Laboratorio di Tecnologie dell'Informazione

Ing. Marco Bertini
bertini@dsi.unifi.it
http://www.dsi.unifi.it/~bertini/

Classes and objects

Why abstraction ?

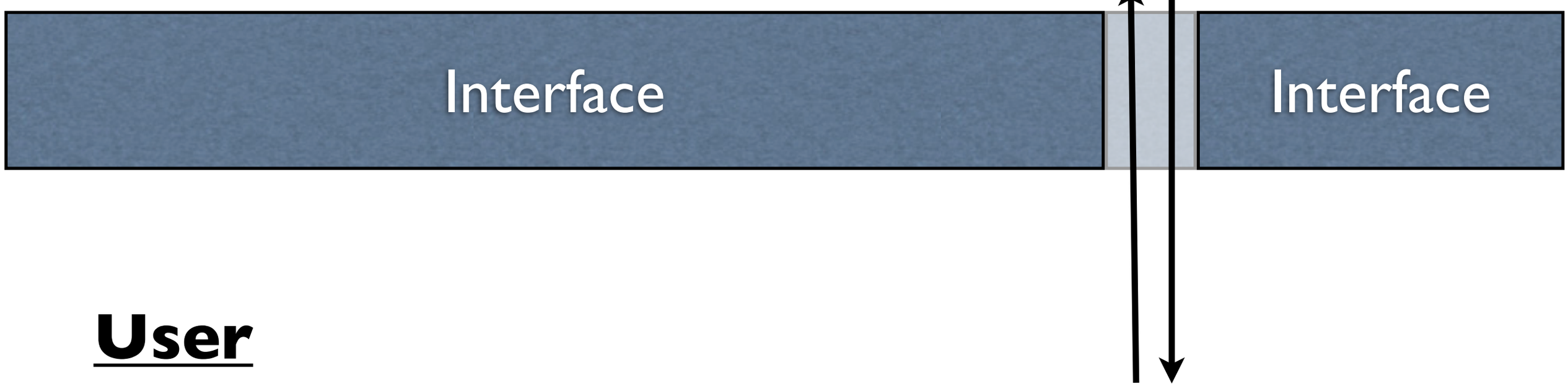
- Helps in modeling the problem, separating between necessary and unnecessary details
- We want to obtain a separation between:
  - operations performed on data
  - representation of data structures and algorithms implementation
- abstraction is the structuring of a nebulous problem into well-defined entities by defining their data and (coupled) operations.

ADT (Abstract Data Type)

- An ADT is a specification of a set of data and the set of operations (the ADT's interface) that can be performed on the data.
- It is abstract in the sense that it is independent of various concrete implementations.
- When realized in a computer program, the ADT is represented by an interface, which shields a corresponding implementation. Users of an ADT are concerned with the interface, but not the implementation, that can change in the future.

ADT (Abstract Data Type) - cont.

	dati	operazioni
Stack ADT	data=<d_1,d_2,...,d_n)	push(x)
	n	pop()
	Top-of-stack	top()



User

Why encapsulation ?

- The principle of hiding the used data structure and to only provide a well-defined interface is known as **encapsulation**.
- The separation of data structures and operations and the constraint to only access the data structure via a well-defined interface allows you to choose data structures appropriate for the application environment.

Why classes ?

- A class is an actual representation of an ADT: it provides implementation details for the data structure used and operations.
- Recall the important distinction between a class and an object:
  - A class is an abstract representation of a set of objects that behave identically.
  - Objects (i.e. variables) are instantiated from classes.
- classes define properties and behaviour of sets of objects.

Classes and objects

- A class is the implementation of an abstract data type (ADT). It defines attributes and methods which implement the data structure and operations of the ADT, respectively.

- An object is an instance of a class. It can be uniquely identified by its name and it defines a state which is represented by the values of its attributes at a particular time.

- The behaviour of an object is defined by the set of methods which can be applied on it.

Procedural programming

- There's a division between data and operations on data
- The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines
- When programming in C we focus on data structures and functions

OO programming

- In object-oriented programming the focus is to break down a programming task into objects and interactions between objects.

- An object is associated to data and operations on its data, e.g.:

  - an object "oven" has an internal data representing temperature and an operation that changes the temperature

Why C++ classes ?

- A C++ class can provide information hiding:
  - Hides the internal representation of data
  - Hides the implementation details of operations
- The class acts like a black box, providing a service to its clients, without opening up its code so that it can be used in the wrong way

Open-Closed Principle

- Encapsulation is a key technique in following the Open-Closed principle:
  - classes should be open for extension and closed for modification
- We want to allow changes to the system, but without requiring to modifying existing code

Open-Closed Principle

- Encapsulation is a key technique in following the Open-Closed principle:
  - clas
    clos
- We w
  withc

Open-Closed Principle:

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

- Bertrand Meyer

Open-Closed Principle - cont.

- if a class has a particular behaviour, coded the way we want, if nobody can change the class code we have **closed it for modification**.
- but if, for some reasons, we have to extend that behaviour we can let to extend the class to override the method and provide new functionality. The class is **open for extension**.
- We'll see how inheritance and composition will help us to follow this principle.

Why OCP ?

- Here's a small example (in C) that shows a case in which the code is not closed to modifications. We'll see how, using inheritance and abstractions, we can solve the problem.

