C++ Parameter Passing; Copy/Move Constructors

Parameter Passing in C++

- In C, parameters are passed to functions using “call-by-value”.
- However, for C++, this will involve expensive copy operations when passing large objects.
- Hence, the preferred option in C++ while passing objects is to use “call-by-reference”.
- To understand this, let’s first define lvalue and rvalue.

Lvalues and Rvalues

- An lvalue is an expression that identifies a non-temporary object.
- An rvalue is an expression that identifies a temporary object or is a value (such as a constant) not associated with any object.
- Consider the following:
  - `IntCell i(50);`
  - `int z = x + y;`
  - `IntCell * ptr = &i;`
- `i, z, x, y, ptr` are lvalues.
- `50, x + y, &i` are rvalues.

References

- A reference defines a new name for an existing value.
- References can be defined for both lvalues and rvalues.
- lvalue reference is declared by placing an & after the type.
  - rvalue reference is declared by placing &&.

r is a reference (or alias) of i

```
IntCell i(50);
IntCell & r = i;
cout << r.read() << endl;
r.write(100);
cout << i.read() << endl;
```

Prints 50

Prints 100

Call-by-reference

- In call-by-reference, we pass a reference to the argument variable, instead of the variable itself.
  - In the function declaration, use reference variables as arguments.

Call-by-value

```
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
...
swap(a,b)
```

This won't work!

Call-by-reference

```
void swap(int & x, int & y)
{
    int temp = x;
    x = y;
    y = temp;
}
...
swap(a,b)
```

This works!

Call-by-constant-reference

- A drawback of call-by-reference is that function calls can now cause changes to the passed arguments—called “side effects”.
- To mitigate this, C++ allows call-by-constant-reference, where the function must guarantee that it does not cause changes to the passed arguments.

```
void maxCell2(IntCell & c1, IntCell & c2)
{
    if (c1.read() > c2.read())
        cout << c1.read() << endl;
    else
        cout << c2.read() << endl;
}
```

Call-by-constant-reference

- A drawback of call-by-reference is that function calls can now cause changes to the passed arguments—called "side effects".
- To mitigate this, C++ allows call-by-constant-reference, where the function must guarantee that it does not cause changes to the passed arguments.

```
void maxCell2(const IntCell & c1, const IntCell & c2)
{
    if (c1.read() > c2.read())
        cout << c1.read() << endl;
    else
        cout << c2.read() << endl;
}
```

Recommended Practice: Use call-by-constant-reference if the function does not cause changes to the passed arguments.

Copy Constructor

- A standard use-case for call-by-constant-reference is the copy constructor. Consider the following example:

```
class IntCell
{
    public:
        explicit IntCell(int initialValue=0)
        {
            storedValue = new int;
            *storedValue = initialValue;
        }
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

```
int main()
{
    IntCell c1(50);
    IntCell c2 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

100 will be printed, even though we expect 50.

Copy Constructor

```cpp
class IntCell
{
    public:
        explicit IntCell(int initialValue=0)
        {
            storedValue = new int;
            *storedValue = initialValue;
        }
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

```cpp
int main()
{
    IntCell c1(50);
    IntCell c2 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

- C++ creates a default copy constructor which will be called to copy c1 into c2.
- This constructor simply copies all the fields, thus resulting in c1 and c2 sharing the same storage. This is also called shallow copying.

Copy Constructor

While copying, we want to allocate new storage, and copy only the value.
Called Deep Copying.

```
class IntCell
{
    public:
        explicit IntCell(int initialValue=0)
        {
            storedValue = new int;
            *storedValue = initialValue;
        }
        IntCell(const IntCell & rhs)
        {
            storedValue = new int;
            *storedValue = *(rhs.storedValue);
        }
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

Copy Constructor

```
int main()
{
    IntCell c1(50);
    IntCell c2 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

Prints 50, as expected.

