Tillämpad programmering

C++ objekt

Johan Montelius

struct

```
struct Person {
     string    name;
     int       age;
};
```

```
         :
    Person p;
         :
    p.name = “Joe”;
    p.age = 42;
         :
    cout << p.name << endl;
```

Java class

```
public class IntCell {
      public IntCell() {
            this( 0 );
      }
      public IntCell( int initial ) {
            stored = initial;
      }
      public int getValue( ) {
            return stored;
      }
      public setValue(int value) {
            stored = value;
      }
      private int stored;
}
```

C++ class

```
class IntCell {

   public:
     IntCell(int initial = 0) {
          stored = initial;
     }
     int getValue( ) {
          return stored;
     }
     void setValue(int value) {
          stored = value;
     }

   private:
      int stored;
};
```

C++ class

- En klass i sig, är inte publik eller privat.
- Metoderna är privata om de inte föregås av “public:”
  - i en struct är allt publikt
- I Java kan en konstruktor anropa en annan, i C++ har man default-värden och flera konstruktorer som ger liknande funktionalitet.

Erlang

```
-record(person, {name, age}).
```

```
        :
P = #person{name = “Joe”, age=23},
        :
Name = P#person.name,
Age = P#person.age,
        :

Q = P#person{age=25},
        :
```

separat definition av metoder

```
class IntCell {
   public:
     IntCell(int initial = 0);
     int getValue( );
     void setValue(int value);
   private:
      int stored;
};

IntCell::IntCell(int value = 0) {
     stored = value;
}

     :
```

intcell.h

```
class IntCell {

    public:
        IntCell(int initial);

        int getValue( );

        void setValue(int value);

    private:
         int stored;
};
        :
```

intcell.cc

```
IntCell::IntCell(int value = 0) {
      stored = value;
}

      :
```

constructor

```
IntCell::IntCell(int value = 0) {
        cout << "IntCell: " << value << endl;
        stored = value;
}

        :
```

main.cc

```
include “intcell.h”
int main() {
    IntCell n1;
    IntCell n2(10);
    IntCell n3 = 42;
    n2 = 37;
    cout << “hmmm?” << endl;
};
```

flera konstruktorer

```
class Date {
        int day, month, year;
    public:
        Date(int dd = 0, int mm = 0, int yy = 0) {
            day = dd;
            month = mm;
            year = yy;
        };
        Date(int dd, string mm, int yy = 0) {
            :
        };
};
```

static

```
class Date {
        int day, month, year;
        static Date default_date;
   public:

        static Date set_def(int dd, int mm, int yy) {
              default_date = Date(dd, mm, yy);
        };

              :
};


Date Date::default_date = Date(1,1,1990);
```

initialisering

```
class Date {
        int day, month, year;

   public:

        Date(int dd, int mm, int yy) {
                day = dd;
                month = mm;
                year = yy;
        };

                :
};
```

initialisering

```
class Date {
        int day, month, year;

    public:

        Date(int dd, int mm, int yy)
                : day(dd), month(mm), year(yy) {
        };

                :
};
```

initialisering

- Att föredra, oftast mest effektivt,
  - undviker att sub-objekt först skapas för att sedan sedan skrivas över.
- Måste användas om sub-objekt inte har en parameterlös konstruktor.
- Kan inte alltid användas för mer komplexa initialiseringar.

const

```
int main() {
    const int a = 3;
    cout << square(a) << endl;
}
```

const

```
class IntCell {

   public:
     IntCell(int initial = 0);

     int getValue()  const;

     void setValue(int value);

     int square()  const;

   private:
      int stored;
};
     :
```