Why OCP ?

```
enum ShapeType {circle, square};

struct Shape {
  ShapeType itsType;
};

struct Circle {
  ShapeType itsType;
  double itsRadius;
  Point itsCenter;
};

struct Square {
  ShapeType itsType;
  double itsSide;
  Point itsTopLeft;
};

// These functions are implemented elsewhere
void DrawSquare(struct Square*);
void DrawCircle(struct Circle*);

typedef struct Shape *ShapePointer;
```

```
void DrawAllShapes(ShapePointer list[], int n)
{
  int i;
  for (i=0; i<n; i++) {
    struct Shape* s = list[i];
    switch (s->itsType) {
      case square:
        DrawSquare((struct Square*)s);
        break;
      case circle:
        DrawCircle((struct Circle*)s);
        break;
    }
  }
}
```

Why OCP ?

```
enum ShapeType {circle, square};

struct Shape {
  ShapeType itsType;
};

s
}
s
}
// These functions are implemented elsewhere
void DrawSquare(struct Square*);
void DrawCircle(struct Circle*);

typedef struct Shape *ShapePointer;
```

It does not conform to the open-closed principle because it cannot be closed against new kinds of shapes.
If I wanted to extend this function to be able to draw a list of shapes that included triangles, I would have to modify the function.
In fact, I would have to modify the function for any new type of shape that I needed to draw.

```
void DrawAllShapes(ShapePointer list[], int n)
{
  int i;
  for (i=0; i<n; i++) {
    struct Shape* s = list[i];
    switch (s->itsType) {
      case square:
        DrawSquare((struct Square*)s);
        break;
      case circle:
        DrawCircle((struct Circle*)s);
        break;
    }
  }
}
```

Class identification

- Identify real world objects or entities as potential classes of software objects
- The usual approach is to think about the real world objects that exist in the application domain which is being programmed.
  Instead of thinking about what processing has to be done, as we so often do in procedural programming, we instead think about what things exist.

Class identification - cont.

- Identify groups of objects that behave similarly, that can be implemented as classes
- Classes are specifications for objects
- Delay decisions about implementation details, such as what data and operations will apply to objects, until we have a clear idea of what classes of object will be required

Class identification - cont.

- Begin class modeling by identifying candidate classes - an initial list of classes from which the actual design classes will emerge.
- A rule of the thumb to identify candidate classes: identify the noun and noun phrases, verbs (actions) and adjectives (attributes) from the use cases and problem description
- there are more formal methods to identify (e.g. CRC cards, use cases) and represent (e.g. UML) classes

Single Responsibility Principle

- Every object in the system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility
- A class should have only one reason to change
- A *responsibility* can be defined as a reason to change
- It's a concept related to cohesion

SRP & OCP

- Ideally, following the Open Closed Principle, means to to write a class or a method and then turn my back on it, comfortable that it does its job and I won't have to go back and change it.
- It's a a "laudable goal", but elusive in practice: you'll never reach true Open-Closed nirvana, but you can get close by following the related Single Responsibility Principle: a class should have one, and only one, reason to change.

SRP - cont

- As an example, consider a module that *compiles* and *prints* a report: the content of the report can *change*, the format of the report can *change*.

- The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules.
- Do not **couple** two things that change for different reasons at different times.

SRP: example

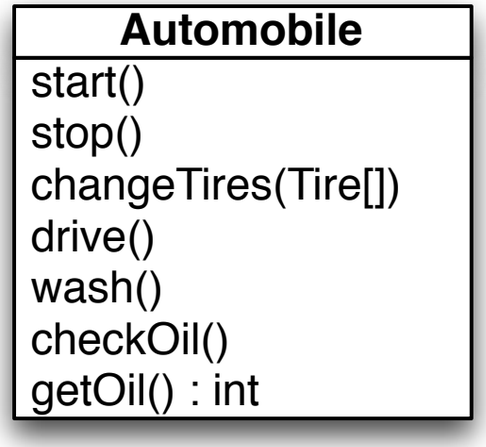
- Here's a simple test to check if a class follows SRP: for each method of the class write a line that says

  The *class name* *write method here* itself.

- Adjust grammar and syntax the read aloud each line. Does it make sense ?
- If it doesn't probably the method belongs to a different class. Use common sense !

SRP: example - cont.

- Apply the method to the Automobile class:



- We are still a bit far from having cars driving themselves (we may need a Driver)
- Surely they won't change their tires or wash themselves (Mechanic and CarWash may help...)
- Think very well about the meaning of the methods: getOil may simply mean that the car has a sensor

C++ Classes

- A C++ class extends the concept of C structs
- It collects together a group of variables (attributes or data members) that can be referenced to using a collective name and a symbolic identifier
- It can have functions (methods or function members) that operate within the context of the class
- It defines a data type: we can create instances (objects)

Class definition

- Use the keyword class, e.g.:

```
class Stack {
bool push(data value);
bool pop(data* pValue);
void init(int size);

int TOS;
data* buffer;
int size;
}; // do NOT forget the ; !
```

A C stack implementation

```
struct stack {
   int TOS;
   data *buffer;
   int size;
};
bool push(struct stack *ptr, data value) {
...
}
bool pop(struct stack *ptr, data *pValue) {
...
}
```

Access level

- All the members of a `struct` are visible as soon as there's a reference to the structure, while in a `class` it is possible to differentiate the access as `public`, `private` and `protected`. The default class access is `private`.
- We can design better the "interface" of the class, i.e. decide what can is hidden and what is visible in the class (encapsulation). We can decouple classes.

Access levels

- `public`: a public member is visible to anyone who has an address or a reference to the object
- `private`: a private member is visible only to the methods of the class in which it is defined
- `protected`: a protected member is visible only to the methods of the class in which it is defined, and in the derived classes (through and inheritance mechanism)

