Encapsulation

- constructors
- destructors
- private vs public
- overloads
- friend
- this

Strings

Encapsulation

- It is the first paradigm of Object Oriented Programming (OOP)
- We bundle together data structures and functions which operate on them
- The functions are defined together with the data structure and are called respectively methods and members

Example - Encapsulation

```
struct Rectangle
{
   int m_width, m_height;                  // variables (note the m_)
   void initialize(int,int);               // method: declared but not defined
   int area() {return m_width*m_height;}   // method: defined inside the class
};

// method definition outside the class(note the ::)
void Rectangle::initialize(int x, int y
{
    m_width = x;
    m_height = y;
}

int main ()
{
   Rectangle rect;
   rect.initialize(3,4);    // method call: variableName.methodName(parameters)
   cout << "area: " << rect.area() << endl;
   return 0;
}
```

Encapsulation

- The variables which are part of the object (e.g. m_width) are called **data members**
- They can have any name, like variables. Prefixing with **m_** is just my naming convention to make the code more readable
- **Methods** can be defined within the class or later, using the scope operator **::**
- If defined later, definition could be in a separate file (struct in H file, definition in CPP file)
- Methods can be accessed with the **.** operator

Problem

- Modify *InputNumberArrayFun2.cpp* and define an object which acts as a dynamic vector
- Let's have a look . . . the data was:
  - `unsigned int capacity;   // storage capacity`
  - `unsigned int size;        // how many numbers have been stored so far`
  - `unsigned int *numbers; // the memory`
- and the operations related to the vector are
  - `initialize, grow capacity, add element at the end, print, release memory`
- Full solution in *InputNumberArrayStruct1.cpp*

Constructors

- Constructors are special functions with no return type which are automatically called when an object is created
- After the object has been constructed, they are no longer used
- The function name must be the same as the struct name
- Constructors can be overloaded, i.e. there can be multiple constructors, with different parameter lists

Constructors

- **The syntax is**

```
StructName(parameters)
  : commaSeparatedMemberInitializers
{
  /* body */
}
```

- StructName must be the name of the struct
- The comma separated member initializer list allows to initialize data members
  - Can initialize only some data members or none
  - Data members must appear in the same order in which they are defined

Example - Constructor

```
struct Rectangle
{
   int m_width, m_height;                  // variables (note the m_)
   Rectangle(int x,int y) // constructor (note there is no return type)
      : m_width(x)     // initialize member m_width
      , m_height(y)    // initialize member m_height
   {
      // here we could do something else
   }
   int area() {return m_width*m_height;}  // method: defined here
};

int main ()
{
   Rectangle rect(3,4);  // constructor called at object creation
   cout << "area: " << rect.area() << endl;
   return 0;
}
```

Constructor – What's wrong?

```
struct Rectangle
{
   int m_width, m_height;
   Rectangle(int x,int y)
      : m_height(x)
      , m_width(y)
   {
      // here we could do something else
   }
};
```

- Height and width are not initialized in the same order in which they are declared

Constructor – What's wrong?

```
struct Rectangle
{
   int m_width, m_height;
   void Rectangle(int x, int y)
      : m_width(x)     // initialize member m_width
      , m_height(y)    // initialize member m_height
   {
      // here we could do something else
   }
};
```

- Constructors cannot have a return type

Constructor – What's wrong?

```
struct Rectangle
{
   int m_width, m_height;
   init(int x, int y)
      : m_width(x)     // initialize member m_width
      , m_height(y)    // initialize member m_height
      {
         // here we could do something else
      }
};
```

- A constructor must have the same name as the struct

What are initializer lists for?

- They are used to call the constructors of data members passing them arguments. These cannot be called from the function body.

```
struct A
{
    int m_x;
    A() : m_x(0) {}             // constructor with no arguments
    A(int x) : m_x(x) {}        // constructor with one argument
};

struct B
{
    A m_a;
    B(int x) : m_a(x)  // from the initializer list we can call the constructor of A
    { // inside the body we cannot use constructors of A }
};

int main()
{
    A a(1);
    B b(1);
    return 0;
}
```

Can we omit the initializer list?