Return-by-reference

- Sometimes, we may also want to return the reference to an object, instead of copying it in the caller.

```
class IntCell {
    public:
        explicit IntCell(int initialValue=0)
            : storedValue(initialValue) {}
        int read() const {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```

```
IntCell maxCell(vector<IntCell> & vec)
{
    int m = 0;
    for (int i = 1; i < vec.size(); i++)
    {
        if (vec[i].read() > vec[m].read())
            m = i;
    }
    return vec[m];
}
```

```
int main() {
    vector<IntCell> v(10);
    for (int i = 0; i < v.size(); i++)
        v[i].write(i);
    IntCell m = maxCell(v);
    m.write(-1);
    cout << v[9].read() << endl;
}
```

What will be the output?

9 will be printed, even though we expect -1.

Return-by-reference

- Sometimes, we may also want to return the reference to an object, instead of copying it in the caller.

```
class IntCell {
    public:
        explicit IntCell(int initialValue=0)
            : storedValue(initialValue) {}
        int read() const {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```

```
IntCell maxCell(vector<IntCell> & vec)
{
    int m = 0;
    for (int i = 1; i < vec.size(); i++)
    {
        if (vec[i].read() > vec[m].read())
            m = i;
    }
    return vec[m];
}
```

maxCell returns-by-value, and hence the variable m contains a copy of v[9].

```
int main() {
    vector<IntCell> v(10);
    for (int i = 0; i < v.size(); i++)
        v[i].write(i);
    IntCell m = maxCell(v);
    m.write(-1);
    cout << v[9].read() << endl;
}
```

Return-by-reference

- Sometimes, we may also want to return the reference to an object, instead of copying it in the caller.

```
class IntCell {
    public:
        explicit IntCell(int initialValue=0)
            : storedValue(initialValue) {}
        int read() const {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```

```
IntCell & maxCell(vector<IntCell> & vec)
{
    int m = 0;
    for (int i = 1; i < vec.size(); i++)
    {
        if (vec[i].read() > vec[m].read())
            m = i;
    }
    return vec[m];
}
```

```
int main() {
    vector<IntCell> v(10);
    for (int i = 0; i < v.size(); i++)
        v[i].write(i);
    IntCell & m = maxCell(v);
    m.write(-1);
    cout << v[9].read() << endl;
}
```

Now, maxCell returns-by-reference, and hence m becomes an alias of v[9]

Copy Assignment

- A standard use-case for return-by-reference is the copy assignment operator. Consider the following example:

```
class IntCell{
    public:
        explicit IntCell(int initialValue=0)
        {
            storedValue = new int;
            *storedValue = initialValue;
        }
        IntCell(const IntCell & rhs)
        {
            storedValue = new int;
            *storedValue = *(rhs.storedValue);
        }

        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

```
int main()
{
    IntCell c1(50);
    IntCell c2;
    c2 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

100 will be printed, even though we expect 50.

Didn't we solve this problem already?😫

Copy Assignment

```
int main()
{
    IntCell c1(50);
    IntCell c2 = c1;
    IntCell c3;
    c3 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

This triggers the copy constructor

This triggers the copy assignment operator

The default copy assignment operator created by C++ simply copies all the field values from RHS to LHS, thus resulting in shallow copying for IntCell

Copy Assignment

```
class IntCell{
    public:
        explicit IntCell(int initialValue=0)
        {
            storedValue = new int;
            *storedValue = initialValue;
        }
        IntCell(const IntCell & rhs)
        {
            storedValue = new int;
            *storedValue = *(rhs.storedValue);
        }

        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

```
int main()
{
    IntCell c1(50);
    IntCell c2;
    c2 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

We want to copy the stored value in c1 to c2, not the location

Copy Assignment

```
class IntCell {
    public:
        explicit IntCell(int initialValue=0) {
            storedValue = new int;
            *storedValue = initialValue;
        }
        IntCell(const IntCell & rhs) {
            storedValue = new int;
            *storedValue = *(rhs.storedValue);
        }
        IntCell & operator=(const IntCell & rhs){
            *storedValue = *(rhs.storedValue);
            return *this;
        }
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

Copy assignment operator definition

Triggered by lhs=rhs
Calls lhs.operator=(rhs)

Copy Assignment