Access levels: example

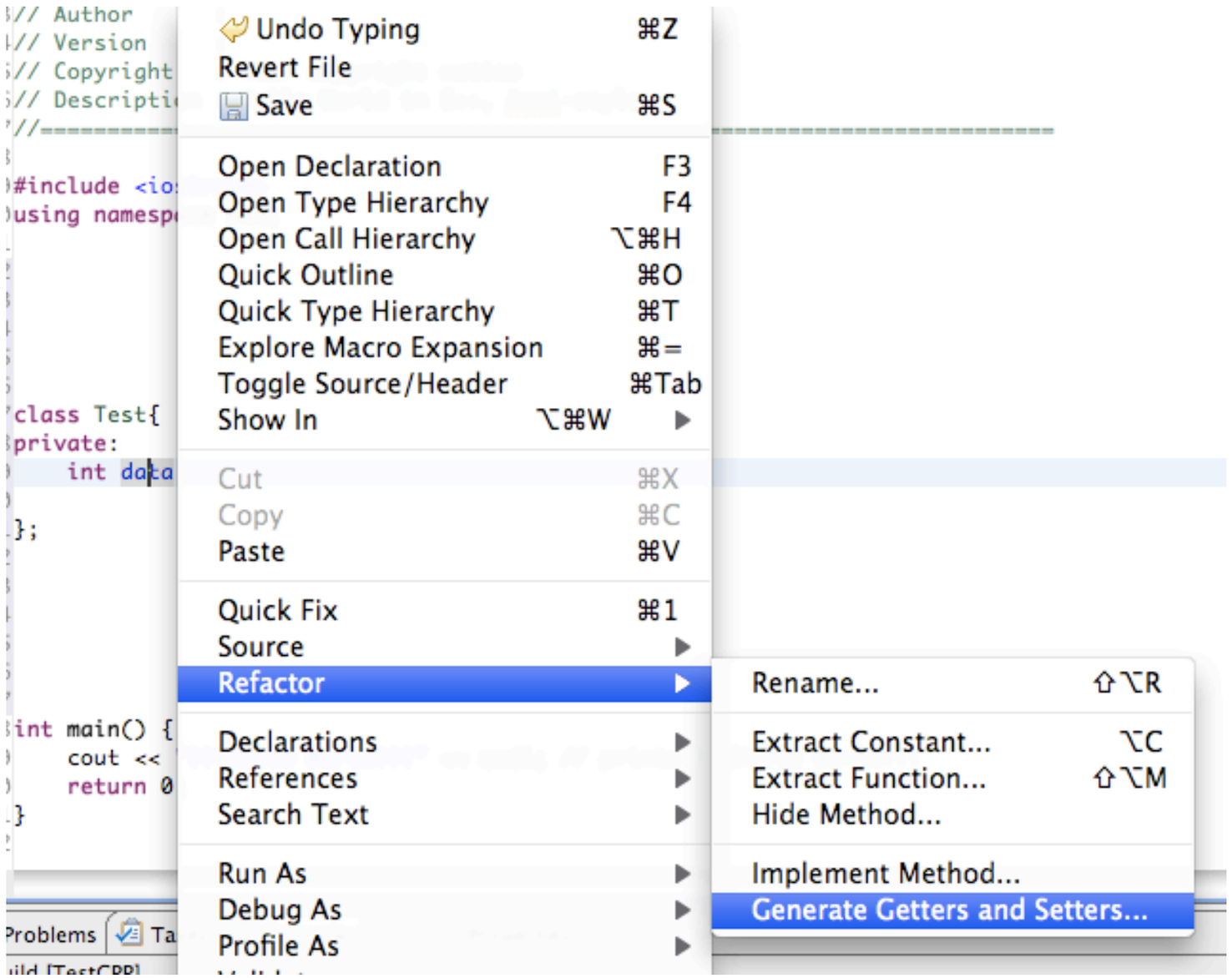
```
class Stack {
public:
        bool push(data value);
        bool pop(data* pValue);
        void init(int size);

private:
        int TOS;
        data* buffer;
        int size;
};
bool Stack::push(data value) {};
```