- Yes, but the constructor with no arguments (default constructor) will be called anyway

```
struct A
{
    int m_x;
    A() : m_x(3) {}
    A(int x) : m_x(x) {}
};

struct B
{
    A m_a;
    B(int x) // no initializer list, hence the default constructor of A is called, which sets m_x=3
    {
        m_a.m_x = x;  // inside the body cannot use constructors of A, so I have to modify m_x directly
    };

int main()
{
    A a(1);
    B b(1);
    return 0;
}
```

Copy and Default Constructor

- We can define multiple overload for constructors, with different parameters list
- There are two special constructors which, if not explicitly implemented, are added implicitly by the compiler
  - default constructor: has an empty parameter list. Implicitly added by the compiler, but only if no other constructor is explicitly defined
  - copy constructor: has just one parameter, a const reference of the same type as the object itself

Implicit Copy Constructor

- Copies every data member, one by one, in the same order in which they are defined
- If any data member has its own copy constructor, it is invoked passing as argument the data member of the copied object with the same name

Example - Explicit Copy Constructor

```
struct Position
{
    double m_x, m_y;
    Position(const Position& p)  // copy constructor
        : m_x(p.m_x)    // copies p.m_x to m_x
        , m_y(p.m_y)    // copies p.m_y to m_y
    {
    }
    Position(double x, double y)
        : m_x(x)
        , m_y(y)
    {
    }
};
```

- Here we define two overloads for the constructor. The first takes as an argument a const reference to an object of the same type: this is called **copy constructor**
- In this case my specific implementation of the copy constructor is redundant, as it does the same thing the compiler would have done automatically. Normally we implement the copy constructor only if we want to do something different from the default copy behaviour.

Implicit Default Constructor

- Takes no arguments
- Calls the default constructor of every data members in the order in which they are declared
- Data members which do not have a default constructor will not be initialized
- Implicitly defined by the compiler only if no other constructor is explicitly defined

Example – Explicit Default Constructor

```
struct Position
{
 double m_x, m_y;
   Position()  // default constructor
      : m_x(0)   // set m_x to zero
      , m_y(0)   // set m_y to zero
   {
   }
   Position(double x, double y)
      : m_x(x)
      , m_y(y)
   {
   }
};
```

- Here we define 3 overloads for the constructor.
- The copy constructor is automatically defined by the compiler
- The default constructor takes no argument initialize m_x and m_y
- Note: the automatic default constructor would have not initialized m_x and m_y, because they have no default constructors, so it is a good thing we defined it explicitly

Non-constructed types

- Same types (e.g. fundamental types like int, double, char, bool, float, ...) have no constructors
- So, if we type `double x;` x is not automatically initialized to anything

Constructor Example

```
void print(const char *name, const Position& p)
{
    cout << name << "(" << p.m_x << "," << p.m_y << ")" << endl;
}

int main()
{
   Position x;        // call default constructor
   Position t();      // call default constructor
   Position y(1,2);   // call constructor Position(int,int)
   Position z(y);     // call copy constructor

   print("x", x);
   print("t", t);
   print("y", y);
   print("z", z);
}
```

Abstraction from Implementation Details

- One of the goals of encapsulation is to segregate the functionality of the class from its internal implementation
- The functionality is defined by the methods exposed to the user
- The internal implementation is hidden to the user and defined by the body of these functions and the data structure
- Advantage: If we change the internal implementation of an object, we do not need to modify the code which uses it, as long as the functionality does not change

Public vs Private

- We define as **public** methods or members which are supposed to be accessed by the user
- We define as **private** methods or members which are part of the implementation detail and can be used only inside the struct
- In a struct everything is public by default
- The keywords public and private change access rights of all methods and members which follow
- The keywords public and private can be used multiple times within a struct