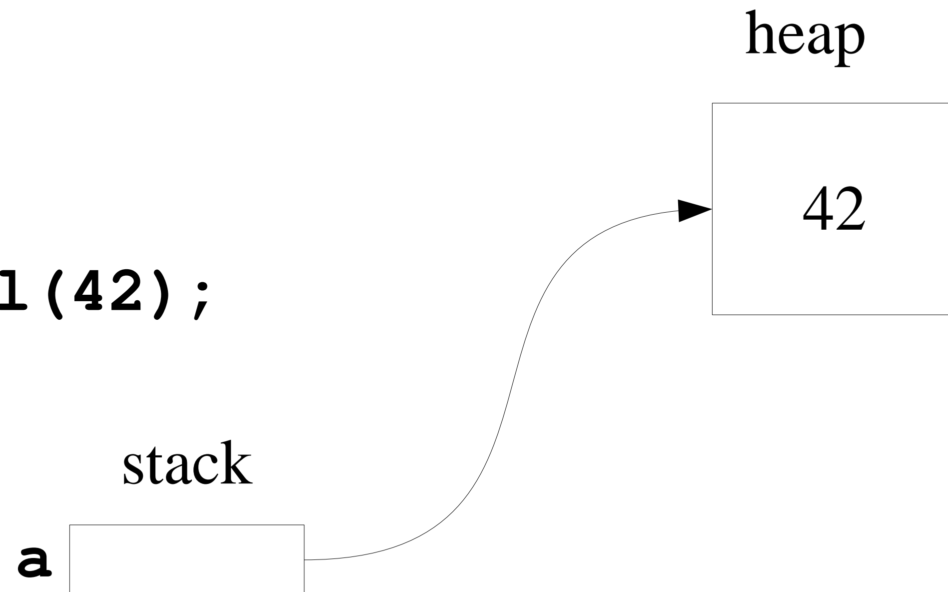
const

- Viktig del i beskrivningen av en metod.
  - ”accessor”
  - ”mutator”
- Kompilatorn kan göra optimeringar men
  - framför allt ett sätt att dokumentera
- Man kan modifiera ett const-deklarerat objekt (med hjälp av mutable) om det inte förändrar semantiken.

delete

```
        :
IntCell *a = new IntCell(42);

        :
delete a;
        :
```



delete

```
bool foo(int x) {

     bool res;

     IntCell *a = new IntCell(42);

     if (a->get_val() < x)
          res = true;
     else
          res = false;

     return res;
}
```

delete

```
IntCell *foo(int x) {

     IntCell *a = new IntCell(42);
          :
          :
     delete a;

          :
     return a;
}
```

delete

```
void foo(int x) {

     IntCell *a = new IntCell(42);
          :
          :
     delete a;
          :

     delete a;
          :

}
```

Student

```
class Student {

        IntCell *age;

   public:
        Student(int a) {
            age = new IntCell(a);
        }
        :

}
```

Student

```
    :
Student *s = new Student(24);

delete s;
    :
```

```
class Student {

        IntCell *age;

    public:
        Student(int a) {
            age = new IntCell(a);
        }
        :

}
```

destructor

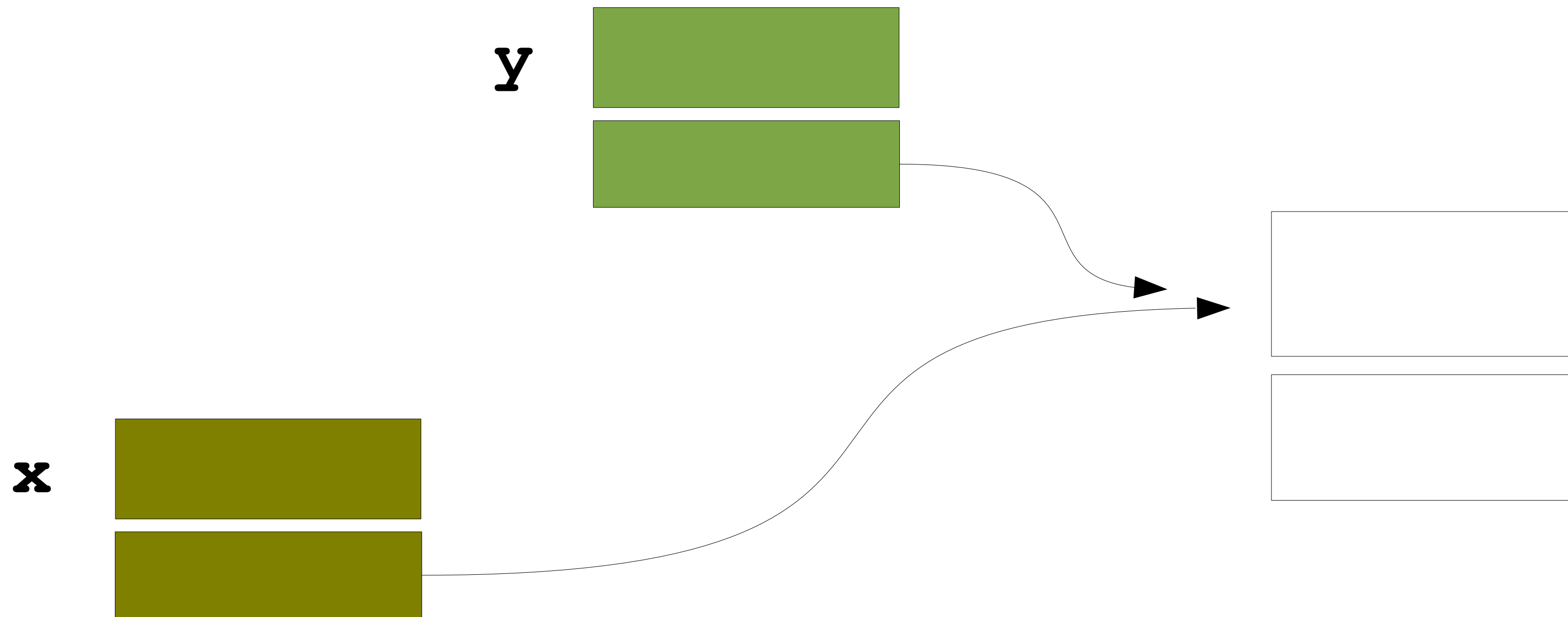
```
class Student {

     IntCell *age;

  public:
     Student(int a) {
        age = new IntCell(a);
     }

     ~Student() {
        delete age;
     }
        :
```

