Object Oriented Programming in C++

The course will deal with

- Basic Concepts of C++:
  - Introduction into “objects”
  - Inheritance
  - Generic Programming
  - Safe design of objects
  - Overloading Operators for Objects
  - Streams for reading and writing to files
- Using C++ Libraries
  - LEDA, POLKA, ...

Course Organisation

- *One Point Course*
- *How to reach me for suggestions and questions:*
  - office:2446;
  - email: khofer@cs.chalmers.se
- *Exercises*
  - every week a small (programming) exercise to be handed in by mail or as a hardcopy before the beginning of next class
  - alternatively you can suggest an exercise yourself

Course Organisation II

- *Literature:*
  - Bjarne Stroustrup
    The C++ Programming Language
    Addison Wesley
  - LEDA Manual
    http://www.cs.chalmers.se/Support/Software/LEDA/index.html
  - POLKA Information
    http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html

Outlook for this Lecture

- Why bothering about C++
- Introduction into object orientation
- Some additional syntax compared to C
- How simple objects can be defined in C++

Why bothering about C++

- C++ provides *object oriented concepts* to the programmer
- Understanding these concepts will make it easy to use other languages using object orientation like JAVA
- C++ includes C as a subset and is still closely related to the efficiency provided by C
- It is still possible to program closely to the machine level

What is an Object

- An object is supposed to represent a concept the programmer needs to find a representation for.
- Many data structures we can describe as objects e.g. a stack, dictionary or even a list
- Some objects can be described as its “real world” metaphor like in graphics circle and square can be
- The programmer describes the object in terms of
  - state variables
  - methods which allow to modify the objects state

Objects in C++

- In C++ an object is defined by class
- A class associates a type (like int, char,...) with each object

Example of Using an Object in C++

```
#include <iostream.h>
#include <LEDA/matrix.h>

void main()
{
  matrix A, B;          // defines objects A and B of type matrix

  cin >> A;             // reads from the input A
  cin >> B;             // reads from the input B

  if ((A.dim1()==A.dim1()) && (A.dim2() == A.dim2()))
    cout << A + B;      // outputs A+B
}
```

Is this Object Oriented?

```
enum Kind {circle, triangle, square}

class Shape
{
  Kind k;
  Point center;
  Color col;
  // ...
public:
  void draw();
  void rotate();
  // ...
};
```



```
void Shape::draw()
{
  switch(k) {
  case circle:                    // draw circle
    break;
  case triangle:                  // draw triangle
    break;
  case square:                    // draw square
    break;
  }
}
```


How is such an Object Defined?

```
class matrix{
  /* internal variables and functions */

public:
  int dim1();
  int dim2();
  /* variables and functions which allow interaction with
     the object */
}
```


Object Oriented Programming

- Using objects is not enough to specify an object oriented program
- The design of an application requires to identify the different concepts needed.
- If concepts are related the objects should take this into account e.g. circle and triangle have both in common that they are a shape.
- A program which uses objects that closely match the concepts of the application and consider common concepts between them can be seen as *objected oriented*.

A Better Approach

We should design shape in a way that easily new objects can be added.

In C++ we can achieve this by *inheritance*.


```
class Shape
{
  Point center;
  Color col;
  // ...

public:
  virtual void draw()=0;
  // ...
};

class Circle : public Shape {
  int radius;
public:
  void draw(){...}
  // ...
}
```


Syntax in C++

- Recall, you can use all that you know about C.
- Commands like if, for, while, switch work all the same
- Also recall your knowledge on basic types, arrays and structures
- Pointers will play an important roll throughout this course
- If you feel not confident with pointers, I recommend reading Part I of Stroustrup's book, in particular (chapter 5).

C program structure

```
#include <stdio.h>
// a set of functions each performing a computation
int f1(char x, int y, char*z)
{
  return y;
}
// f2,...,fn
main()                                        // the main function
{
  char x='a';
  int y = 2;
  printf(''Computed:  %d'',f1(x,y,&x));
}
```

Types, Pointers, Arrays and Structures to Recall

- *char, int, float, double, ...*
- Pointer to a variable gives the address where a variable is located.
- A static array can be created by [], e.g. *int* c[20].
- For creation of a dynamic array you had to use malloc or alloc.
- To implement more complicated data structures we could use *struct*

References

A *reference* is an alternative name to an object.

References are mainly used for specification of

- arguments
- return values for functions

Notation:

X& means reference to X

Example (Reference I)

```
#include <iostream.h>

void change_val(int& x)
{
  x++;
}

main()
{
  int y=0;                      //initialize y with 0
  change_val(y);
  cout << y << ''\n'';          // result 1
}
```


Example (Reference II)

```
int i=1;
int& r = i;        // r and i refer now to the same int
int* p = &i;       // p points to the address of i
int x = r          // x = 1
*p = 7             // r = i = 7
r = 4              // i?
```

Memory Allocation

Instead of using *alloc, malloc* and *free* one can use *new* and *delete* for memory allocation and deallocation.

```
main()
{
  int *x, *y;                      // pointers of int
  y = new int;                     // allocates memory for one integer;
  x = new int[10];   // allocates memory for an array of 10 integers
  for (int i=0; i<10; i++)
    x[i]=0;
  delete[] x;                      // deallocates memory of the array
  delete y;                        // deallocates memory of the integer
}
```


Classes

The class concept is used for creating user defined types. Hereby the idea is that a program tends to be easier to understand when providing types that closely match the concepts of the application.

In C++ a class is defined by using either

- *class* X $\{\ldots\}$;

or

- *struct* X $\{\ldots\}$;

Member Functions I

Implementation of the concept of a date using *struct* to define the representation of a *date* and a set of functions for manipulating variables of this type:

```
struct Date{
  int d, m, y;
};

void init_date(Date& d, int dd, int mm, int yy);    // initialize d,m,y
void add_year(Date& d, int n);                      // add n years to d
void add_month(Date& d, int n);                     // add n years to m
void add_day(Date& d, int n);                       // add n years to y
```


Member Functions II

In the previous example there is no connection between the representation and the manipulating functions. This can be achieved in the following way:

```
struct Date{
  int d, m, y;

  void init_date(int dd, int mm, int yy);    // initialize d,m,y
  void add_year(int n);                      // add n years to d
  void add_month(int n);                     // add n years to m
  void add_day(int n);                       // add n years to y
};
```

Member Functions III

Functions within a class definition are called *member functions*. Note that *struct* is a kind of class. When a member function is called it knows for which object it is invoked.

Example:

```
void f()
{
  Date today;
  today.init(2,12,1999);

  Date tomorrow = today;
  tomorrow.add_day(1);
}
```


Access Control

In order to express that the representation of Date (given by d, m and y) can only be manipulated by the member functions we can write:

```
class Date{
  int d, m, y;
public:
  void init_date(int, int, int);      // initialize d,m,y
  void add_year(int n);               // add n years to y
  void add_month(int n);              // add n years to m
  void add_day(int n);                // add n days to d
};
```


Access Control II

The *public* label separates the class body in two parts. The first part (*private*) is only accessible by member functions; the second part (*public*) is the public interface to the class.

Reasons for using Access Control:

- any illegal value (35/13/1999) must be caused by code in a member function ⇒ easier localisation of programming errors
- Changing representation of the class implies only changing the code of the member functions, but the interface for accessing the class may stay the same. ⇒ program code using the class can stay the same
- A potential user may only need to study the definition of the public class members in order to use the class in his programs.