Example – Private vs Public

```
struct Position
{
    // anything defined here is public
private:
    // members m_x and m_y are declared as private
    // they can only by used inside the methods of this struct
    double m_x, m_y;

public:
    // the constructors is defined as public
    // and can be accessed outside of this struct
    Position() : m_x(0), m_y(0) {}  // m_x, m_y can be used from inside the struct
    Position(double x, double y) : m_x(x), m_y(y) {}
};

int main()
{
   Position x(1,2);       // this is ok: the constructor can be used from here
   cout << x.m_x << endl; // this is not ok: m_x is private and cannot be used here
}
```

struct vs class

- A **class** is conceptually the same thing as a struct
- class and struct have different private vs public rules
- In a class everything is private by default, in a struct everything is public by default
- There are also other differences, still related to public vs private, but related to **inheritance** (we will discuss these later when we introduce inheritance)

Example – class

```
class Position
{
    // anything defined here is private
    // members m_x and m_y are declared as private
    // they can only by used inside the methods of this struct
   double m_x, m_y;

public:
    // the constructors is defined as public
    // and can be accessed outside of this struct
    Position() : m_x(0), m_y(0) {}
    Position(double x, double y) : m_x(x), m_y(y) {}
};
```

Accessing Private Members

- Now that the coordinate of the Position are private, the function *print* we wrote does not work anymore, because it has not access to *m_x* and *m_y*
- We need to introduce **accessor** methods

Example

```
class Position
{
   double m_x, m_y;

public:
    Position() : m_x(0), m_y(0) {}  // m_x, m_y can be used from inside the struct
    Position(double x, double y) : m_x(x), m_y(y) {}

    double getX() { return m_x; }  // read only public accessor
    double getY() { return m_y; }  // read only public accessor

    void set(double x, double y) { m_x = x; m_y = y; }  // set new coordinates
};

void print(const char *name, Position& p)
{
    // getX and getY are public and const methods, so they can be accessed from here
    cout << name << "(" << p.getX() << "," << p.getY() << ")" << endl;
}
```

- We introduce a new methods to manipulate private data (set) and read only accessor methods, to retrieve the current coordinates

What's wrong?

```
class Position
{
   double m_x, m_y;

public:
    Position(int x, int y) : m_x(x), m_y(y) {}
private:
    double getX() { return m_x; }  // read only public accessor
    double getY() { return m_y; }  // read only public accessor
};

void print(const char *name, Position& p)
{
    // getX and getY are private and cannot be used from here from here
    cout << name << "(" << p.getX() << "," << p.getY() << ")" << endl; // compile error
}
```

- The methods getX and getY are private, therefore they cannot be used by the function print

Encapsulation – Benefits summary

- bundle together data and functions operating on them, thus making the code easier to read and better organized
- reduce collisions of identically named variables
- abstract the user of an object from its implementation details, facilitating maintenance.

Benefits of Abstraction

- To understand the benefits of abstraction, let's change the internal representation of the position class
- Instead of using the coordinates x and y, let's use radius and angle
- A user of the object, which uses the public methods getX and getY, does not need to know that the internal representation is changing
- Because m_x and m_y were declared private, we can be sure that no code outside the class is referring to them directly

Example

```
class Position
{
    double m_radius;
    double m_phase;

public:
    Position() : m_radius(0), m_phase(0) {}
    Position(double x, double y)
        : m_radius(sqrt(x*x+y*y))
          , m_phase(x == 0 ? (y>=0? M_PI/2: -M_PI/2) : atan(y/x))
    {
    }

    double getX() { return m_radius*cos(m_phase); }
    double getY() { return m_radius*sin(m_phase); }
};
```

- The internal implementation and data representation has changed completely, but this has no impact on an external user, as the head of the public methods (constructors, getX and getY) has not changed

const methods

- If we mark an object as const, we do not want its content to be modified
- In general methods can modify the object: we want that on a const object only methods which do not modify it can be called
- We can identify methods which do not modify the object by tagging them with the keyword **const** at the end of the function head

```
struct A {
    returnType foo(argumentList) const; // foo is not allowed to modify any data member
};
```

Example

```
class Position
{
   double m_x, m_y;

public:
    Position() : m_x(0), m_y(0) {}  // m_x, m_y can be used from inside the struct
    Position(double x, double y) : m_x(x), m_y(y) {}

    double getX() const { return m_x; }  // tagged as const
    double getY() const { return m_y; }  // tagged as const

    void set(double x, double y) { m_x = x; m_y = y; }  // set new coordinates
};

void print(const char *name, const Position& p)
{
    // getX and getY are public and const methods, so they can be accessed from here
    cout << name << "(" << p.getX() << "," << p.getY() << ")" << endl;
}
```

