Programming II

Genericity

Lecture Outline

- Genericity – WHY, WHEN, HOW
- Example

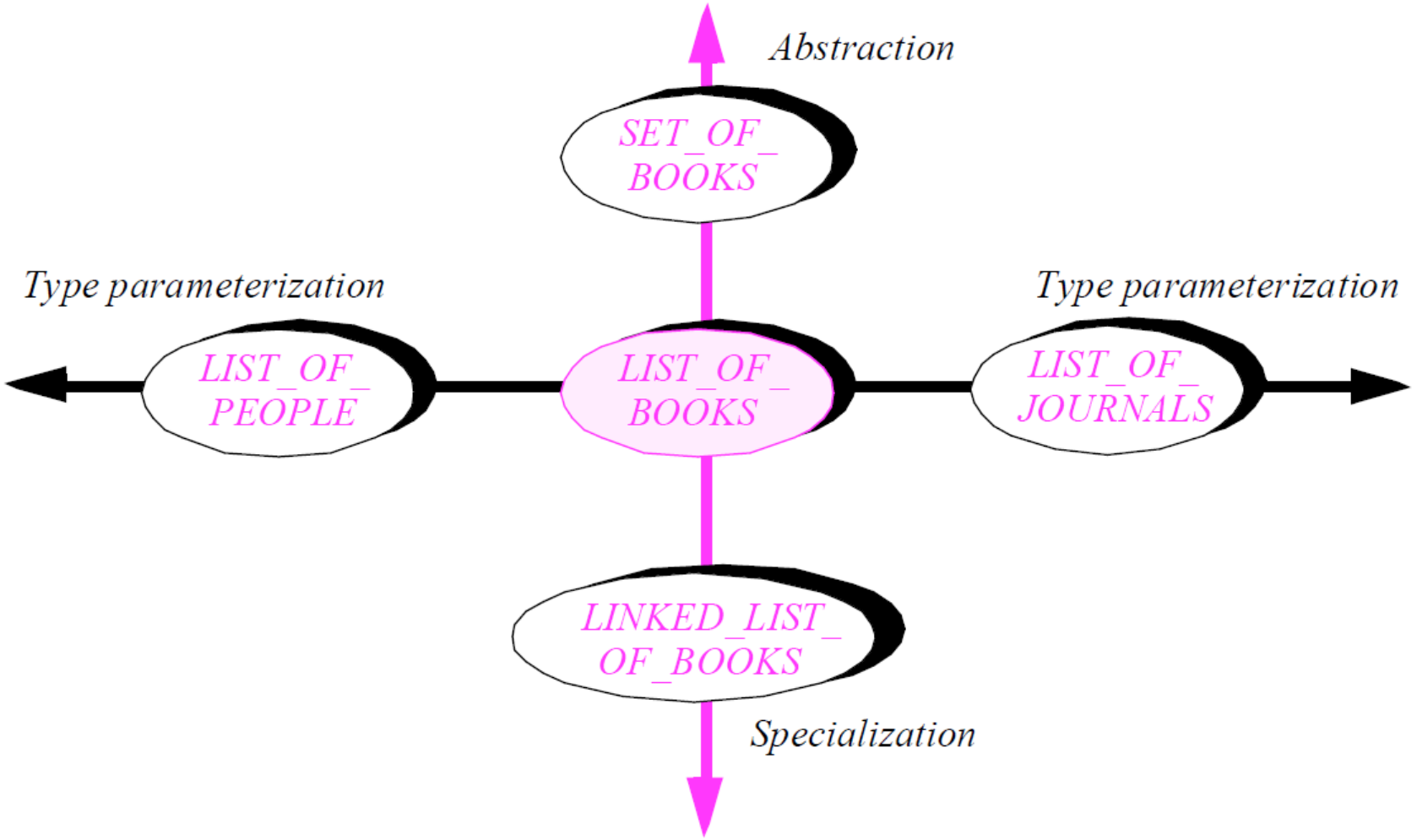
Genericity – WHY, WHEN, HOW

Genericity

- Generic – general, universal
  - opposite – special, specific
- *Genericity* is the ability of a programming language to use types like parameters in definitions.

What we solve?

- Re-usability, scalability (extendibility) and reliability.
- Inheritance provides an *abstraction* based on special cases of classes.
- Genericity provides an *abstraction* based on parameterization of classes.


