Programming in C++

Session 3 – Overloading

Dr Christos Kloukinas

City St George's, UoL
https://staff.city.ac.uk/c.kloukinas/cpp
(based on slides originally produced by Dr Ross Paterson)

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)

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.
  Operator overloading makes for concise programs, but overuse may impair readability.

Implicit conversions and overloading

- Recall that numeric types may be implicitly converted, e.g. given a definition

  ```
  void f(double x) { ... }
  ```

  it is legal to write `f(1)`, because `1`, of type `int` can be implicitly converted to `double`. (Later: similar situation with inheritance of class types.)
- Now suppose there was another definition

  ```
  void f(int n) { ... }
  ```

  If we call `f(1)`, the best (most specific) definition is selected, i.e. the one closest to the call type.

So, to be explicit - say which you really want: `f(1.0)`
or even better `f( double(1) )`

Ambiguity

Given the definitions

```
void f(int i, double y) { ... }
void f(double x, int j) { ... }
```

the following is rejected by the the compiler:

```
f(1, 2);      // ambiguous!
```

We could get around this by also defining

```
void f(int i, int j) { ... }
```

Then every application would have a best match.

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

You're writing programs for PEOPLE first!

So, DOCUMENT THEM!

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

Over$^{\text{Riding}}_{\text{Loading}}$ – Write fewer `if`'s with OOP!

Overriding – compare:

```
void move(person p) {
if (p isA driver) {
} else if (p isA cyclist) {
} else if (p isA pilot) {
} else { //*DEFAULT* }
```

```
class person {//*DEFAULT*
 void move(){...} }
class driver :person{
  void move(){...} }
class cyclist :person{
  void move(){...} }
class pilot  :person{
  void move(){...} }
```

Overloading – compare:

```
void f( x ) {
if (x isA double) {
} else if (x isA float) {
} else if (x isA int) {
} else {assert(0);}//*ERROR*
```

```
void f(double x) {...}
void f(float  x) {...}
void f(int    x) {...}
//*NO* (runtime) *ERROR*!!!
```

They allow us to write if/then/else's better – the compiler does it!

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 `vector`s.
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. . .)

Expanding overloaded operators

An operator can be either an independent function or a member function, in each case with a special name starting with `operator`:

Binary operators An expression $a == b$ could mean either of

- `operator==(a, b)` *(independent function)*
- `a.operator==(b)` *(member function)*

a is the implicit 1st argument!

Unary operators An expression `!` a could mean either of

- `operator!(a)` *(independent function)*
- `a.operator!()` *(member function)*

As with ordinary overloading, there must be a unique best match.

`a.method(b, c, d)` is in reality: `method(a, b, c, d)`

Comparing points

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

    int x()  const {return _x;}//p1.x();p1 as if const
    int y()  const {return _y;}
        // p1 == p2; stands for p1.operator==( p2 );
    bool operator==( const  point &p)  const  {
        return _x == p._x && _y == p.y();
    }// methods can read private fields
};
```

- Use `const` as much as possible.
- Put it in by default, only remove it if you (***really***) need to.
- If you need a non-`const` version, see if you can also provide a `const` one (for use with constant objects).

An alternative definition

We could instead have defined an independent function:

```
   // p1 == p2; now stands for operator==( p1, p2 );
  bool operator==(const point &p1, const point &p2){
   return p1.x() == p2.x() && p1.y() == p2.y();
  }
```

In either case we can then write

```
   point p1, p2;
   ...
   if (p1 == p2)
        ...
   if (p1 == point(0, 0)) // temporary object
   // (only works if second parameter is *const*)
...
```


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:

Reflective $x \; R \; x$

Symmetric $x \; R \; y \rightarrow y \; R \; x$

Transitive $x \; R \; y \;\; \wedge \;\; y \; R \; z \rightarrow x \; R \; z$

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;
```


Operators available for overloading

Only built-in operators can be overloaded:

unary	~	!	+	−	&	*	++	−−	++	−−
binary	+	−	*	/	%	^	&	|	<<	>>
	+=	−=	*=	/=	%=	^=	&=	|=	<<=	>>=
	==	!=	<	>	<=	>=	&&	||		
	=	,	->*	->	()	[]				

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

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

are always equivalent, no matter how the operators are overloaded.

++a; is a.operator++();
a++; is a.operator++(int);//*dummy argument (ignored)*

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

The << operator

- 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.

Bitwise left shift			
5 << 0	101	=	5
5 << 1	1010	=	10
5 << 2	10100	=	20
5 << 3	101000	=	40

$$x \texttt{<<} y = x * 2^y$$
$$x \texttt{>>} y = x * 2^{-y} = x / 2^y$$

The `ostream` class

```
class ostream {
public:
        ostream& operator<<(char c);
        ostream& operator<<(unsigned char c);
        ostream& operator<<(int n);
        ostream& operator<<(unsigned int n);
        ostream& operator<<(long n);
        ostream& operator<<(float n);
        ostream& operator<<(double n);
        ...
};
```

In the **`string`** header file an independent function:

```
ostream& operator<<(ostream &out, const string &s);
```

Why not define it as a member function???

```
ostream& operator<<(ostream &out);
```


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!

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 **`point`**s is defined as a non-member function:

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

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

Output of a user-defined type – ATTENTION!!!

1. Always output to the ostream parameter (**`s`**), NOT to **`cout`**/**`cerr`**!!!
   - **`cout`**/**`cerr`** might not exist!
   - May want to print to a socket, a **`string`** buffer...
2. Always return the stream received as parameter
   - To enable chaining: **`cout << a << b;`**
3. Return type and the parameters should all be references – the object should be a **`const`** reference.
   - We need to change the **`ostream`** (so a reference) and we want to avoid copying the object (so a **`const`** reference).
4. We NEVER print a newline at the end!
   - Some may need to print more things before the newline.
5. We output the bare minimum – nothing more!
   Never print things such as:
   "The point object is (3,4)"
6. We keep the output simple and easy to READ BACK.
   - We must be able to eat our own dog food!

Using various versions of the << operator

Suppose we have an expression *a* << *b*, where *a* has type **A**, and *b* has type **B**. Then the relevant definition of << 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:

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

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.)*

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); }
```


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.

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:

```
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())**"

Input of a user-defined type

```
istream& operator>>(istream &s, point &p) {
  int x, y;
  char lpar, comma, rpar;
  if (s >> lpar) { //not met EOF (End Of File)
    if ((s >> x >> comma >> y >> rpar) &&
        (lpar == '(' && comma == ',' && rpar == ')'))
      p = point(x, y); // *constructor*, not setters!
    else
      s.setstate(ios::badbit); //read failed
  }
  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.

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

1. Always read from the stream received as parameter – NEVER **cin**!
   - **cin** may not exist!
   - May want to read from a file/buffer/socket. . .
2. Always return the stream received as parameter
   - To allow checking for input success.
   - To allow for chaining.
3. Return and all parameters should be references (non-**const**).
4. 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.
5. Always read what you print – always (so, keep the format simple!).
6. NEVER use **getline()** – you're corrupting the stream!
7. NEVER read into a **string** and parse that – stream corruption!
8. NEVER, EVER print anything!
9. Prefer constructors over setter member functions.
   Avoid setters altogether – not very OO. Same with getters. . .

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.
    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?"

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.

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.

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`