- The *print* function receives the object p as const, hence it is allowed to call only its const methods, e.g. it can call *getX*, but not *set*, because *set* is not marked as a const method

What's wrong?

```
class Position
{
    double m_x, m_y;

public:
    Position(int x, int y) : m_x(x), m_y(y) {}

    double getX() { return m_x; }  // read only public accessor
    double getY() { return m_y; }  // read only public accessor
    void set(double x, double y) const { m_x = x; m_y = y; }  // set new coordinates
};

void print(const char *name, const Position& p)
{
    // getX and getY are public but non const methods, while p is declared as const
    // so they cannot be accessed from here
    cout << name << "(" << p.getX() << "," << p.getY() << ")" << endl; // compile error
}
```

- The methods getX and getY are not declared as const, therefore they cannot be called from p, which is declared as a const function argument
- The method set is declared as const, but it modifies the data members, which is not allowed for a const method

friend

- The **friend** keyword allows another class or function to use private members
- Example:

```
class A
{
   int x;  // x is private
   friend class B;  // class B has access to private members
   friend void foo(); // function foo() has access to private
   members
};

void foo(const A& a) { cout << a.x; }  // this is ok

class B
{
    void zoo(const A& a) { cout << a.x; }  // this is ok
};
```

Problem

- Then let's define a rectangle object as delimited by the position in the top-left corner and right-bottom corner (let's use the Position struct previously defined)
- Let's define various constructors for the rectangle object
- Let's implement the methods, width, height, perimeter, area, shift

Solution

```
struct Rectangle
{
   Position m_topLeft, m_bottomRight;
   Rectangle(const Position& topLeft, const Position& bottomRight)
      : m_topLeft(topLeft)   // calls constructor of Position object
      , m_bottomRight(bottomRight)
   {
   }
   Rectangle(const Position& topLeft, int width, int height)
      : m_topLeft(topLeft)
      , m_bottomRight(topLeft.m_x + width, topLeft.m_y + height)
   {
   }
   ...
};
```

- We define two constructors, with different parameter lists
- Note here I did not explicitly define the copy or default constructors, so they are automatically defined by the compiler. We have 3 constructors in total, 2 defined by us and 1 defined by the compiler (copy)
- There is no default constructor. It is not automatically added because we eplicitly defined other constructors

Solution (cont)

```
struct Rectangle
{
   Position m_topLeft, m_bottomRight;

   // constructors
   // ...

   // read only methods
   int width() const { return m_bottomRight.getX() - m_topLeft.getX(); }
   int height() const { return m_bottomRight.getY() - m_topLeft.getY(); }
   // note the use of the above accessor functions width and height in the
   // functions perimeter and area
   double area() const { return width()*height(); }
   double perimeter() const { return 2*(width()+height()); }

   // modifiers methods
   void shift(int dx, int dy)
       { m_bottomRight.shift(dx, dy); m_topLeft.shift(dx, dy); }
};
```

Destructors

- Every class or struct has exactly 1 destructor
- It is called when the object is deleted (if on the heap) or goes out of scope (if on the stack)
- **The synthax is**:

```
~StructName()  // the symbol ~ identify the destructor
{
  /* body */
}
```

- It takes no arguments and has no return type
- Used for cleanup tasks, e.g. releasing resources, close database connections

Example - Destructors

```
struct IntArray
{
   int *m_p;
   IntArray () : m_p(NULL) {}
   IntArray (int size) : m_p(new int[size]) {}
   ~IntArray() { delete [] m_p; }  // cleanup memory
};

void foo()
{
   // x is a local variable, i.e. it is on the stack
   // it internally allocates memory from the heap,
   // thus requiring at some point memory de-allocation
   IntArray x(100);
   ... // here do something with x
} // here x goes out of scope, the destructor is automatically
  // called, so releasing memory
```