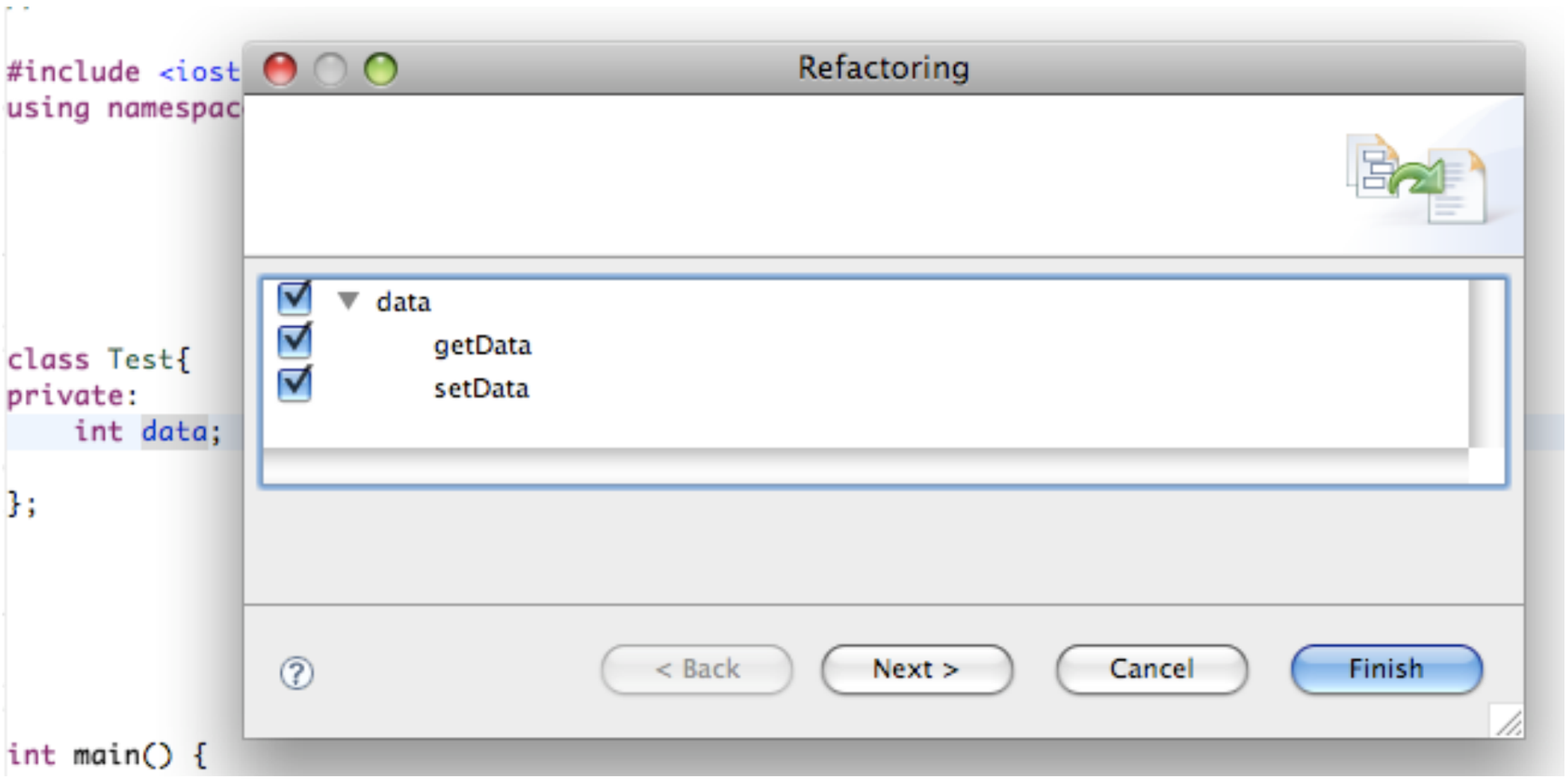
Access levels: rules of the thumb

- Always use explicit access control
- Do not have public data members
  - use public methods to set/get their values
    - many IDEs can create these methods automatically

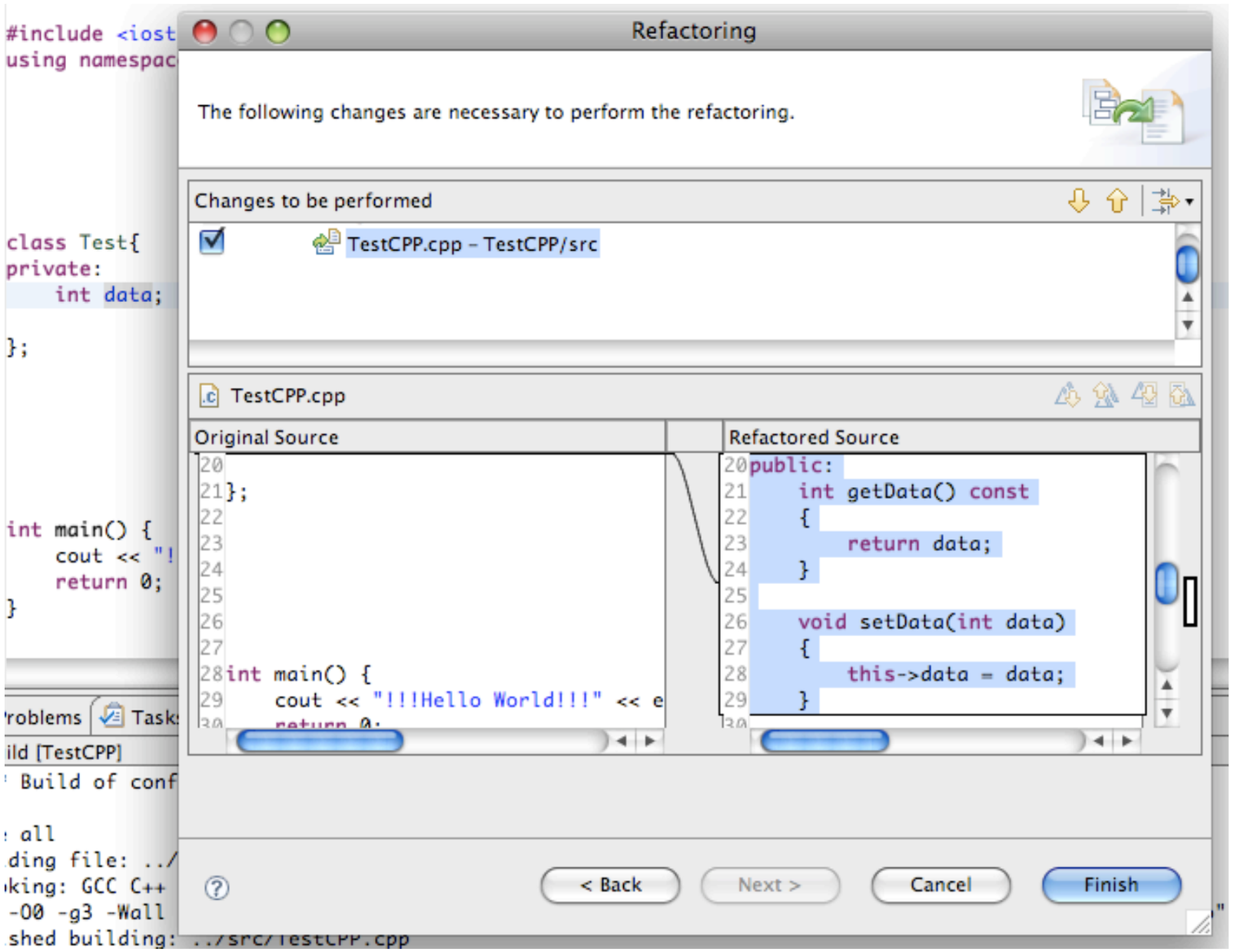
Getter/setter creation in Eclipse



Getter/setter creation in Eclipse



Getter/setter creation in Eclipse



Getter/setter creation in Eclipse

```
class Test{
private:
    int data;
public:
    int getData() const
    {
        return data;
    }

    void setData(int data)
    {
        this->data = data;
    }

};
```

Method implementation

- The methods are usually defined (implemented) in the .cpp files: add the class name in front of the method, e.g.:

```
bool stack::push(data value) {
// code to implement the method
}
```

- We can implement them also in the header (inline), but usually this is done only if they are very short (e.g. ~5-7 lines)

Attributes

- A method may access the attributes of the class: the attributes are visible within the methods
  - this greatly reduces the complexity of C "interfaces": compare the C++ implementation with a C implementation
- The attributes maintain the "state" of the object

Attributes - cont.

- The attributes are a sort of a context shared by the methods (that's why interfaces are simpler).
- However, the methods are more coupled with the attributes.
- It's well worth to pay for this price, if the classes have been designed to have cohesive responsibilities*

*a responsibility is something that a class knows or does.

