Tampere University

Copying, assigning, and move

10.10.2019

Assignment and copy

- Typical in “normal” (imperative) programming
  - Especially with primitive types
  - With abstract data types, as well
- Assigning and copying objects?

Copying objects

- Automatic copying in value parameters and return values
- What is a *copy*?
- Primitive types: a copy is created by copying the original content of a memory cell

Copying objects

- Copying memory?: **No** — an object is more than its member variables
- An identical copy?: **No** — a copy is not the same as the original one
- **The *value* or *state* will be the same as that of the original one**

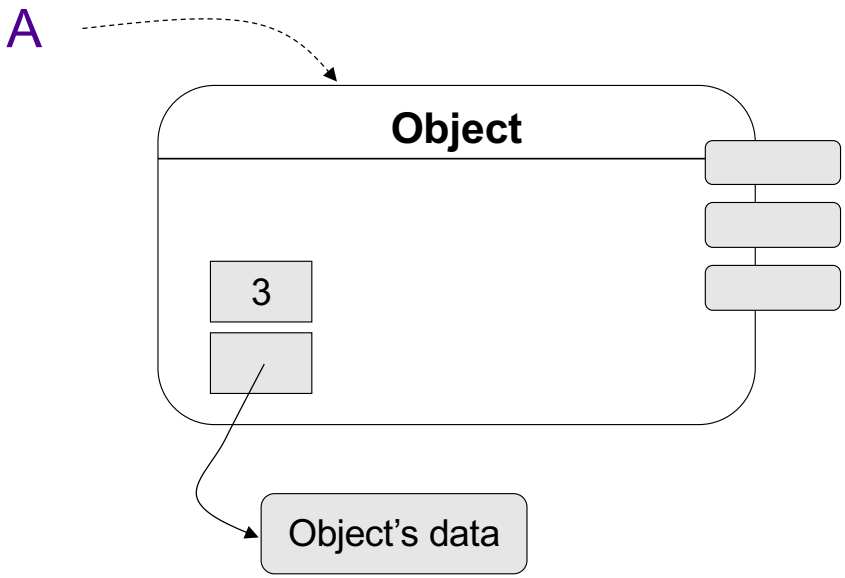
Copying objects

- The way and semantics of copying depend on the type of object!
  - Compiler cannot always create a copy automatically
  - Programmer must tell how to copy an object
  - All objects are not reasonable to be copied → preventing copying

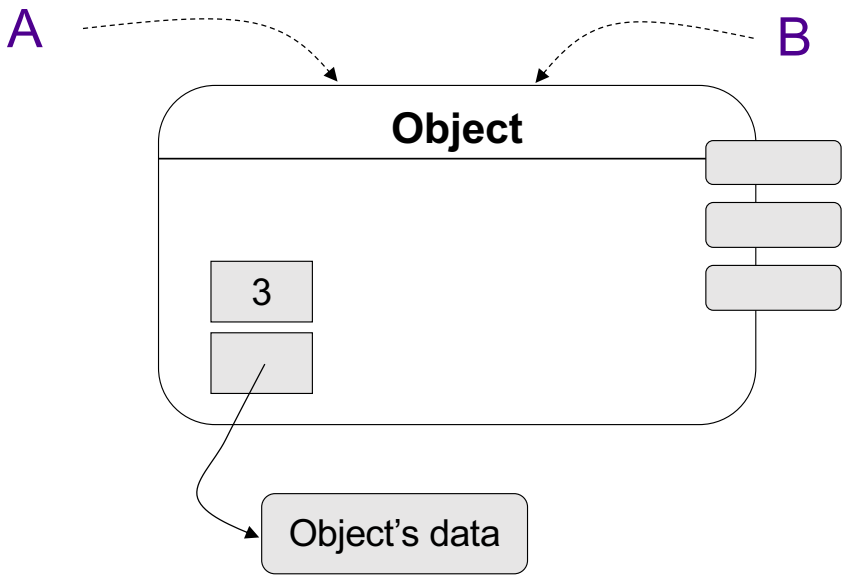
Reference copy

- **Idea:** New object is a reference to the old one
- Especially in languages with reference semantics (Smalltalk, Java)