Default Destructor

- If no destructor is explicitly defined, a default one is created by the compiler, which simply calls the destructors of all data members in reverse order (if they are objects with an associated destructor, trivial data members like int, or double do not have a destructor, only *struct* and *class* do)

Example – Default Destructor

```
struct IntArray
{
   int *m_p;
   IntArray() : m_p(NULL) {}
   IntArray(int size) : m_p(new int[size]) {}
   ~IntArray() { delete [] m_p; }
};

struct MyObject
{
   int m_x;
   IntArray m_p1, m_p2;  // MyObject data members are instances of IntArray
};
```

- No destructor is declared for MyObject, so a default one is created
- It will simply call in order the destructors of m_p2 and m_p1 (note that m_x has no destructor)

Destructor – Multiple Exit Points

- It is a very convenient feature, as, if there is any cleanup to do, it softens the burden of keeping track of all the possible exit point of functions
- For instance in the pseudo-code below there are 2 exit points
- In absence of a destructor, we would have to explicitly release memory in correspondence of each exit point

```
void foo(int n)
{
   IntArray x(n);
   if (someCondition)
      return;   // terminate the function: x goes out of scope
   doSomething
   return;      // terminate the function: x goes out of scope
}
```

Classes - Overloading Operators

- For struct and class it is possible to overload operators
- These are simply defined as data members
- Unary operators (e.g. !, ...) are intended to operate on the class itself
- Binary operators (e.g. [], +, +=, *, *=, =, ...) are intended to operate on the class itself as LHS and the argument of the function as RHS

Example - Overloading Operators

We overload the [] operator, for IntArray. It allows to use [] directly on an instance of the class. We create two versions, one const and one non-const. Note they returns a reference **int&** which allows us to also make assignments (e.g. x[0] = 4). Also, if instead of an array of int this was an array of some big objects, returning a reference would be more efficient as it avoids copying the object.

```
struct IntArray
{
   int *m_p;
   IntArray() : m_p(NULL) {}
   IntArray(unsigned size) : m_p(new int[size]) {}
   const int& operator[](unsigned i) const { return m_p[i]; } // const version
   int& operator[](unsigned i) { return m_p[i]; }  // non const version
   ~IntArray() { delete [] m_p; }
};

void foo( const IntArray& a)
{
   cout << a[0] << endl;  // this use the const version, because a is const
}

void main()
{
   IntArray x(100);
   x[0] = 4;    // this uses the non const version (we are making changes)
}
```

this

- **this** is a keyword
- It can be used inside any class members
- **It is a pointer to the current class**
- Example:

```
struct A
{
   A(int n) : m_x(n) {}
   int m_x;
   // here this is a pointer of type A*
   // pointing to the current instance
   void foo() { cout << this->m_x << endl; }
};

int main()
{
   A x(1), y(2);
   x.foo();  // prints 1
   y.foo();  // prints 2
}
```

Problem

- Define a struct with name String, which contains a null-terminated dynamic array of char
- It should have methods to assign a new string and to retrieve the string currently contained
- It should overload the **+=** method to append another string

Solution

- See: String.cpp
- Let’s familiarise with every single line of code in this example file
- Note we used the keywords **friend** and **this**

Assignment Operator

- Note in the String class we also defined an assignment operator: operator=
- This is another thing the compiler defines automatically, if not explicitly defined
- The default assignment operator, simply does an assignment of each member, calling the assignment operator of such members, if one exist
- In the case of the String class, we defined one explicitly, because it does something more complicated than just copying the data members

std::string

- To define a proper string class is quite complicate. Lot of functionality is needed.
- The C++ Standard Library provides with the **string** class
- Need to use **#include <string>**
- See: http://www.cplusplus.com/reference/string/string/
- It offers lot of functionality

Problem

- Let's use the std::string class. Write a program does:
  - Input from the console name and surname (hint: **>>** is overloaded also for **std::string**)
  - Concatenate the name and surname adding a space (hint: **+** is overloaded also for **std::string**)
  - Print the length of the final string (hint: the string class implements a method **length**)
  - Search if the full name contains the wovel '*i*' (hint: search the user guide of std:: string for an appropriate function to use)
  - Search if the full name contains the substring "*an*" (hint: search the user guide of std:: string for an appropriate function to use)