Argument passing

- In C the argument passing mechanism is "pass by value": the value of run-time arguments are copied in the formal parameters
  - a function uses the copy of the values to carry out its computation
  - we have to use pointers to simulate a "pass by reference"
- In C++ we can pass parameters by reference

Pass by reference

- A reference is essentially a synonym (alias) in the sense that there is no copying of the data passed as the actual argument.
- It is indicated by the ampersand (&) characters following the argument base type.
- C++ call by reference and C-style simulated call by reference using pointers are similar, but there are no explicit pointers involved: no need to dereference the argument.

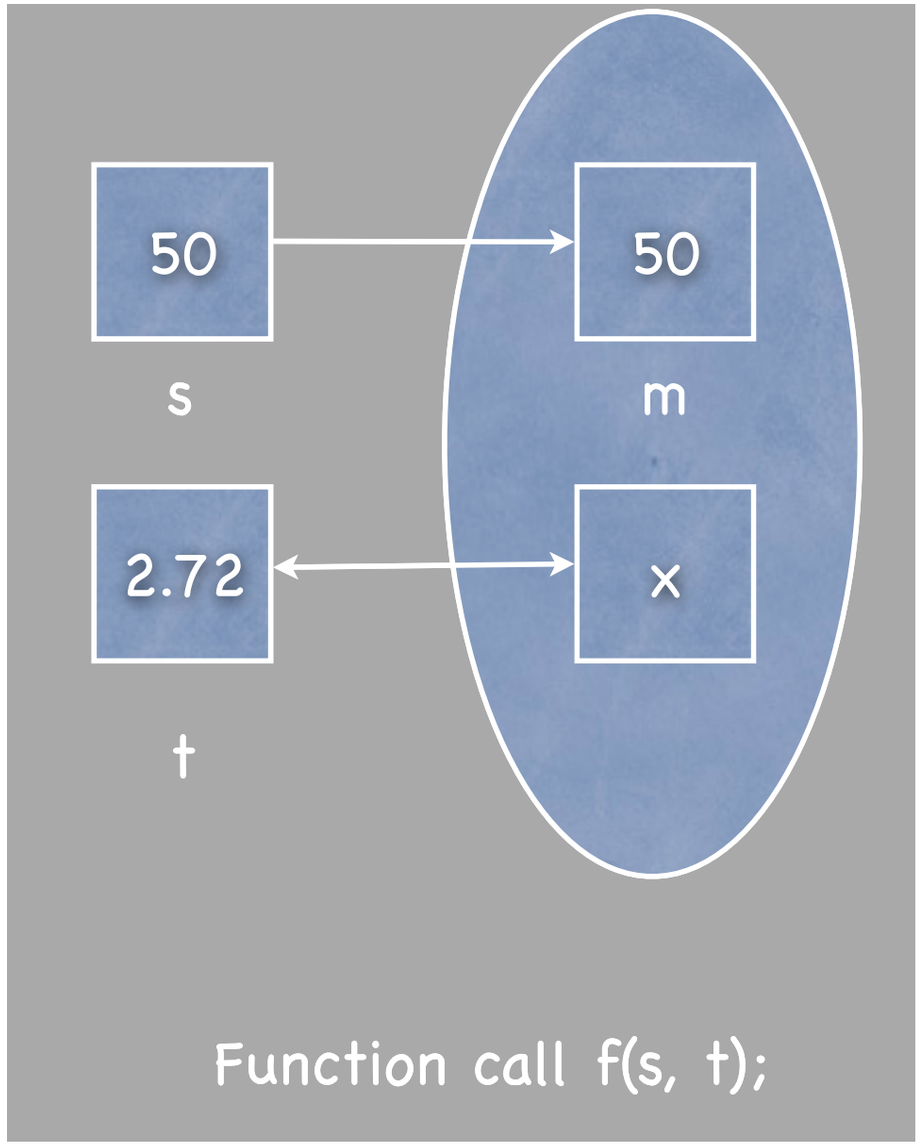
Pass by reference - cont.

```
void add(int a, int b, int& sum) {
  sum = a + b;
}

int main() {
  int i = 1;
  int j = 2;
  int total = 0;
  cout << “total: “ << total << endl;
  add( i, j, total);
  cout << “total: “ << total << endl;
}
```

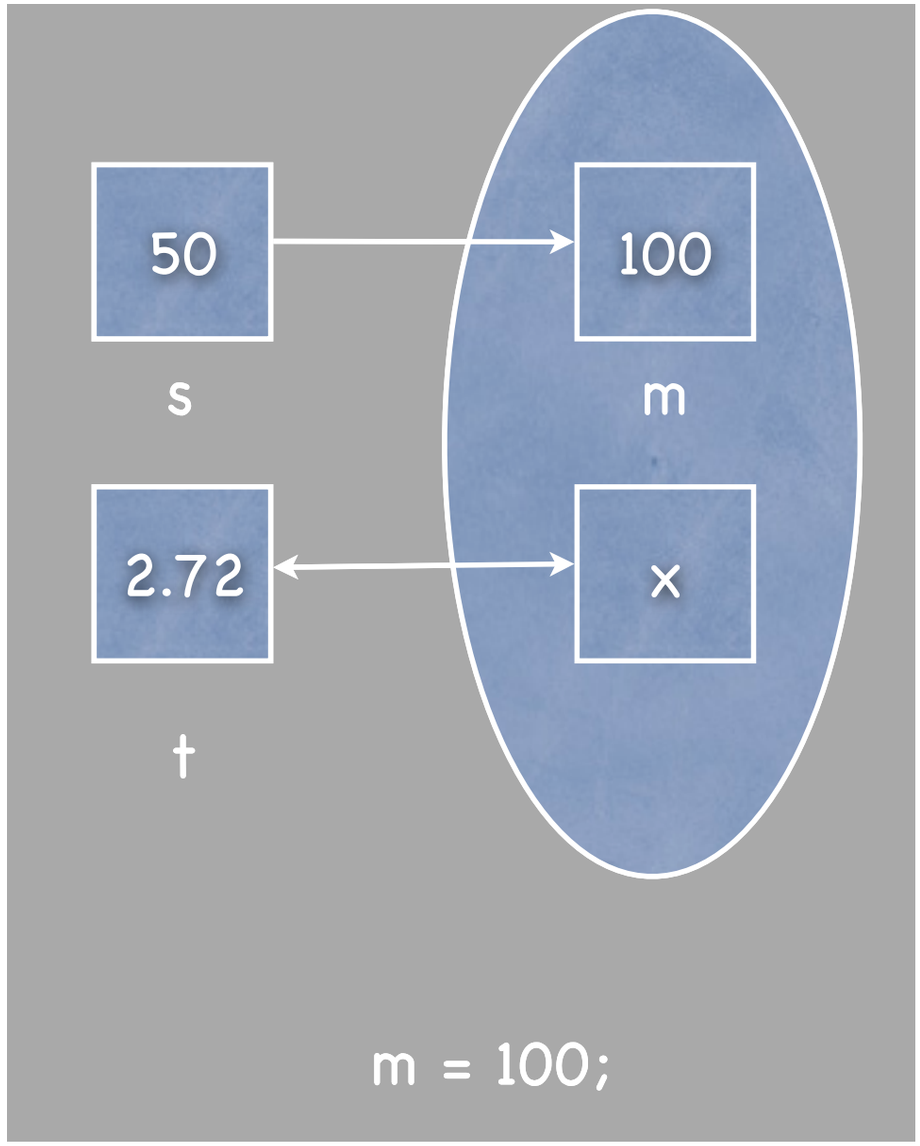
Pass by reference - cont.

```
void f(int m, double& x) {
  m = 100;
  x = 3.14
}
int main() {
  int s = 50;
  double t = 2.72;
  f(s,t);
  return 0;
}
```