Reference copy



Before copy

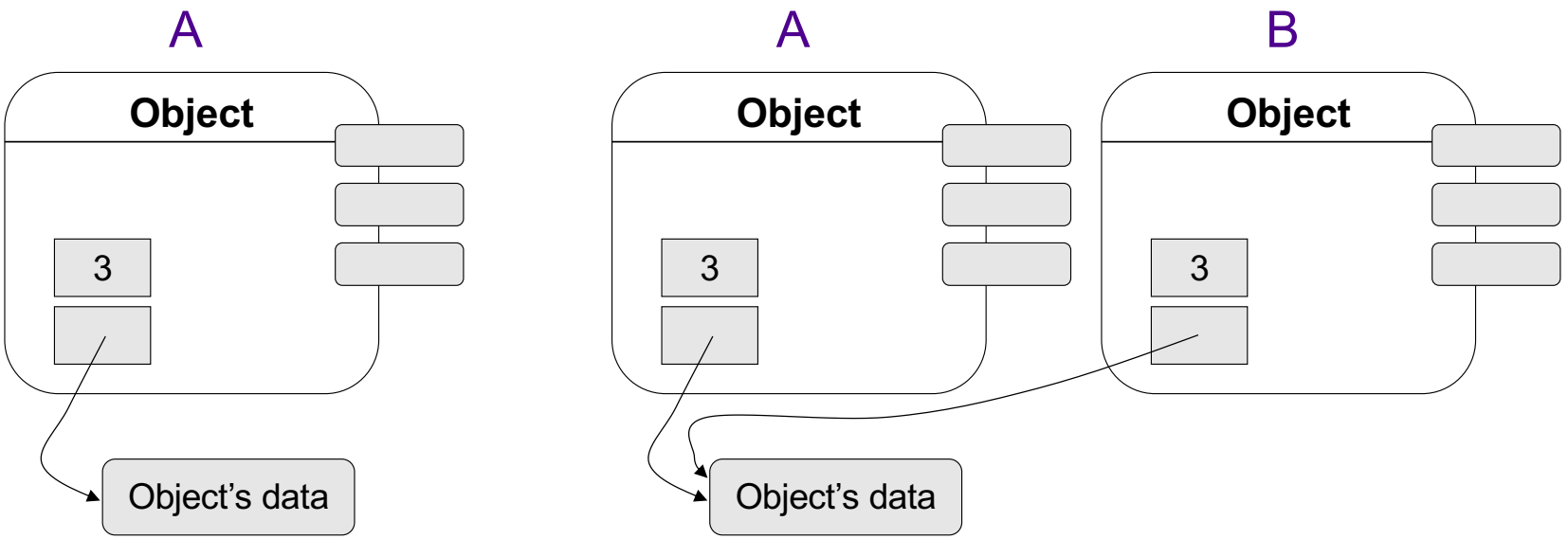


After copy

Shallow copy

- **Idea:** Copy of the object itself and its member variables, but of no data outside the object
- Easy to implement in a programming language
- **Problem:** Part of the data describing the state of the object can lie outside the object