Abstraction
SET_OF_
BOOKS
Type parameterization
Type parameterization
LIST_OF_
PEOPLE
LIST_OF_
BOOKS
LIST_OF_
JOURNALS
LINKED_LIST_
OF_BOOKS
Specialization

Abstraction #1

- *Containers are objects that store other objects…*
- *SET_OF_BOOKS* is a general *container* for books.
- *LIST_OF_BOOKS* is a special case of *SET_OF_BOOKS*
- *LINKED_LIST_OF_BOOKS* is a special case of *LIST_OF_BOOKS*.
- **It is abstraction projected to inheritance.**

Abstraction #2

- *LIST_OF_BOOKS*, *LIST_OF_PEOPLE*, *LIST_OF_JOURNALS* are special cases of lists storing different objects.
- All lists have the same behavior (*put, get, remove, ...*).
- **Really**?
- They work with different parameters (*Book, Person, Journal*)!!!
- **It is abstraction projected to genericity.**

Inheritance x Genericity

- Together, they solve the problem of abstraction. How do they solve?
- Inheritance: *Vertically* (hierarchy with behavior extension or change)
- Genericity: *Horizontally* (preserves the functionality, works with different types).

When to use it?

- Especially for the implementation of abstract data types (stacks, queues, lists, trees).
- It is a structure that implies storing of objects of the same type.
- However, the structures must work for any class.

How to design?

- We declare a class (type) the generic class will work with (name of a *generic parameter*).
- We write a *generic class* so that it works with the declared parameter (the type is a parameter).
- When using the generic class, we substitute the declared generic parameter with a particular class.

Operations on entities of generic types

Uses of entities of a formal generic type

The valid uses for an entity *x* whose type *G* is a formal generic parameter are the following:

G1 • Use of *x* as left-hand side in an assignment, *x* := *y*, where the right-hand side expression *y* is also of type *G*.

G2 • Use of *x* as right-hand side of an assignment *y* := *x*, where the left-hand side entity *y* is also of type *G*.

G3 • Use of *x* in a boolean expression of the form *x* = *y* or *x* /= *y*, where *y* is also of type *G*.

G4 • Use of *x* as actual argument in a routine call corresponding to a formal argument declared of type *G*, or of type *ANY*.

G5 • Use as target of a call to a feature of *ANY*.

Limitations?

- Polymorphic data structures ...
  - It is still true that we can only work with the behavior of a class that is substituted for a generic parameter.
- If the generic class uses behavior of a generic parameter, the class representing the parameter must provide this behavior!!!
  - If we only store and return instances of classes representing a generic parameter, there is no problem.

Example

Generic Class

```
template<class T>
class BOX {
private:
    T * instance;

public:
    BOX(T * i);
    T * GetInstance();
};

template<class T>
BOX<T>::BOX(T * i){
    this->instance = i;
}

template<class T>
T * BOX<T>::GetInstance(){
    return this->instance;
}
```

Class Representing Generic Parameter

```
class A {
private:
    int value;

public:
    A(int v);
    int GetValue();
};

A::A(int v){
    this->value = v;
}

int A::GetValue(){
    return this->value;
}
```

Using

```
int main() {
    A * a = new A(50);
    BOX<A> * ta = new BOX<A>(a);

    cout << ta->GetInstance()->GetValue();

    delete ta;
    delete a;

    getchar();
    return 0;
}
```


50

New Class

```
class B : public A{
public:
    B(int v);
};

B::B(int v) : A(v){
}
```

Results?

```
A * a = new A(50);
B * b = new B(100);

BOX<A> * ta = new BOX<A>(a);
BOX<B> * tb = new BOX<B>(b);

cout << ta->GetInstance()->GetValue() << endl;
cout << tb->GetInstance()->GetValue() << endl;

delete ta;
delete tb;
delete a;
delete b;
```