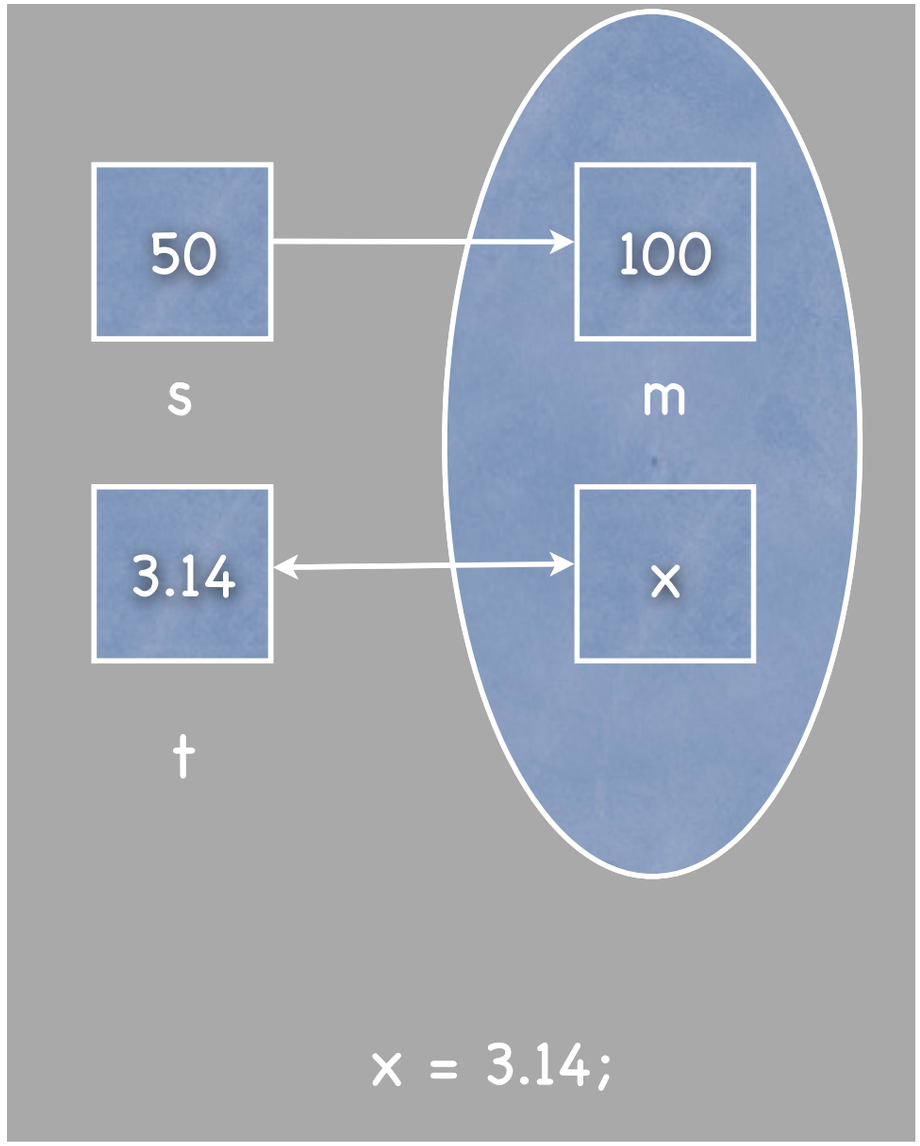
Pass by reference - cont.

```
void f(int m, double& x) {
  m = 100;
  x = 3.14
}
int main() {
  int s = 50;
  double t = 2.72;
  f(s,t);
  return 0;
}
```



Pass by reference - cont.

```
void f(int m, double& x) {
  m = 100;
  x = 3.14
}
int main() {
  int s = 50;
  double t = 2.72;
  f(s,t);
  return 0;
}
```



Pass by reference - cont.

- A reference can be specified as `const`: the function/method can not modify the content of the variable
  - pass large data structures that should not be modified as const references (it's fast)

```
42 void add(int a, int b, const int& sum) {
43     sum = a + b;
44 }
45
```

Problems | Tasks | Console | Properties

C-Build [TestRef]

```
**** Build of configuration Debug for project TestRef ****

make all
Building file: ../TestRef.cpp
Invoking: GCC C++ Compiler
g++ -O0 -g3 -Wall -c -fmessage-length=0 -MMD -MP -MF"TestRef.d" -MT"Te
../TestRef.cpp: In function 'void add(int, int, const int&)':
../TestRef.cpp:43: error: assignment of read-only reference 'sum'
make: *** [TestRef.o] Error 1
```

Reference variables

- It is possible to have a reference variable, but it must always hold a valid reference and so must be initialised when it is created.

```
int x;
int& y=x; // reference
y=2; // also x is modified
int& z; // Error: doesn't compile ! Why ?
int *z; // pointer
z = &x; // & on the left is different from & on the right of =
*z = 3; // x is modified
```

Overloading

- We can define more than one methods with the same name and return type, but with different (number and types) parameters (signature). The method is said to be overloaded.
- Commonly used to provide alternate versions of functions to be used in different situations
  - the same abstract operation may have different concrete implementations

Overloading - cont.

