

Overloading Operators

- Overloading operators
 - Unary operators
 - Binary operators
 - Member , non-member operators
- Friend functions and classes
- Function templates and class templates

Overloading Operator

- Operator overloading is a very neat feature of object oriented programming
- Operator overloading allows it to give normal C++ operators such as +, -, ==, < additional meanings
- It makes statements more intuitive and readable for example:

```
Date d1(12,3,1989);
Date d2;
d2.add_days(d1,45);
// can be written with the + operator as
d2=d1+45;
```

Operator Overloading

- The name of an operator function is the keyword operator followed by the operator itself.
- ```
class complex
{
 double re, im; // re and im part of a complex number
public:
 complex (double r, double i) : re(r), im(i) {};
 complex operator+(complex c);
};
complex c1(2.2,3.0); // instantiate complex c1
complex c2(1.0,-4.5); // instantiate complex c2
complex c3=c1+c2; // shorthand notation for c1 + c2
complex c4=c1.operator+(c2); // explicit call
```

## Overloading Unary Operators

```
class Date
{
 void operator++(); // prefix increment operator
}
void Date::operator++ ()
{
 if (++day > days_in_month())
 {
 day=1;
 if (++month > 12)
 month=1;
 year++;
 }
}
Date d(31,12,1999);
++d; // results in 1.1.2000
```

## Overloading Unary Operators

```
class Date
{
 Date operator++(); // prefix increment operator
}
Date Date::operator++ ()
{
 if (++day > days_in_month())
 {
 day=1;
 if (++month > 12)
 month=1;
 year++;
 }
 return *this; // self-reference to the object
}
Date d1(31,12,1999);
Date d2=++d1; // results in 1.1.2000
```

## Overloading Unary Operators

```
class Date
{
 Date operator++(int); // postfix increment operator notice int!!
}
Date Date::operator++ (int)
{
 Date tmp=*this;
 if (++day > days_in_month())
 {
 ...
 }
 return tmp; // return copy of object
}
Date d1(31,12,1999);
Date d2=d1++; // results in d2 = 31.12.1999, d1 = 1.1.2000
```

## Overloading Binary Operators

```
class Date
{
 Date operator+(int days) const;
};
Date Date::operator+(int days) const;
{
 Date tmp=*this; // copy object
 for (int i=0; i < days; i++)
 tmp++;
 return tmp;
}
Date d1(1,4,1999);
Date d2=d1+25; // results in 26.4.2000
```

## Overloading Binary Operators

```
class Date
{
 Date& operator+=(int days); // must be reference as += modifies
 // the left hand argument
};
Date& Date::operator+=(int days) // return type reference to object
{
 for (int i=0; i < days; i++)
 *this++;
 return *this; // return reference to object
}
Date d1(1,4,1999);
d1+=25; // results in 26.4.2000
```

## Overloading Relational Operators

```
class Date
{
 bool operator==(Date d)
 {
 return (day==d.day) && (month==d.month) && (year==d.year);
 };
 bool operator<(Date d)
 {
 if (year < d.year)
 return true;
 else
 if (year==d.year) && (month < d.month)
 return true;
 else
 return (month==d.month) && (day < d.day);
 };
};
```

## Overloading Binary Operators

```
int Date::operator-(Date d) const
{
 int days=0;
 if (*this > d)
 while (*this != ++d)
 days++;
 else
 while (*this != --d)
 days--;
 return days;
}
Date d1(24,4,1988);
Date d2(13,3,1998);
int diff = d1-d2; // diff = 42
```

## Non-Member Function Operator

- Operators can also be implemented as non-member functions instead of member functions
- However this approach is usually not recommended as member data is private to the operator
- Necessary for some operators in which the class object does not appear on the left hand side of the operator for example <<, >>

## Non-Member Function Operator

- A binary operator can be defined either by a member function taking one argument or a non-member function taking two arguments

```
class Date
{
 Date operator+(int n); //member function one argument
}
Date Date::operator+(int n)
{ ... };

Date operator+(Date d, int n) //non-member func two arguments
{
 Date temp=d;
 for (int i=0; i<n; i++)
 temp++;
 return temp;
}
```

## Non-Member Function Operator

- An unary operator can be defined either by a member function taking no argument or a non-member function taking one argument

```
class Date
{
 Date operator++(); //member function postfix increment no argument
}
Date Date::operator++()
{ ... };

Date operator++(Date& d) // non- member function one reference argument
{
 // more complicated as member data is private to non- member function
 int tmp_day= d.get_day();
 int tmp_month= d.get_month();
 d.set_day(tmp_day+1);
 ...
}
```

## Overloading << Operator

- Overloading the << operator allows you to specify the way in which an object is displayed
- As the << operator expects a stream object as the left hand argument it must be overloaded as a non-member function :

```
ostream& operator<<(ostream& os, Date d);
```

- It is possible to grant the non-member function access to the private data member of the class by declaring the function as a *friend* of the class.