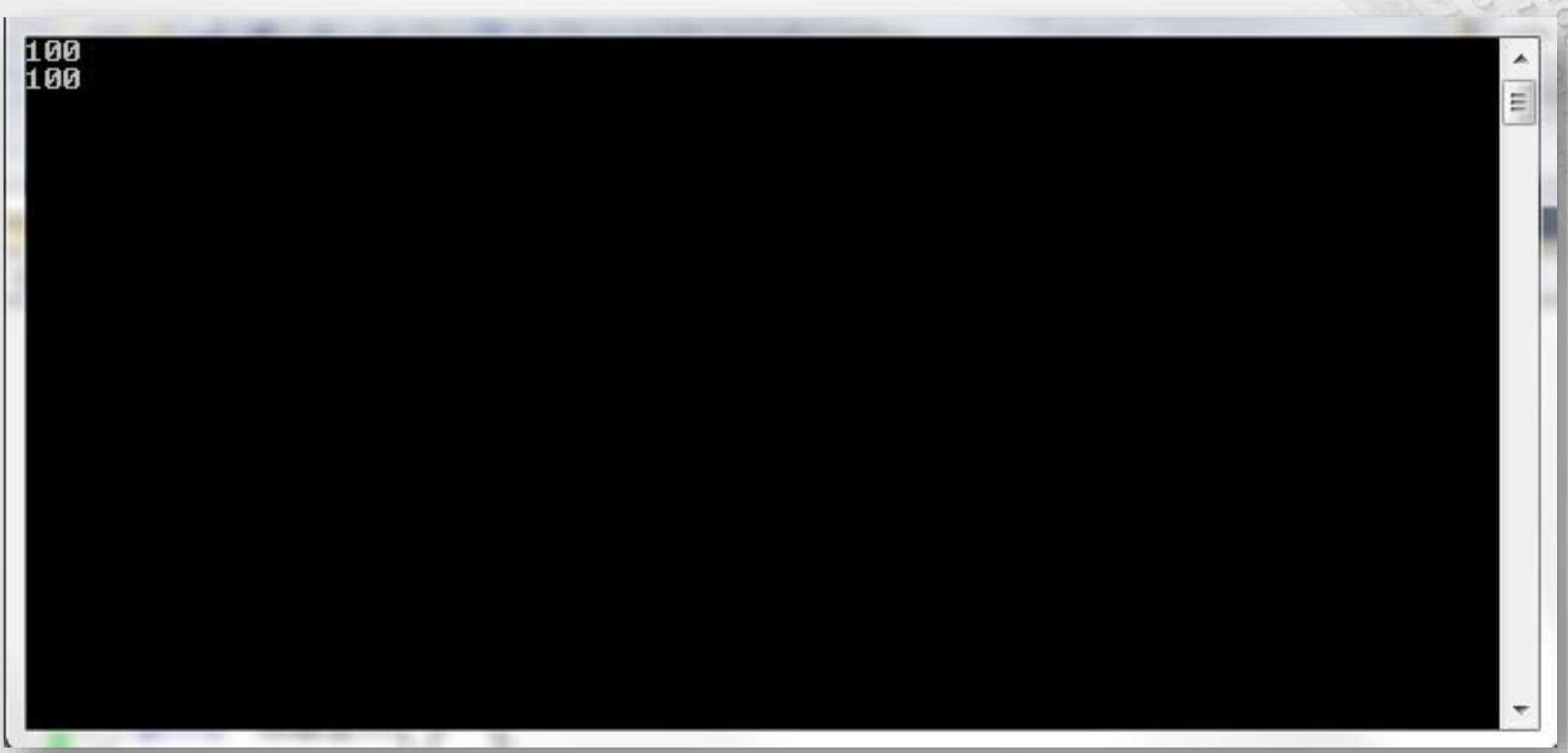
Does it work?

```
A * a = new A(50);
B * b = new B(100);

BOX<A> * ta = new BOX<A>(b);
BOX<B> * tb = new BOX<B>(b);

cout << ta->GetInstance()->GetValue() << endl;
cout << tb->GetInstance()->GetValue() << endl;

delete ta;
delete tb;
delete a;
delete b;
```


100
100

Summary

- Classes can have formal generic parameters representing the type.
- Generic classes serve as a description of structures implemented in the same way, regardless of the type they work with.
- The client of a generic classes must provide a type they will work with.
- Generic classes can work only with operations that classes in a role of the generic parameter provide.
  - Genericity has restriction based on polymorphism!!!

Sources

- Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall 1997. [317-331]

Questions

- What is genericity?
- Explain differences between inheritance-based and genericity-based abstraction.
- When to use genericity?
- Which operations can be used on entities of generic types?
- How to design classes using genericity?
- What are limitations of genericity?
- How to implement generic classes in C++?