Generisk programmering

- C++ erbjuder möjligheter att definiera och implementera generella (eller abstrakta) funktioner och klasser
- Detta system kallas *Template Mechanism*
- Genom att specificera ett template, kan kompilatorn generera kod, som är anpassad till den egentliga datatyp som används
- Template-mekanismen kan ses som jobbig i början, men efterhand brukar man se nytta av den

Generisk programmering: intro

```
// Generell funktion: Addera två argument
Type add(Type const &lvalue, Type const &rvalue) {
  return lvalue + rvalue;
}
//För type double
double add(double const &lvalue, double const
  &rvalue) {
  return lvalue + rvalue;
}
```

– Detta kan förstås upprepas förvarje tänkbar datatyp, användande av överladdade funktioner... men leder till ...många... versioner av funktionen

Generisk funktion

```
template <typename Type>
Type add(Type const &lvalue, Type const
  &rvalue) {
  return lvalue + rvalue;
}

// Nu kan vi använda
double a=1.2, b=3.34;
printf(”sum of %f and %f is %f\n”, a, b,
  add(a,b));

// Kompilatorn genererar nu en överladdad
  funktion av rätt typ
```

Format på template-funktion

Keyword

Kommaseparerad lista med templat-parameterlista

```
template <typename Type>
Type add(Type const &lvalue, Type const
  &rvalue) {
  return lvalue + rvalue;
}
```

Vi kan notera, att hittills har vi endast använt formella variabelnamn som argument till funktioner, typerna har varit givna.

Nu låter vi även typerna på argumenten fungera som formella namn.

Exempel med den komplexa datatypen

```
template <typename Type>
Type add(Type const &lvalue, Type const
  &rvalue) {
  return lvalue + rvalue;
}

int main() {
  cout << "Sum of " << x << " and " << y <<" =
  " << add(b,c) <<endl;
}
```

Hur avgörs argumenttyp??

- Endast funktionargument används, inte lokala variabler eller retur-värden
- 3(4) konversioner kan användas
  - lvalue till rvalue
  - tilläggande av const till icke-const
  - konversion till bas-klass
  - standard konversioner (int to double, int to unsigned etc)

Template-klasser

- Används i STL (Standard Template Library)
- Typisk för klasser som enkapsulerar andra datatyper såsom
  - Vektorer
  - Matriser
  - Länkade listor

Komplexa klassen med templat

```
template <typename T1>
class Complex {
public:
   Complex(T1 re=0.0, T1 img=0.0) {m_re=re;m_img=img;}
   Complex(const Complex &other) {Copy(other);}

   Complex &operator=(const Complex &other) {Copy(other); return *this;}
   Complex operator*(const Complex &other) const;
   Complex operator/(const Complex &other) const;
   Complex operator+ (const Complex &other) const {Complex res(*this);
   res.m_re+=other.m_re;res.m_img+=other.m_img; return res;}
    //Complex const operator+(const Complex &other) {Complex res(*this);
   res.m_re+=other.m_re;res.m_img+=other.m_img; return res;}
   int a;
   //ostream &operator<<(ostream &stream, Complex const &c);

   T1 const Re() {return m_re;}
   T1 const Img() {return m_img;}
   const char *c_str();
private:
   void Copy(const Complex &other) {m_re=other.m_re; m_img=other.m_img;}
   double m_re;
   double m_img;
   char m_c_str[20];
};
```

Komplexa tal 2

```
// (ac-bd) + i(ad+bc)
template <typename T1>
Complex<T1> Complex<T1>::operator*(const Complex<T1> &other) const {
   Complex res;
   res.m_re = m_re*other.m_re - m_img*other.m_img;
   res.m_img = m_re*other.m_img + m_img*other.m_re;
   return res;
}
//  (ac+bd) + i(bc-ad)
//  ------------------
//      c*c+d*d
template <typename T1>
Complex<T1> Complex<T1>::operator/(const Complex<T1> &other) const{
   Complex<T1> res;
   double den = other.m_re*other.m_re + other.m_img*other.m_img;
   res.m_re = (m_re*other.m_re + m_img*other.m_img) / den;
   res.m_img = (m_img*other.m_re - m_re*other.m_img) / den;
   return res;
}
template <typename T1>
const char *Complex<T1>::c_str() {
   sprintf(m_c_str, "%f%s%fj", m_re, m_img<0?"":"+", m_img);
   return m_c_str;
}
```

Komplexa 3

```
typedef Complex<double> complex; // Komplex klass med double som element
//typedef Complex<int> complex; // Komplexx klass med heltal som element

int main(int argc, char* argv[])
{
   complex b(3.4, 1.2);
   complex c(2.2, -5.2);
   complex sum = b+c;
   complex prod;
   prod = b*c;
   double x=1.22, y=4.55;

   using namespace std;

   printf ("sum of %s and %s is %s\n", b.c_str(), c.c_str(), sum.c_str());
   printf ("prod of %s and %s is %s\n", b.c_str(), c.c_str(),
   prod.c_str());
   printf ("div of %s and %s is %s\n", b.c_str(), c.c_str(),
   (b/c).c_str());

   cout << "Sum of " << x << " and " << y <<" = " << add(b,c) <<endl;
}
```

Ex: Stack

```
template <class T>
class Stack {
public:
Stack(int = 10) ;
~Stack() { delete [] stackPtr ; }
int push(const T&);
int pop(T&) ;
int isEmpty()const { return top == -1 ; }
int isFull() const { return top == size - 1 ; }
private:
int size ; // number of elements on Stack.
int top ;
T* stackPtr ; } ;
```

Ex: Stack (2)

```
//constructor with the default size 10
template <class T>
Stack<T>::Stack(int s) {
  size = s > 0 && s < 1000 ? s : 10 ;
  top = -1 ; // initialize stack
  stackPtr = new T[size] ;
}
// push an element onto the Stack
template <class T>
int Stack<T>::push(const T& item) {
   if (!isFull()) {
        stackPtr[++top] = item ;
        return 1 ; // push successful
   }
  return 0 ; // push unsuccessful
}
```

Ex: Stack (3)

```
// pop an element off the Stack
template <class T>
int Stack<T>::pop(T& popValue) {
  if (!isEmpty()) {
     popValue = stackPtr[top--] ;
     return 1 ; // pop successful
  }
  return 0 ; // pop unsuccessful
}
```

Ex: Stack (4)

```
#include <iostream>
#include "stack.h"
using namespace std ;
void main() {
  typedef Stack<float> FloatStack ;
  typedef Stack<int> IntStack ;
  FloatStack fs(5) ;
  float f = 1.1 ;
   cout << "Pushing elements onto fs" << endl ;
   while (fs.push(f)) { cout << f << ' ' ; f += 1.1 ; }
   cout << endl << "Stack Full." << endl << endl << "Popping elements
   from fs" << endl ;
   while (fs.pop(f))
     cout << f << ' ' ; cout << endl << "Stack Empty" << endl ;
   cout << endl ;
   IntStack is ;
   int i = 1.1 ;
   cout << "Pushing elements onto is" << endl ;
    while (is.push(i)) { cout << i << ' ' ; i += 1 ; }
    cout << endl << "Stack Full" << endl << endl << "Popping elements
   from is" << endl ;
   while (is.pop(i)) cout << i << ' ' ; cout << endl << "Stack Empty"
   << endl ;
}
```