- The compiler will create a different code segment and symbol (through name mangling), obtained extending the method name with suffixes related to the types of the parameters

```
class Test{
private:
    int data;
public:
    int getData() const
    {
        return data;
    }

    void setData(int data)
    {
        this->data = data;
    }
    void setData(int data, float mul)
    {
        this->data = data*mul;
    }
};
```

```
Terminal — b
nirvana:Debug bertini$ nm TestCPP
0000200c D _NXArgc
00002008 D _NXArgv
00001ec4 s __GLOBAL__I_main
00001e6c s __Z41__static_initialization_and
00001e2a T __ZN4Test7setDataEi
00001e3a T __ZN4Test7setDataEif
00001e1c T __ZNK4Test7getDataEv
         U __ZNSolsEPFRSoS_E
         U __ZNSt8ios_base4InitC1Ev
```

- we can not just change the return value: the compiler should always check the type of the variable where we put the value... and what if we discard it ?

Operator overloading

- It is possible to overload also operators, not only methods (in real life: + is an operator used for integers, real numbers, complex numbers...)
- Overload operators when it really makes sense
- e.g. overload == to compare strings, do not overload * for strings...
- Some OO languages do not allow operator overloading, most notably Java*

*sort of... String + is operator overload

Operator overloading - cont.

- Operators are overloaded so that the objects behave as primitive types. New operators cannot be created, only the functionality of existing operators on objects can be modified
- If you overload + do NOT expect that += is given automatically ! Define also that operator !
- Often operators are just friends... (more later)

Operator overloading - cont.

```
class Array {
public:
  Array(int size); // constructor
  bool operator ==(const Array& right) const; //the method can't modify
anything
  // ... other members
private:
  int size;
  int* data;  // pointer to first element of array
};
bool Array::operator==(const Array& right) const {
  if ( size != right.size )  // start checking the size of the arrays
    return false;
  // then check the whole content of arrays
  for ( int i=0; i < size; i++ ) {
    if ( data[i] != right.data[i] )
      return false;
  }
  return true;  // both size and content are equal
}
```

Type checking

- C++ has a stricter type checking than C: depending on the parameter cast you determine the method that is executed !

- E.g.: cast void pointers when assigning them to other pointers (in C it compiles)

```
29      int aInt = 3;
30      void* ptr = &aInt;
31      int* pInt;
32      pInt = ptr;
33      pInt = (int*)ptr;
34
```

Object creation

- Once a class is defined we can create the instances (objects) from it, as it is done for the variables of base types.
- Creation can be static (on the stack) or dynamic (on the heap)
- The code of the methods is represented in the code segment, shared between all the instances of a class
  - each object has the address of the function that implements the methods

Dynamic object creation

- It is similar to the use of `malloc/free` functions, but syntactically simplified, using `new` and `delete`:

```
Stack* sPtr;

sPtr = new Stack;

...

delete sPtr;
```

Constructors

- A member function that will be invoked when an object of that class is created. Returns no values.
  - Always has the same name as the class. Constructors generally perform some kind of initialisation on a new object. If not constructor is defined a default one is created, with no parameters .
  - Common to overload a constructor function (i.e. provide several versions) so the object can be created in a number of different ways.
- Consider how objects of a new type may be created and what constructors are needed.

Constructors - cont.

- If no constructor is defined the compile generates a "default" constructor that takes no parameters
- The default constructor is invoked (usually) without parentheses, e.g. in the previous example:

```
sPtr = new stack;
```

Constructors - cont.

- If a class has any constructors but no default constructor, its creation will be constrained to situations handled by the constructors, e.g.

```
class B {
public:
      B(int i) { ... }
};
B b1; // illegal
B b3(123); // ok
```

Constructors - cont.

- There's a compact and compiler-friendly way to init attributes in a constructor:

```
class Stack {
protected:
      int TOS;
      data *buffer;
      int size;
public:
      Stack(int s) : TOS(0), size(s), buffer(new data[s])
      {...};
      }
```

Constructors - cont.

- Constructors are `public` (usually, but not necessarily)
- If we do not want that a class is instantiated we can declare a constructor as `protected`. We can instantiate derived classes (if their constructor is `public`).
- In other cases we can declare a constructor as `private`.
    - typically its use is related to `static` methods

Explicit constructors

- C++ constructors that have just one parameter automatically perform implicit type conversion, e.g.:
  if you pass an `int` when the constructor expects a `string` pointer parameter, the compiler will add the code it must have to convert the int to a string pointer.
- You can add `explicit` to the constructor declaration to prevent these implicit conversions.

Explicit constructors - cont.

- Declaring a constructor that has multiple arguments to be explicit has no effect, because such constructors cannot take part in implicit conversions.
- However, explicit will have an effect if a constructor has multiple arguments and all except one of the arguments has a default value.

Explicit constructors: example

```
class A {
public:
 A();
};

class B {
public:
 explicit B(int x=0, bool b=true);
};

class C {
public:
 explicit C(int x);
};


void doSomething(B objB);
```

```
B objB1;

doSomething( bObj1 ); // OK

B objB2( 28 ); // OK, b arg is set
to default

doSomething(28); // BAD: we need a
B obj, and we do not allow implicit
conversion

doSomething(B(28)); // OK

doSomething(“foo”); // BAD, thanks
the compiler for not allowing it
```

Explicit constructors - cont.

- It's preferable to use explicit constructor (there is even a Google C++ guideline for it)
- When designing a type (i.e. class) think about what conversions should be allowed: should you write a type conversion function or a non explicit constructor (with a single argument) ?