Vem gör vad



the big three

- destructor
  - anropas när ett objekt skall av-allokeras
- copy constuctor
  - används när ett objekt initialiseras utgående från ett existerande objekt
- operator=
  - används vid tilldelning i.e. x = y

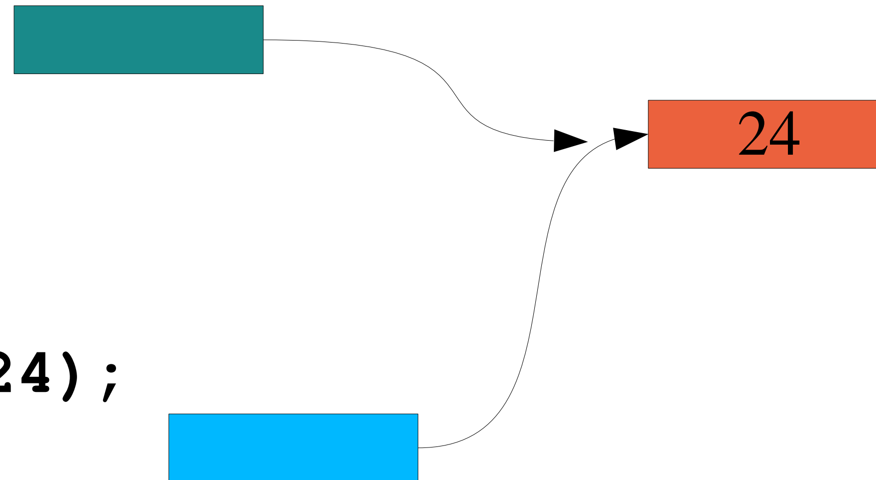
call by value

```
void foo(Student s) {

}

     :
 Student joe(24);

 foo(joe)
     :
```



call by value

```
class Student {

        IntCell *age;

    public:
                :
        Student(const Student &s) {
            age = new IntCell(s.getAge());
        }
            :
```

call by value

```
class Student {

        IntCell *age;

    public:
                :
        Student(const Student &s) {
            age = new IntCell(s.getAge());
        }
            :
```