## Overloading << Operator

```
#include <iostream>
class Date
{
 friend ostream& operator<<(ostream& os, Date d);
 // this non- member function is a friend of class date
 ...
};

ostream& operator<<(ostream& os, Date d)
{
 os << d.day << "." << d.month << "." << d.year; //access private data as friend
};

Date d1(16,3,1998);
cout << "Today's date is: " << d1 << endl;
```

## Friend Mechanism

- The friend mechanism is useful when some function has to operate with two classes simultaneously so that it can not be a member function of both classes at the same time

```
class Matrix; // declares that class Matrix exists somewhere
class Vector {
 double v[4];
 friend Vector operator* (const Matrix&m, const Vector& v);
};
class Matrix{
 Vector v[4];
 friend Vector operator* (const Matrix&m, const Vector& v);
};
```

## Friend Mechanism

```
Vector operator* (const Matrix& m, const Vector& v)
{
 Vector r;
 for (int i=0; i<4; i++) {
 r.v[i]=0; //friend is allowed to access private data of Vector
 for(int j=0; j<4; j++)
 r.v[i]+= m.v[i].v[j] * v.v[j]; // friend access to
 //private data of Matrix
 }
 return r;
}
```

## Friend Mechanism

- It is also possible that one class is friend to another class in which case all member function of the friend class can access the private data

```
class Vector {
 friend class Matrix // declares Matrix as a friend of Vector
 ...
}

Vector Matrix::operator*(const Vector& v) // member function of Matrix
{
 Vector r;
 ...
 r.v[i] += v.v[i] * v.v[j]; // Matrix accesses private data of class Vector
 ...
}
```

## An Array Class

```
class Array
{
public:
 Array(int sz) { size = (sz > max_size) ? max_size : size; };
 double& operator[](int i); // returns a reference to i-th element
 double operator()(int i); // returns the value of i-th element
private:
 static int max_size=100;
 int size;
 double array[100];
};
```

## An Array Class

```
double& Array::operator[](int i)
{
 if (i < sz)
 return array[i]; // return reference to i-th element
 else
 {
 cout << "index exceeds range !" << endl;
 return array[sz-1]; // return last element
 }
};
Array x(50);
x[23]=3.5; // assignment requires reference
x[12]=x(13)+x[10]; // right hand side either value or reference
```

## Templates

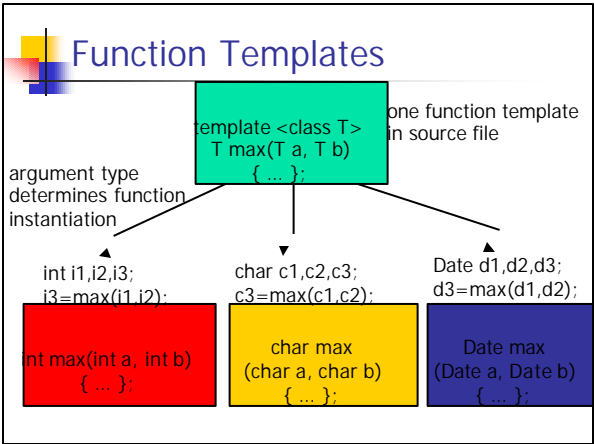
- A template is a place-holder for an arbitrary built-in or user-defined data type
- Templates make it possible to use one function or class to handle many different data types
- Function templates allow a parameter to assume an arbitrary data-type
- Class templates allow a member data to assume an arbitrary data-type
- Templates are another example for polymorphism

## Function Templates

```
int max(int a, int b) // one max function for int
{
 return (a>b) ? a : b;
}
double max(double a, double b) // max function for double
{
 return (a>b) ? a : b; //identical code
}
```

## Function Templates

```
template<class T> // class Type placeholder for concrete data type
T max(T a, T b) // substitute template T for concrete data type
{
 return(a>b) ? a : b; // identical code
}
void main()
{
 int a=3, b=2;
 Date d1(17,5,1998); // operator > is defined for class Date
 Date d2(23,6,1997);
 int c=max(a,b); // template T replaced with int
 char z=max('f','q'); // template T replaced with char
 Date d3=max(d1,d2); // template T replaced with date
}
```





## Class Templates

```
template<class Type> // template class Type
class Array // array of arbitrary data type
{
public:
 Array(int sz) { size = (sz > max_size) ? max_size : size; };
 Type& operator[](int i); // returns a reference to i-th element
 Type operator()(int i); // returns the value of i-th element
private:
 static int max_size=100;
 int size;
 Type array[100]; // placeholder for concrete data type
};

Array<double> x(20); // instantiates a double array of size 20
Array<Date> days(10); // instantiates a Date array of size 10
```