Destructors

- It's a method with the name of he class preceded by ~, e.g.: ~Stack();
- No parameters, no return values, no overload
- Called automatically when an object is destroyed
  - should perform housekeeping

C'tor and D'tor

```
class Stack {
public:
  Stack(int s);
  ~Stack();
  //..
protected:
  int _TOS;
  data* _buffer;
  int _size;

}
```

```
// C'tor allocates memory
Stack::Stack(int s)
{
  _TOS=0;
  _size=s;
  _buffer = new data[size];
}

// D'tor has to release memory
Stack::~Stack()
{
  delete(_buffer);
}
```

How to use methods and attributes ?

- Class members can be referenced to analogously to struct members:

```
<var>.member_name
<expr_addr>->member_name
```

but taking into account their visibility, defined by the access level, e.g.

How to use methods and attributes: example

```
Stack S;
Stack* pS;
...
pS = &S;
...
S.push(3);
...
pS->push(8);
```

Self reference

- An object can refer to itself using the keyword `this`
- An object implicitly uses this when it refers to a method or attribute

```
Stack::Stack(int s)
{
  _TOS=0;
  _size=s;
  _buffer = new data[_size];
}
```

```
Stack::Stack(int s)
{
  this->_TOS=0;
  this->_size=s;
  this->_buffer = new data[this->_size];
}
```

Self reference - cont.

- The use of `this` is essential when an object has to pass a reference of itself to another object
- A typical application is the callback: obj A gives a reference to itself that will be used by obj B to invoke a method on obj A
- This is used to implement inversion of responsibility schemas:
  obj A does not call obj B to perform an operation but lets obj B call obj A

Self reference - example

```
class Observer;
class Subject;
class Observer {
public:
  void update(subject*
pSubj);
  int getState() {
    return state;
  }
private:
  int state;
};

class Subject {
public:
  Subject(Observer* pObs);
  void setState(int aState);
  int getState() {
    return state;
  }
private:
  int state;
  Observer* pObs;
}
```

```
Observer::update(subject* pSubj) {
  if ( ... )  // possible condition that
              // starts an update
    this->state = pSubj->getState();
}
Subject::Subject(Observer* pObs) {
  this->pObs = pObs;
}
Subject::setState(int aState) {
  this->state  = aState;
  this->pObs->update( this );
}


int main() {
  Subject* pSubj;
  Observer* pObs;
  pObs = new Observer;
  pSubj = new Subject( pObs );
  // ...
  pSubj->setState( 10 );
  cout << “subj state: “ << pSubj->getState << endl;
  cout << “obs state: “ << pObs->getState() << endl;
}
```

Static members

- A `static` member is associated with the class, not with object (instance of the class), i.e. there's only one copy of the member for all the instances
  - extends the static variables of C
- Static data member: one copy of the variable
- Static function member: can be invoked without requiring an object

Static data members

```
class Point {
public:
  Point() {
    x=y=0;
    n++;
  }
  ~Point() {
    n--;
  }
  int count() const {
    return n;
  }
  // ...
private:
 int x,y;
 static int n; // declaration
}
```

```
// definition: must be in
// namespace scope
int Point::n = 0;

int main() {
  Point a,b;
  cout << “n: “ << p.count()
<< endl;
}
```

Static method members

```
class Point {
public:
  Point() {
    x=y=0;
    n++;
  }
  ~Point() {
    n--;
  }
  static int n;
  static float distance(const
Point a, const Point b) {
    //...calc distance
  }
  // ...
private:
 int x,y;
}
```

```
// definition: must be in
// namespace scope
int Point::n = 0;

int main() {
  // access static members even before
  // the creation of any instance of
  // the class
  cout << “n: “  << Point::n << endl;
  Point a,b;
  // set a and b coordinates
  Point::distance(a,b);
}
```

Friend

- A class can allow access to its members (even if private) declaring that top-level functions (or even classes) are its friends

- Friends should only used in very special situations, e.g. I/O operator overloads where it is not desirable to provide accessor member functions.

  - It hinders encapsulation

Friend - cont.

```
class Point{
private:
  int x,y;
public:
  friend bool operator==(Point a,
Point b);
  Point() : x(0), y(0) {};
  //...
}

bool operator==(Point a, Point b) {
  if ( ( a.x != b.x ) ||
       ( a.y != b.y ) )
    return false;
  else
    return true;
}
```

```
int main() {
  Point p, q;
  //...
  if (p == q)
    std::cout << "p and q are equal"
<< endl;
   return 0;
}
```

Inner class

- A inner class or nested class is a class declared entirely within the body of another class or interface. An instance of an inner class cannot be instantiated without being bound to a top-level class.
- Inner classes allow for the object orientation of certain parts of the program that would otherwise not be encapsulated into a class.

Inner class - cont.

- C++ nested classes are in the scope of their enclosing classes.
- Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class, without qualifying the name
(other classes that are not one of its enclosing classes have to qualify its name with its enclosing class's name).

Inner class

- Let a nested class to access the non-static members of the including class using `friend`

```
class Outer {
    string name;

    // Define a inner class:
    class Inner1;
    friend class Outer::Inner1;
    class Inner1 {
        Outer* parent;
        public:
            Inner1(Outer* p) : parent(p) {}
            void foo() {
                // Accesses data in the
                // outer class object
                cout<< parent->name<< endl;
            }
} inner1;
```

```
    // Define a second inner class:
    class Inner2;
    friend class Outer::Inner2;
    class Inner2 {
        Outer* parent;
        public:
            Inner2(Outer* p) : parent(p) {}
            void bar() {
                cout<< parent->name<< endl;
            }
    } inner2;

    public:
        Outer(const string& nm)
            : name(nm), inner1(this),
              inner2(this) {}
}; // Outer
```

Credits

- These slides are (heavily) based on the material of:
  - Dr. Ian Richards, CSC2402, Univ. of Southern Queensland
  - Prof. Paolo Frasconi, IIN 167, Univ. di Firenze
  - "Head first: Object Oriented Analysis and Design", O'Reilly