Solution

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name, surname;

    cout << "Enter name:" << endl;
    cin >> name;

    cout << "Enter surname:" << endl;
    cin >> surname;

    string fullName = name + " " + surname;  // + overloaded for string

    cout << "Full name: " << fullName << endl;
    cout << "Full name length: " << fullName.length() << endl;
    cout << (fullName.find_first_of( 'i' ) != string::npos ? "'i' found" : "'i' not found") << endl;
    cout << (fullName.find( "an" ) != string::npos ? "\"an\" found" : "\"an\" not found") << endl;
}
```

Composition and construction order

- A struct (or class) can have as data members other variable of type struct (or class). E.g. The Rectangle class had data members of type Position. (We have already seen this with MyObject, which contained instances of IntArray)
- When an object is constructed, all data members are also constructed in the same order in which they are declared
- When the object is destroyed, all data members are destroyed in reverse order

Nested classes and typedef

- Inside a class we can use **typedef**, we can declare **enum**, or we can define nested classes.
- These type definitions can be marked as public or private, like any other member
- Rationale: if a typedef, or an enum, or a class is only relevant in the context of another class, better to define it inside the class itself, to avoid pollution of the outer namespace. This is particularly useful in the context of templates and meta-programming, but we will not cover that in this course.
- The types defined inside a class can be accessed from outside the class with the same scope operator **::** used for namespaces.

Example – Nested Definitions

```
struct A
{
   // define a public subclass B
   struct B
   {
   };

   // define a public typedef
   typedef unsigned short mytype;

   // enum DayOfWeek {Monday, Tuesday, ...};
};

int main()
{
   A::B x;  // declare x of type A::B
   A::mytype y; // declare y of type unsigned short
   A::DayOfWeek d = A::Monday; // declare d of type A::DayOfWeek
}
```

Further Practice

Problem

- Design a data structure to describe the 3 towers of the Hanoi puzzle with N discs
- Hint: define a class Tower with the associated functionality, i.e. that of a stack: *push* and *pop* (a stack is like a tube of tennis balls, you can insert or extract balls, one at a time, from only one side of the tube, i.e. it is a queue that implements a LIFO policy).
- Implement a recursive solver to transfer all discs from tower 1 to tower 2 using tower 3 as an auxiliary tower. Print out your moves.
- See: https://en.wikipedia.org/wiki/Tower_of_Hanoi
- If you need help, look at my implementation: HanoiArray.cpp

Problem

- Complete as appropriate the implementation of the Matrix class in the next page
- The data storage should be of type **double***
- Overload the operator() with indices i and j to access the i-th row and j-th column of the matrix. Implement both const and non-const version
- Add functions nRows and nColumns
- Implement copy constructor.
- Implement assignment operator
- Careful: with copy constructor and assignment operators: every matrix must have its own storage for matrix data (i.e., they cannot share via a pointer the same storage space), so you cannot just copy the pointer.
- Implement destructor (release memory)
- To test your code, you can use the *main* function provided in the next page

Problem

```
class Matrix
{
   // write your implementation here
   // if you need help, you can look at my implementation in Matrix.cpp
};


int main()
{
    // create a 3 by 2 matrix and initialize all elements to 0.0
    Matrix m1(3, 2, 0.0);
    Matrix m2(m1);  // create copy of the matrix via copy constructor

    // requires methods nRows() and nColumns()
    for (unsigned i=0, n = min(m2.nRows(),m2.nColumns()); i < n; ++i)
       m2(i,i) = 1;   // requires ' double &operator()(int,int)'

    Matrix m3;  // invoke default constructor, make we set m_data=0
    m3 = m2;    // create copy of the matrix via assignment operator
    m3 = m3;    // is your assignment operator smart enough to handle this correctly?

    const Matrix& cm = m3;  // const reference to m3
    for (unsigned i=0; i < cm.nRows(); ++i)
        for (unsigned j=0; j < cm.nColumns(); ++j)
            cout << cm(i,j) << endl; // requires 'const double operator()(int,int) const' const

    return 0;
}
```