```
class IntCell {
    public:
        explicit IntCell(int initialValue=0) {
            storedValue = new int;
            *storedValue = initialValue;
        }
        IntCell(const IntCell & rhs) {
            storedValue = new int;
            *storedValue = *(rhs.storedValue);
        }
        IntCell & operator=(const IntCell & rhs){
            *storedValue = *(rhs.storedValue);
            return *this;
        }
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;
    private:
        int * storedValue;
};
```

```
int main()
{
    IntCell c1(50);
    IntCell c2;
    c2 = c1;
    c2.write(100);
    cout << c1.read() << endl;
}
```

50 will be printed, as expected.

Homework: Implement a swap function for IntCell which swaps two IntCells. Use copy assignment in swap function, and experiment with different implementations of the copy assignment operator

Destructor

- A destructor is called (automatically) when an object goes out of scope/is destroyed.
- A destructor is helpful when some cleanup is required at the end of life of an object.
  - Closing an open file
  - Releasing a lock
  - malloc-free, new-delete

Destructor-Example

```
class IntCell {
    public:
        explicit IntCell(int initialValue=0) {
            storedValue = new int;
            *storedValue = initialValue;
        }
        IntCell(const IntCell & rhs) {
            storedValue = new int;
            *storedValue = *(rhs.storedValue);
        }
        IntCell & operator=(const IntCell & rhs){
            *storedValue = *(rhs.storedValue);
            return *this;
        }
        ~IntCell()  {delete storedValue;}
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

Destructor

Destructor-Example

```
class IntCell {
  public:
    explicit IntCell(int initialValue=0) {
      storedValue = new int;
      *storedValue = initialValue;
    }
    IntCell(const IntCell & rhs) {
      storedValue = new int;
      *storedValue = *(rhs.storedValue);
    }
    IntCell & operator=(const IntCell & rhs){
      *storedValue = *(rhs.storedValue);
      return *this;
    }
    ~IntCell(){
      cout << "deallocating " << *storedValue << endl;
      delete storedValue;}
    int read() {return *storedValue;}
    void write(int x) {*storedValue = x;}
  private:
    int * storedValue;
};
```

```
int main()
{
    IntCell c2;
    {
        IntCell c1(50);
        c2 = IntCell(100);
    }
    cout << c2.read() << endl;

}
```

Prints:
deallocating 100
deallocating 50
100
deallocating 100

Move Constructor and Move Assignment

- In C++11 onwards, we can also define move constructor and move assignment operator.
  - These are called when copying temporary objects, i.e. rvalues.
- Example: for IntCell class, the following statements will trigger move constructor and assignment respectively:
  - `IntCell c = IntCell(50);`
  - `IntCell c; c = IntCell(50);`

Move Constructor and Move Assignment: Example

```
class IntCell {
    public:
        . . .
        }
        IntCell (IntCell && rhs) : storedValue(rhs.storedValue) {
            rhs.storedValue = nullptr;
        }
        IntCell & operator=(IntCell && rhs){
           std::swap(storedValue, rhs.storedValue);
           return *this;
        }
        ~IntCell()  {delete storedValue;}
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

Move Constructor

Move Constructor and Move Assignment: Example

```
class IntCell {
    public:
        . . .
        }
        IntCell (IntCell && rhs) : storedValue(rhs.storedValue) {
            rhs.storedValue = nullptr;
        }
        IntCell & operator=(IntCell && rhs){
           std::swap(storedValue, rhs.storedValue);
           return *this;
        }
        ~IntCell()  {delete storedValue;}
        int read() {return *storedValue;}
        void write(int x) {*storedValue = x;}
    private:
        int * storedValue;
};
```

Move Assignment

Note that copy constructor/operator can also be used for copying rvalues. However, defining specialised move operators can be more efficient.