Shallow copy



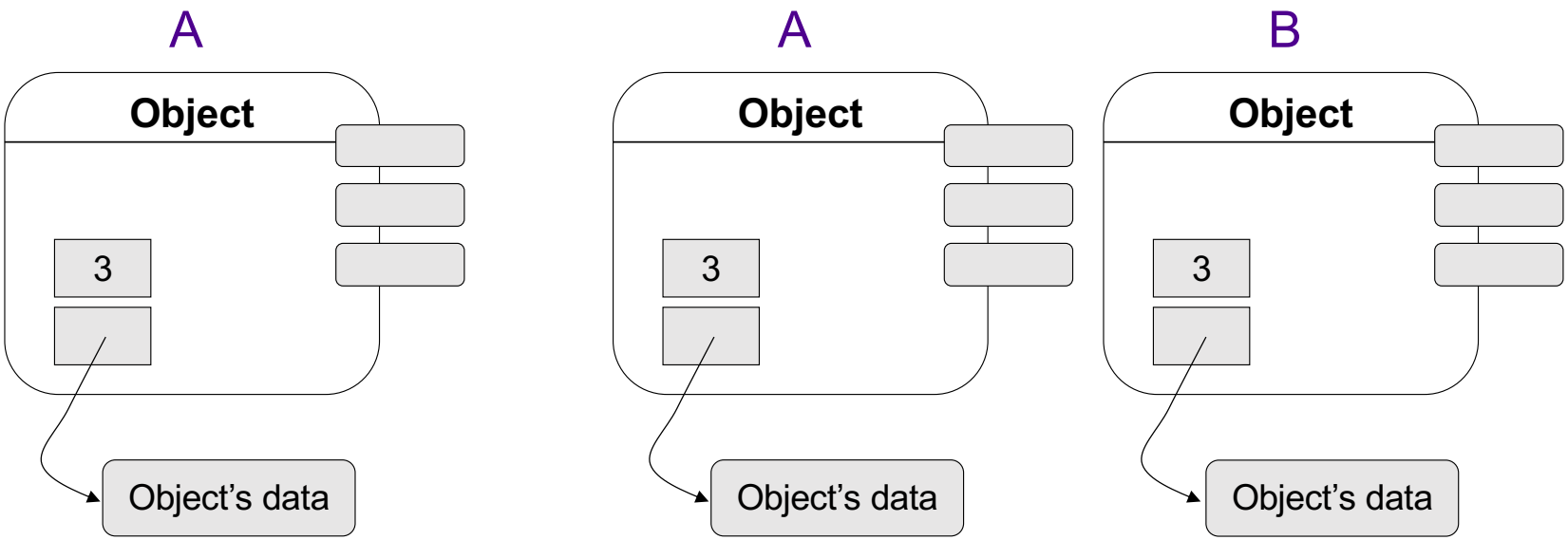
Before copy

After copy

Deep copy

- **Idea:** Copy covers also the outside data describing the state of the object → “correct” way to copy an object
- **Problem:** Which outside parts describe the state of the object?
- **Solution:** Compiler cannot conclude this → programmer writes implementation

Deep copy



Before copy

After copy

C++: copy constructor

- Object is copied with copy constructor
- Copy constructor gets a reference to the original object → enable to initialize a new object to be similar than the original one

C++: copy constructor

- Possible actions in copy constructor:
  - Initialization (copy) of member variables directly from the original ones
  - Memory allocation and copying the outside data
  - ...

Copy constructor for string

```
class MyString
{
public:
    MyString(char const* characters);
    MyString(MyString const& old); // Copy constructor
    virtual ~MyString();
private:
    unsigned long size_;
    char* characters_;
};
```

Copy constructor for string

```
MyString::MyString(MyString const& old) : size_(old.size_), characters_(0)
{
    if (size_ != 0)
    { // Allocates space for strings longer than zero
        characters_ = new char[size_ + 1];
        for (unsigned long i = 0; i != size_; ++i) {
            characters_[i] = old.characters_[i]; // Copies character by
                                                 // xscharacter
        }
        characters_[size_] = '\0'; // Ending character
    }
}
```

Inheritance and copy constructor

- Copy constructor is a constructor → **subclass copy constructor must call base class copy constructor**
- Base class copy constructor: initializes the base class part as a copy
- Subclass copy constructor: initializes the subclass part as a copy

Copy constructor: dated string

```
class DatedString : public MyString
{
public:
    DatedString(char const* characters, Date const& date);
    DatedString(DatedString const& old); // Copy constructor
    virtual ~DatedString();
private:
    Date date_;
};

// Assumes Date class to have a copy constructor
DatedString::DatedString(DatedString const& old) : MyString(old),
date_(old.date_)
{
}
```

