Overview on Today's Lecture

- Last Lecture
- Problems in Exercises?
- Object Initialisation
- Dynamic Objects: allocation and deallocation of memory
- Copying Objects and Assignments
- About Operators

Last Lecture

- Object Oriented Programming
- Definitions of Objects in C++
- References

Initialisation

- Definition of a variable will allocate a memory position for the variable
- Unless you initialise or assign a value to this variable the value can be anything
- A clean solution provides a possibility to initialise objects to a certain state.

Constructors

Constructors are used to initialise class objects.

```
class Date{
  int d, m, y;
public:
  // ...
  Date(int, int ,int);              // day, month, year
  Date();                           // default date
  Date(const char*);                // date in string representation
}

Date today(1,12,99);
Date now;
Date xmas = Date(25,12,1990);
Date str = Date(''December 1, 1999'');
```

Why Using Constructors?

- Constructors force the initialisation of objects. Uninitialised Objects might lead to errors which are difficult to locate.
- More elegant approach in comparison to *date.init()*
- For dynamic objects we can allocate memory.

Using Default Arguments in Constructors

For constructors you are allowed to use default arguments:

```
class Date{
  int d, m, y;
public:
  //...
  Date(int dd=0; int mm=0; int yy=0);
}

Date day(12);
Date month(12, 7);
Date year(12, 7, 99);
```


Static Members

- Can be used to initialise objects to a default value
- While the program evolves the default value can be changed
- Later objects are initialised to the new default value
- The name *static* originates from the term *static variable* in C

Recall Static Variables from C

```
#include<iostream.h>
void f(){
  static int i=0;
  if (i==0){
    cout << ''if branch \n'';
    i++;
  }
}
main(){
  while(1)
    f();
}
```

Static Members Example

```
class Date{
  int d,m,y;
  static Date default_date;
public:
  Date(int, int, int);
  static void set_default(int, int, int);
};
Date Date::default_date(1,12,99);
void Date::set_default(int dd, int mm, int yy)
{
  default_date = Date(1, 1, 2000);
}
void f() {Date::set_default(25,12,99)}
```


Remarks on Static Members

A static member function can be called without initialising an object! Note this is not possible with ordinary member functions.

Static members must be defined somewhere in the program code.

Copying Class Objects

- class objects can be copied
- by default copying is done by copying each member
- copying can also be defined by the programmer which might be necessary for dynamic objects

```
Date d=today;                    // initialization

void f(Date& d)
{
  d = today;
}
```


Self References

We would like to write expressions like

d.add_day(1).add_month(1).add_year(2)

Therefore it is allowed for member functions to return a reference of its object after updating it:

```
class Date {
  // ...
  Date& add_year(int n);
  Date& add_month(int n);
  Date& add_day(int n);
}
```

How to Return a Reference?

For returning a reference a member function can return the *this* pointer:

```
Date& Date::add_year(int n)
{
  this→y += n;
  // y +=n;
  return this;
}
```


Objects and Dynamic Memory

When an object is destroyed during the program execution, e.g. after the end of a function call, then all static memory is automatically deallocated.

This does *not* automatically hold for dynamic memory allocated within an object. In this case the programmer must handle deallocation of memory in order to make sure that the program will not run out of memory.

In classes which allocate memory the programmer should define a *destructor*.

Bad Object Definition

```
class Table
{
  int* t;
public:
  Table(int sz){t = new int[sz];}
};

void f(){Table tab(5);}

main()
{
  while (1) f();                    // will continously allocate memory
}
```


Object Definition with Destructor

```
class Table
{
  int* t;
public:
  Table(int sz){t = new int[sz];}
  ~Table(){delete[] t;}   // deallocate dynamic memory
};

void f(){Table tab(5);}

main()
{
  while (1) f();
}
```

Copying dynamic Objects

In addition to the constructor, discussed during last lectures, there exists a copy constructor for every object.

The copy constructor is called when an object is given as a parameter to a function or an object is initialised by some other object.

It creates a copy of the object such that the original object will not be changed when changing the copy.

By default (i.e. without further definition) the copy constructor copies the members of the class.

For pointers this is not sufficient since the copy of an object will point to the same address than the original object.


```
class Date{
  int d,m,y;
public:
  Date(int, int, int);                    // constructor
  Date(const Date&);                      // copy constructor
  int get_day() const;
  int get_month() const;
  int get_year() const;
};

Date::Date(const Date& dat)
{
  d = dat.get_day();
  m = dat.get_month();
  y = dat.get_year();
  cout << ''Hi!\n'';
}
```


When is the Copy Constructor Used?

```
Date f(Date x)                            // Copy of Date
{
  Date y=x;
  return Date;
}

main()
{
  Date x, y, z;
  y = f(x);                               // ?
  z = y;                                  // assignment
}
```


Assignment vs. Copying

In assignments we have an already existing object which should have the same value than the expression that we assign to the object.

By copying we create a new object which will have the same value than the expression we used for initialisation.

Analogous to the copy constructor, the assignment is by default defined as memberwise assignments.

More Operators Overloading

We can declare functions that have meanings for following operators:

+	−	∗	/	%	∧	&
|	∼	!	=	<	>	+=
−=	∗=	/=	%=	∧=	&=	|=
<<	>>	>>=	<<=	==	!=	<=
>=	&&	||	++	−−	−>∗	,
−>	[]	()	*new*	*new*[]	*delete*	*delete*[]

For the following operators you cannot declare a meaning:

:: . .∗

Benefits of Operators

In particular for algebraic objects, operators overloading allows a more convential and convenient notation for manipulation of objects.

Also for more abstract types it might be more convenient to use operators like [] or ().

However, using operators in a way they are not intentionally meant to be used can lead rather to confusion.

The Assignment Operator

```
class Date{
  int d,m,y;
public:
  // ...
  Date& operator=(const Date&);            // assignment operator
};

Date::Date(const Date& dat)
{
  d = dat.d;                 // using private d,m and y is allowed!
  m = dat.m;
  y = dat.y;
  return *this;
}
```


Example for a Dynamic Object

Try to define a save vector object which is initialised by its size.

Which member functions are necessary?

Binary and Unary Operators

A *unary operator* is either

- a non static member function taking no argument
- a non-member function taking one argument

A *binary operator* is either

- a non static member function taking one argument
- a non-member function taking two arguments

Example

```
class X{
public:
  void operator+(int);                // ?
  X(int);
};

void operator+(X,X);                  // ?
void operator+(X,double);             // ?
void f(X a)
{
  a+1;                                // ?
  1+a;                                // ?
  a+1.0;                              // ?
}
```


Notes

- Operators can only be overloaded in the way they are defined in the grammar.
- For instance, you cannot define a unary operator% or a ternary operator+.
- Some operators like = or & have predefined meanings.
- Predefined meanings can be made inaccessible by declaring them as private.
- Be aware that for operator definitions for + and = do *not* imply a definition for + =. You will have to define this relation yourself.
- Double definitions of operators e.g. one as a member and one a a non-member are not possible