call by value

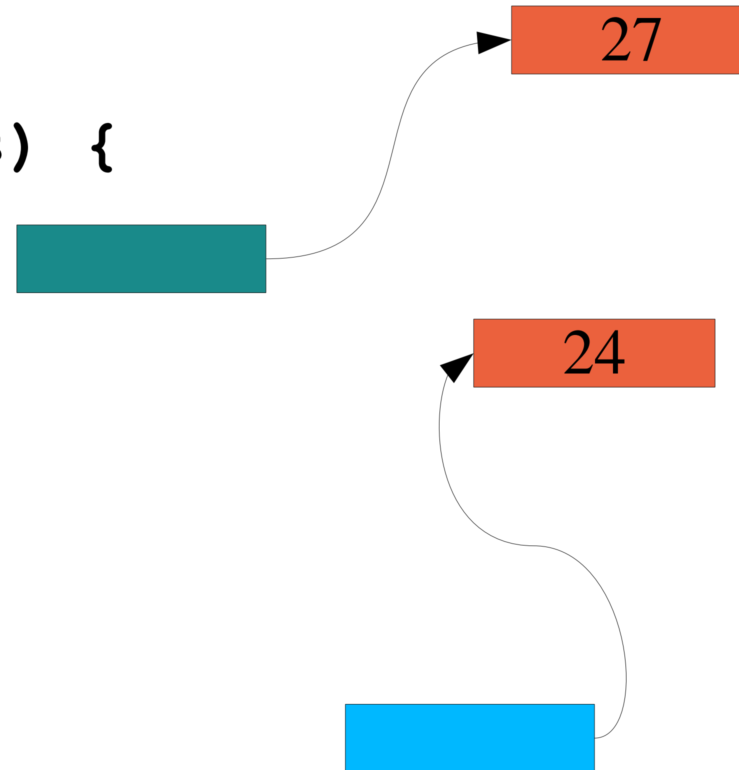
```
void foo(Student s) {


}



    :
 Student joe(24);

 foo(joe)
    :
```

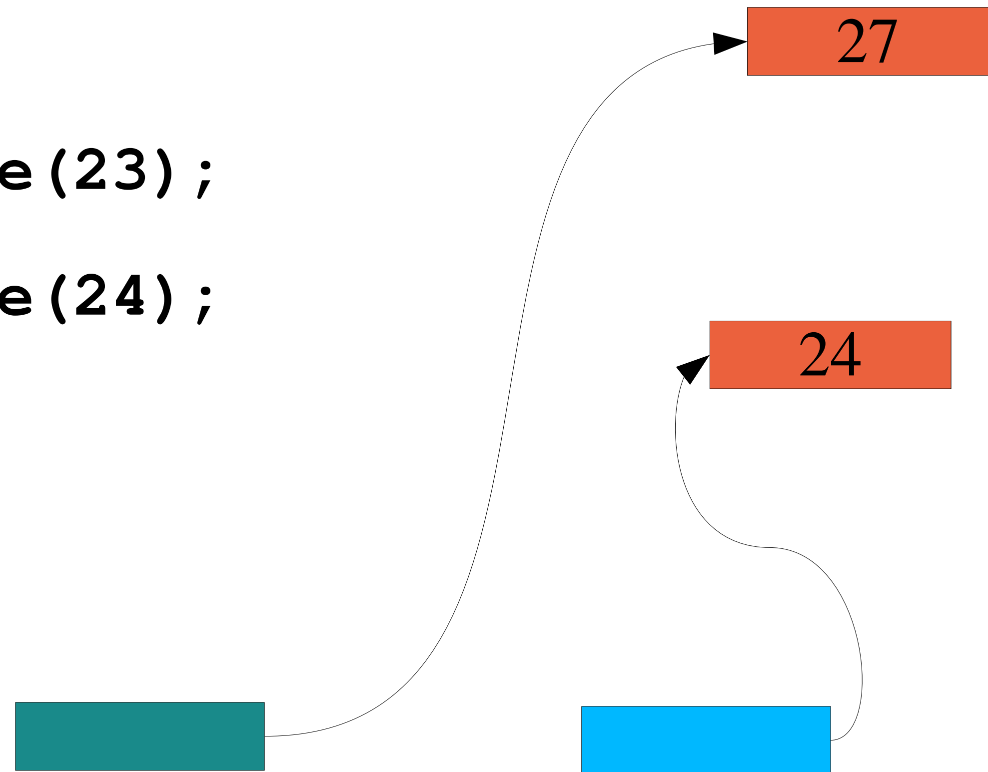


operator=

```
Student joe(23);

Studnet sue(24);

joe = sue;
```



operator=

```
const Student &operator=(const Student rhs) {
        delete age;
        age = rhs.age;
        return *this;
}

           joe = sue;
```

operator=

```
const Student &operator=(const Student rhs) {
     if (this != &rhs) {
          delete age;
          age = rhs.age;
     }
     return *this;
}
```

```
joe = sue = ellen;
```

= är höger-associativ

```
joe = (sue = ellen);
```

returnerar ett modiferat sue

Minneshantering

- Ger möjlighet att skriva mycket *effektiv* kod.
- Mycket svårt att få rätt.