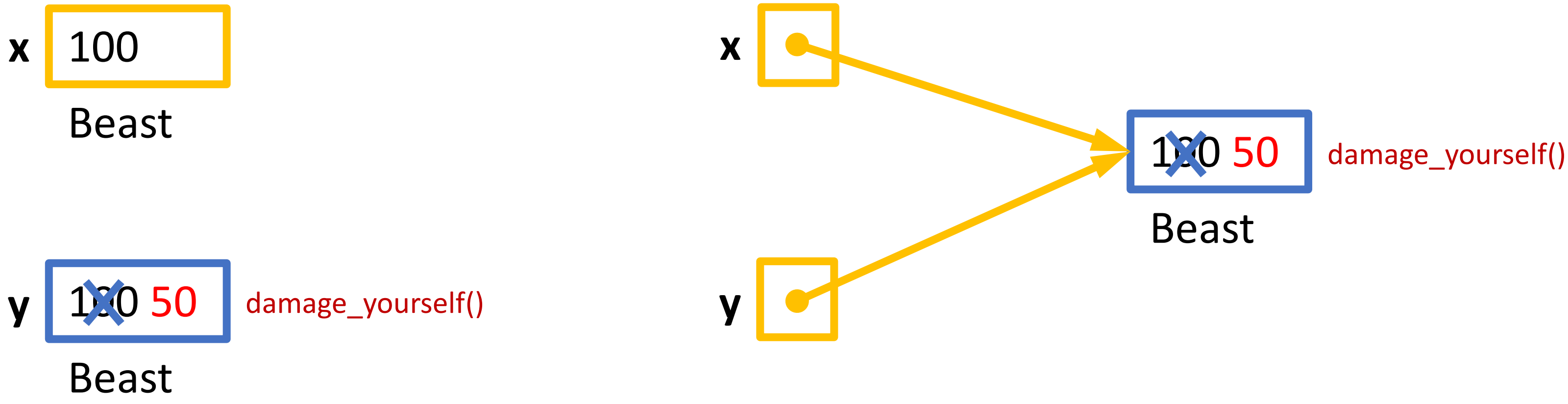
Values vs. references

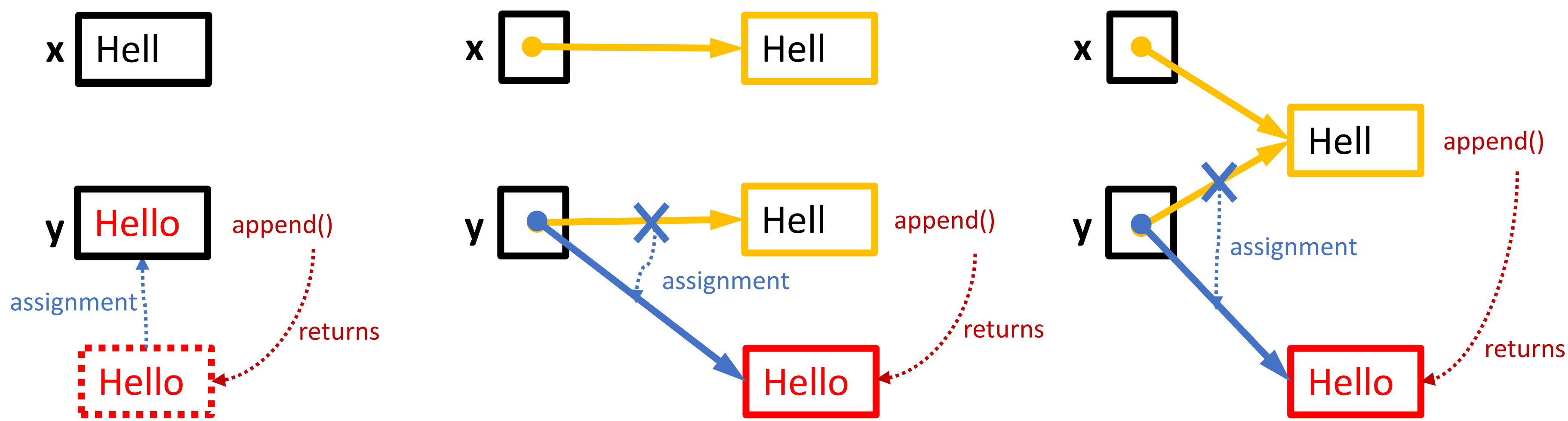
Value vs. reference types



- How does this work in your preferred language?

```
x = create_beast(100);
print(x.health);    // 100
y = x;       // does it create a copy or share a reference?
y.damage_yourself(50);    // y.health -= 50;
print(x.health);    // 100 if copy, 50 if shared
```