Default copy constructor

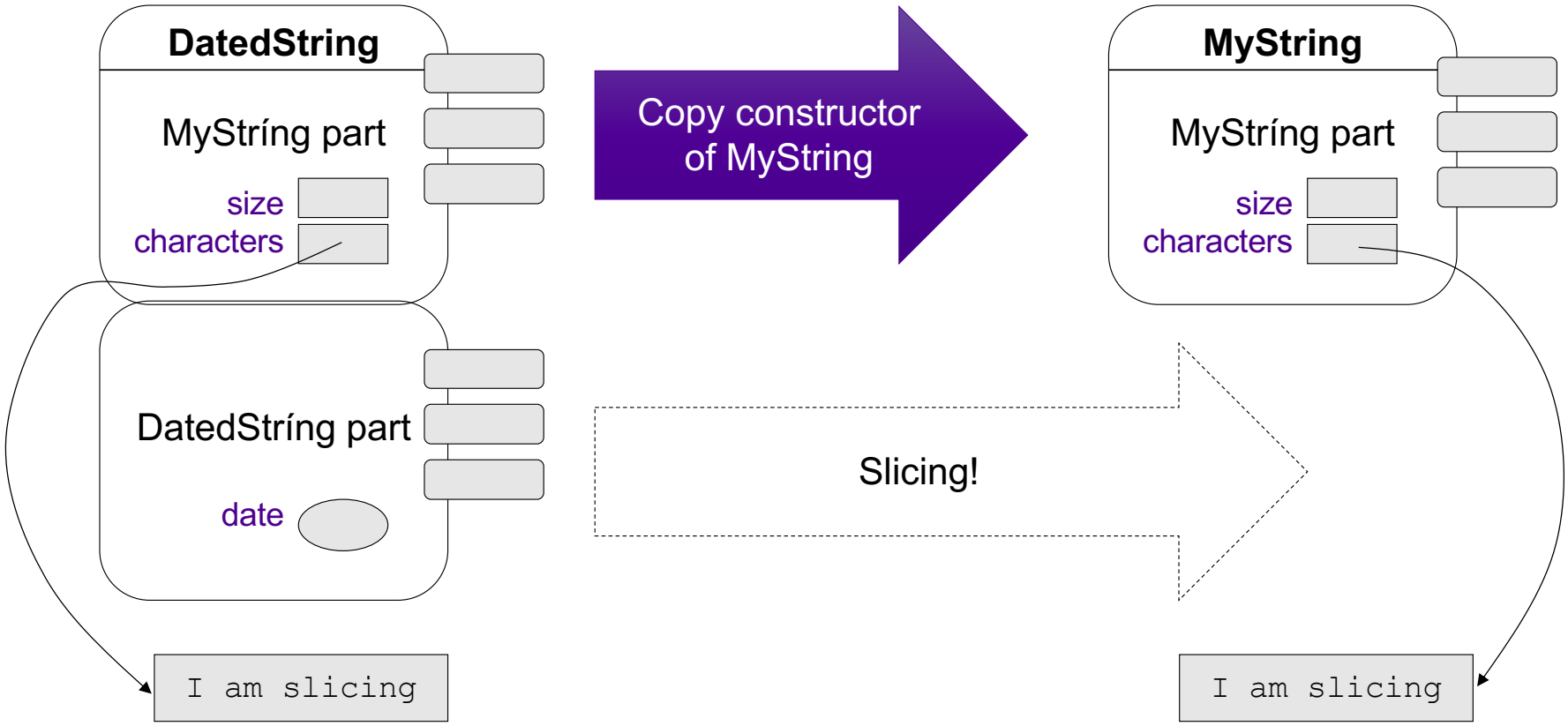
- Programmer has not written a copy constructor for a class → compiler provides a default copy constructor
- Default copy constructor copies member variables directly
  - Works in simple classes
  - Class is complex or has pointers → default copy constructor usually does not work correctly!

Preventing copying

- Sometimes copying is not meaningful → then it should be impossible
- No written copy constructor → default copy constructor → copying enabled (possibly incorrectly)
- To prevent copying add to **public** interface (C++11):


```
AClass(const AClass& value) = delete;
```

Slicing


DatedString
MyStríng part
size
characters
DatedStríng part
date
I am slicing
Copy constructor
of MyString
Slicing!
MyString
MyStríng part
size
characters
I am slicing

To avoid slicing

```
class MyString :
{
public:
    MyString(MyString const& s);
    virtual MyString* clone() const;
    …
};

MyString* MyString::clone() const
{
    return new MyString(*this);
}
```

```
class DatedString : public MyString
{
public:
    DatedString(DatedString const& s);
    virtual DatedString* clone() const;
    …
};

DatedString* DatedString::clone() const
{
    return new DatedString(*this);
}
```

