre>
<pre>if ((s >> x >> comma >> y >> rpar) && (lpar == '(' && comma == ',' && rpar == ')')) p = point(x, y); // *constructor*, not setters! else</pre>
<pre>s.setstate(ios::badbit); //read failed</pre>
} return s; }
When "if (s >> lpar)" fails, that means there is no more input. We have not read any data so far, so have not corrupted the input.

Therefore, we simply return the input stream. Dr Christos Kloukinas (City St George's, UoL) Programming in C++

Input of a user-defined type - ATTENTION !!!

- Always read from the stream received as parameter NEVER cin!
 cin may not exist!
 - May want to read from a file/buffer/socket...
- Always return the stream received as parameter
 To allow checking for input success.
 - To allow for chaining.
- Return and all parameters should be references (non-const).
- Set the badbit if there's a problem (*i.e.*, you've read something but cannot use it to set your object) failing to read *anything at all* because of an EOF is NOT a problem.
- Always read what you print always (so, keep the format simple!).
- NEVER use getline() you're corrupting the stream!
- NEVER read into a string and parse that stream corruption!
- NEVER, EVER print anything!
- Prefer constructors over setter member functions.
- Avoid setters altogether not very OO. Same with getters... stos Kloukinas (City SI George's, UoL) Programming in C++

Getters/Setters - Why Not Avoid getters Objects should be asked to do tasks themselves: point1.move(3,5); shape2.scale(.5); employee3.clock_in(log_register); etc. When you're using getters, you end up doing the task yourself using the state data you got. But that's procedural, not OO programming... Avoid setters (OK, you can write ForTran in any language...) Object's state should only change because of actions they've performed on your behalf, not because you've done a task and are now giving them the results. Don't spoon-feed your objects - they can take care of themselves. Setters need to preserve the class invariance. ۵ Much easier to get this right once (in the constructors) and re-use the constructors from that point on.

- Delegate! "What can I ask an object of this class to do for me?"
- Dr Christos Kloukinas (City St George's, UoL) Programming in C++
- 26/27

Next session

- Genericity (parametric polymorphism)
- Template classes and functions in C++.
- Reading: Savitch 16.1–2; Stroustrup 13.2–3; Horstmann 13.5.
- Introducing the Standard Template Library: some container classes.
- Reading: Savitch 19.1; Stroustrup 16.2.3,16.3; Horstmann 13.5.

ristos Kloukinas (City St George's, UoL) Programming in C++

27/27

Programming in C++ 2024-11-13 └-Next session **Final Notes**

- a + b, can be either a. operator+(b) or operator+(a, b). All methods receive the current object (*this) as their implicit first argument.
- Avoid friend functions use helper methods. "Treat your friend as if he might become an enemy." - Publilius Syrus, 85-43 BC.
- Output: Read again slides 17-18. Repeat.
- Input: Read again slides 24-25. Repeat.
- More on Operators: https://www.cplusplus.com/doc/tutorial/operators/
- More on Operator overloading: https://en.cppreference.com/w/cpp/language/operators
- More on friends: https://isocpp.org/wiki/faq/friends