Immutable types

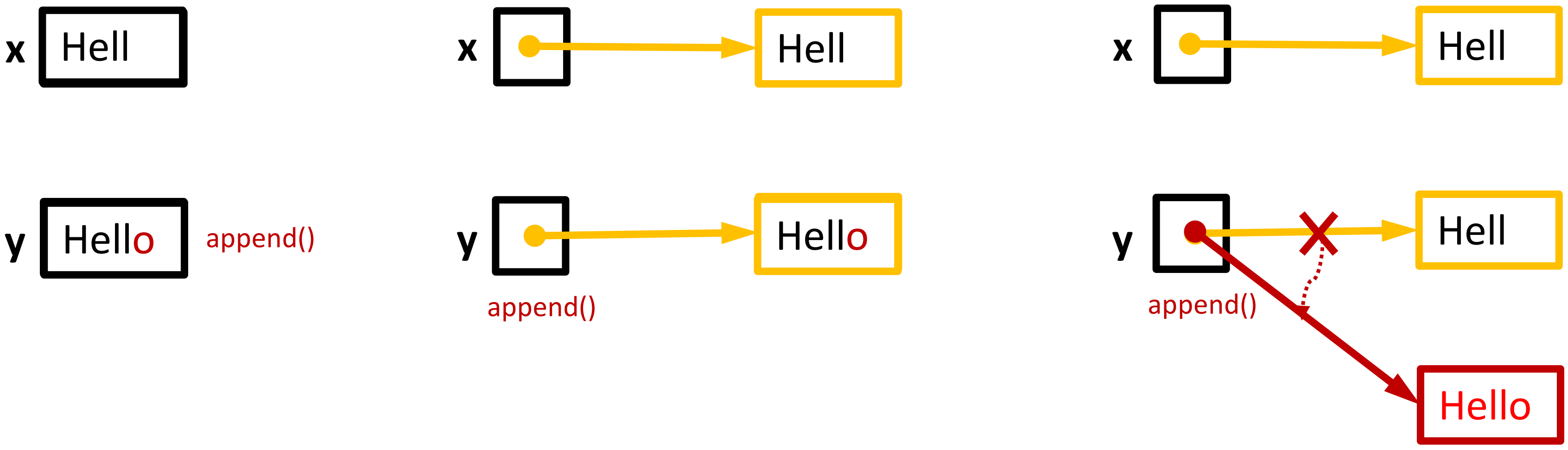


- Note: The distinction is irrelevant for immutable types
  - In many languages (not in C++), strings are immutable

```
x = "Hell";
y = x;       // is it a copy, deep copy, or shared reference?
// y.append("o");      we cannot tell because we cannot modify y in place
y = y.append("o");     // we only have this interface, returning a new object
```

  - Boxed primitive types (e.g. Integer in java) are usually immutable reference types
  - High-level languages always work with objects – numbers are immutable objects there

```
z = z + 1       // creates a new object (of type int) in python
```



- In C++, std::string is mutable

```
std::string x, y;
x = "Hell";
y = x;                   // this always copies the characters
y.append("o");     // this call modifies y but not x
```

- y.append() calls a method on the variable y (not on some distant object)
  - this call (logically) modifies y
- (in some implementations) small strings may be located inside the std::string object
- (larger) strings are stored in a dynamically allocated block owned by the std::string object
  - if the appended chars can fit inside the block, they are just appended
  - otherwise, a larger block is allocated, characters copied, old block deallocated

- How does this work in various languages?

```
x = create_beast(100);
print(x.health);    // 100
y = x;       // does it create a copy or shared reference?
y.damage_yourself(50);
print(x.health);    // 100 if copy, 50 if shared
```

- Modern languages are reference-based
  - At least when working with classes and objects
  - Modifying y will also modify x
  - Garbage collector takes care of recycling the memory
- Archaic languages sometimes give the programmer a choice
  - The behavior depends on the type of x,y ...
  - ... if x,y are "structures", assignment copies their contents
    - Records in Pascal, structs in C#, structs/classes in C++
  - ... if x,y are pointers, assignment produces two pointers to the same object
    - Which pointer is now responsible for deallocating the object?
  - Usually, different syntax is required when accessing members via pointers:

```
x^.health       (* Pascal *)
(*x).health or x->health  /* C/C++ */
```

Value vs. pointer types in C++

- When variable is the object

```
Beast x, y;
```

  - What are the values now?
    - Defined by the default constructor Beast::Beast()

```
x = create_beast(100);
print(x.health);// 100
```