To avoid slicing

```
void useCopy(MyString const& ms)
{
    MyString* copyp = ms.clone(); // Perhaps copy
                                  // of an inherited
                                  // object
    // The copy is used here
    delete copyp; copyp = 0; // Remember to destroy
}
```

Assigning objects

- Assignment and copying lead to the same result
- In assignment, there is already an object → its old value will be replaced with a new one
- Problems in assignment are related to the old value

Assigning objects

- If you need to be prepared for errors:
  - An error occurring during assignment?
  - Should the old value be returned? → difficult
- Sometimes assignment is not meaningful → to be prevented
- Sometimes assignment is not meaningful, even if copying is such

C++: assignment operator

- C++ has a specific operator for assignment (written either **operator** = or **operator=**)
- Combines actions from both destructor and copy constructor

a=b

- Calls assignment operator of **a** with **b** as a reference parameter
- Destructs the old value of **a** and replaces it with that of **b**
- Returns reference to the object itself, here **a** (enables assignment chains: a = b = c)
- “Object-like” syntax: a.**operator** =(b)

operator= with strings

```
class MyString
{
public:
    …
    MyString& operator =(MyString const& old);
private:
    unsigned long size_;
    char* characters_;
};
```

operator= with strings

```
MyString& MyString::operator =(MyString const& old) {
    if (this != &old) { // If not assigned to itself
        delete[] characters_; characters_ = 0; // Releases the old
        size_ = old.size_; // Sijoita koko
        if (size_ != 0) { // Allocates space for strings longer
                          // than zero
            characters_ = new char[size_ + 1];
            for (unsigned long i = 0; i != size_; ++i)
            { // Copies character by character
              characters_[i] = old.characters_[i];
            }
            characters_[size_] = '\0'; // Ending character
        }
    }
    return *this;
}
```

Self assignment

- Assignment **a = a** is stupid, but allowed
- Danger in normal assignment:
  - First thing is to release memory cell and other resources related to the old value
  - Next step is to allocate new memory cells and perform the actual assignment
  - In self assignment new value is the same as old value
  - → do not work in self assignment!

Self assignment

- A simple solution:
  - Self assignment should do nothing
    - → check if the action is self assignment
    - → if it is, then do nothing
    - → if not, then assign as normally
    - – check by comparing **this** reference and the reference to the parameter

Inheritance and assignment

- The same work distribution as in copying:
  - Subclass assignment operator: calls base class assignment operator, assignment of subclass part
  - Base class assignment operator: assignment of base class part
  - Recall to call base class assignment operator in subclass assignment!
- Compiler does not warn about missing call!

Subclass operator=

```
class DatedString : public MyString {
public:
    DatedString& operator =(DatedString const& old);
    // ...
private:
    Date date_;
};

DatedString& DatedString::operator =(DatedString const& old) {
    if (this != &old) { // If not assigned to itself
        MyString::operator =(old); // Base class assignment operator
        // Own (subclass) assignment, assumes Date class to have an
        // assignment operator
        date_ = old.date_;
    }
    return *this;
}
```

Default assignment operator

- Programmer has not written an assignment operator → compiler provides a default assignment operator
- Default assignment operator assigns member variables directly
  - Works in simple class
  - If a class is complicated or has pointers → default assignment operator usually does not work correctly!

Preventing assignment

- Sometimes assignment is not meaningful → then it should be impossible
- No written assignment operator → default assignment operator → assignment enabled (possibly incorrectly)
- To prevent assignment: to public interface


```
AClass& operator=(const AClass& value) = delete;
```

Slicing and assignment

- Slicing is possible also in assignment
- Subclass object is also a base class object → it can be assigned to a base class object
- Base class references and pointers enable assignment of a subclass object to another subclass object!
- Assignment slicing is possible also in other o-o languages (but they usually have no built-in assignment operator)

Move (C++11)

- “The purpose of a move constructor is to steal as many resources as it can from the original object as fast as possible, because the original does not need to have a meaningful value …”
- Faster than copying!

Move

- Move constructor:

```
AClass( AClass&& old );
```

- Move assignment:

```
AClass& operator =( AClass&& other );
```

- It is important to think if you want to move