  - Assignment copies x over the previous value of y

```
y = x;
y.damage_yourself(50);
print(x.health);// 100
```

  - Who will kill the Beasts?
    - The compiler takes care

- When variable is a pointer
  - Raw (C) pointers

```
Beast * x, * y;
```

    - Undefined values now!
  - C++11 smart pointers

```
std::shared_ptr< Beast> x, y;
```

    - Initialized as null pointers
  - Different syntax of member access!

```
x = create_beast(100);
print(x->health);   // 100
```

  - Assignment creates a second link to the same object

```
y = x;
y->damage_yourself(50);
print(x->health);   // 50
```

  - Who will kill the Beast?
    - Raw (C) pointers:

```
delete x; // or y, but not both!
```

    - shared_ptr takes care by counting references (run-time cost!)

Reference vs. pointer types in C++

- When variable is a reference

```
Beast & x = some_beast(100);
Beast & y2 = some_beast(200);
```

  - References must be initialized!
  - After initialization, references behave as if they were the objects
- Assignment copies the object!

```
y2 = x;
```

  - The effect of assignment is consistent with the syntax of member access

```
print(y2.health);// 100
y2.damage_yourself(50);
print(x.health);     // 100
```

- Who will kill the Beasts?
  - Someone else must own the Beasts
  - some_beast() only makes it accessible by returning a reference
    - It must not kill while the references are alive
    - That's why the name is not "create"

- When variable is a pointer

```
Beast * x, * y2; // either raw ...
std::shared_ptr< Beast> x, y2;//or smart
x = create_beast(100);
y2 = create_beast(200);
```

  - For copying contents, * is needed

```
*y2 = *x;
```

  - Member access requires ->

```
print(y2->health);   // 100
y->damage_yourself(50);
print(x->health);// 100
```

- Who will kill the Beasts?
  - Depends on the semantics of create_beast()
  - If it gives away ownership, the pointers will be responsible
    - difficult with raw (C) pointers
    - shared_ptr takes care
  - Otherwise, the creator must keep the object (or a pointer) and take care
    - It must not kill while the raw pointers are alive

Value vs. reference types in C++

- When variable is the object

```
Beast x, y;
```

  - What are the values now?
    - Defined by the default constructor Beast::Beast()

```
x = create_beast(100);
print(x.health); // 100
```

  - Assignment copies the object

```
y = x;
y.damage_yourself(50);
print(x.health); // 100
```

  - Who will kill the Beasts?
    - The compiler takes care

- When variable is a reference
  - References **must be initialized**!

```
// Beast & x, & y;
Beast & x = some_beast(100);
```

    - Initialization ensures that the reference points to something
    - The programmer can see that it is an initialization of a reference
    - References **cannot be redirected**
  - References act as the objects

```
print(x.health); // 100
```

  - Assignment copies the object

```
Beast & y2 = some_beast(200);
y2 = x;          // copy of contents
print(y2.health);// 100
```

  - For references, **initialization is different from assignment**

```
Beast & y = x;   // shared reference
y.damage_yourself(50);
print(x.health); // 50
```

  - Who will kill the Beast?
    - The references cannot kill!

```
// delete &x;
```

    - Someone else must own the Beast
    - some_beast() only makes it accessible by returning a reference

- Variable may be an object with complex behavior
  - The object may contain a pointer to another object

```
BeastWrapper x, y;
x = create_beast(100);
print(x.health);// 100
```

  - Assignment does what the author of the class wanted
    - defined by BeastWrapper::operator=

```
y = x;              // ???
y.damage_yourself(50);
print(x.health);    // ???
```

  - C/C++ programmers expect consistent behavior:
    - if members are accessed using '.', assignment shall copy contents

```
y = x;              // copy contents
y.damage_yourself(50);
print(x.health);    // 100
```

    - if members are accessed using '->', assignment shall share object

```
y = x;              // copy link
y->damage_yourself(50);
print(x->health);       // 50
```

  - Who will kill the Beast?
    - The destructor BeastWrapper::~BeastWrapper

Value vs. reference types in C++

- Variable may be an object with complex behavior
  - C/C++ programmers expect consistent behavior:
    - if members are accessed using '.', assignment shall copy contents

```
y = x;              // copy contents
y.damage_yourself(50);
print(x.health);    // 100
```

    - if members are accessed using '->', assignment shall share object

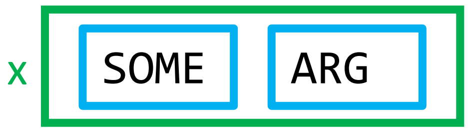
```
y = x;              // copy link
y->damage_yourself(50);
print(x->health);      // 50
```

  - If a class assigns by sharing references, it shall signalize it
    - Name the class like “BeastPointer” (e.g. std::shared_ptr)
    - Use -> for member access (define BeastPointer::operator->)
  - If a class ...
    - ... assigns by deep-copying the contents, or ...
    - ... the represented object is immutable, or ...
    - ... if it does copy-on-write ...
  - ... then it behaves like a value, therefore
    - Pretend that the class *contains* all the data (like containers do)
    - Name the class like "Beast", not "BeastWrapper"
    - Use . for member access (by implementing all the methods in the object)

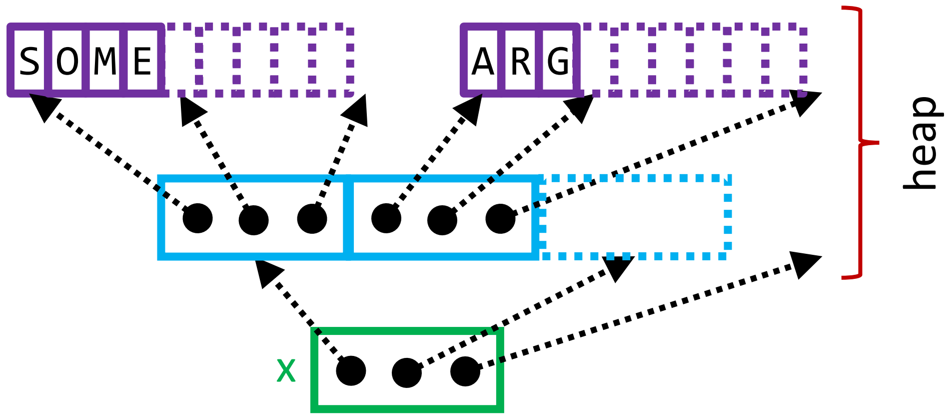
Value vs. reference types in C++

- If a class ...
  - ... assigns by deep-copying the contents, or ...
  - ... the represented object is immutable, or ...
  - ... if it does copy-on-write ...
- ... then it behaves like a value, therefore
  - Pretend that the class *contains* all the data (like containers do)
  - Name the class like "Beast", not "BeastWrapper"
  - Use . for member access (by implementing all the methods in the object)

- Example: std::vector<std::string>



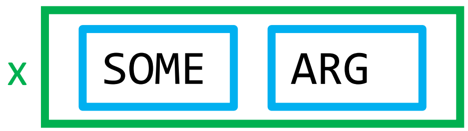
containers and strings pretend that they contain the data



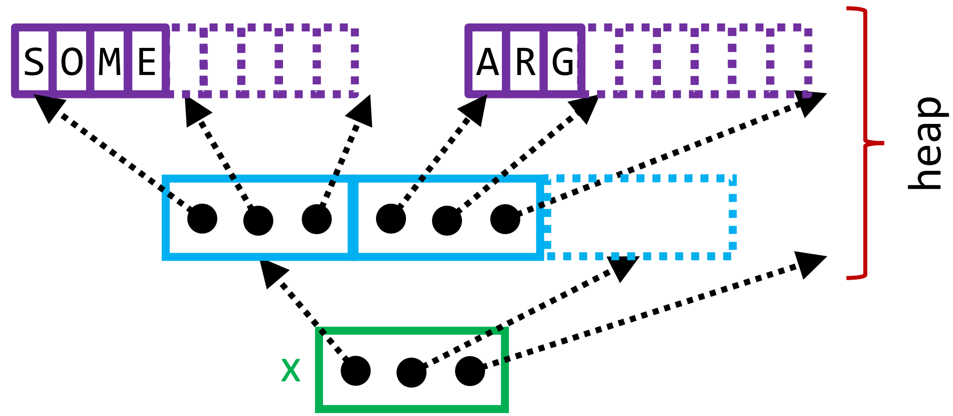
actually, the data reside elsewhere – in dynamically allocated blocks

Value vs. reference types in C++

- Example: std::vector<std::string>



containers and strings pretend that they contain the data



actually, the data reside elsewhere – in dynamically allocated blocks

- The value-like behavior is implemented in these functions:

```
string::string(const string &)               // copy-constructor
string & string::operator=(const string &)   // copy-assignment
string::~string()                            // destructor
```

  - The copy methods of string and containers perform allocation and deep copying
    - If they were not implemented explicitly, their behavior would be shallow copying of the pointers
  - The destructor performs deallocation
- Implementing these methods is now considered an **advanced** technique
  - It can be avoided in most cases (e.g. by using containers as elements)
  - Details later