operator overloading

operators are just like functions in the sense x+y; is same as operator+(x,y); which means that they can also be overloaded like functions. when adding two double values, the double function will be used and if two other data types are added, their respective function will be used. All operators can be overloaded except conditional (?:), sizeof, scope (::), member selector (.), pointer member selector (.*), typeid, and the casting operators. we can use operator overloads to implement operators when one of the operators is program/ user defined, we can even slightly change functionality but this all is only possible when both operands are not in built data types.

only limitations is that, these overloads are not future proof because std::string is a user defined data type today could become a in built data type tomorrow making the overloading code obsolete and we can also not change number of operands & their precedence criterias.

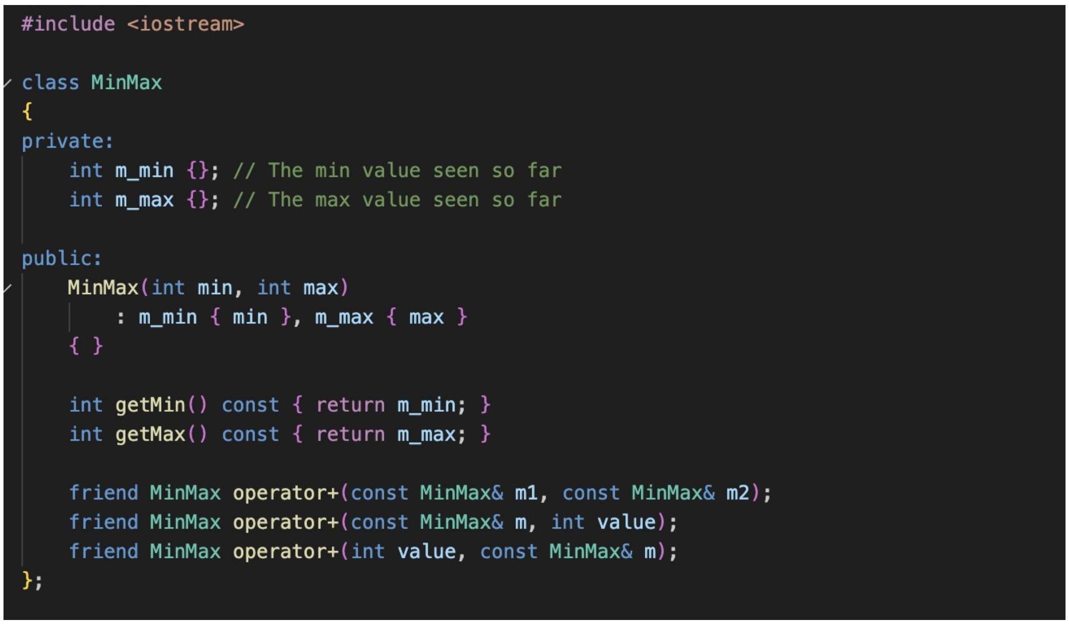
you will find plenty of useful functionality to overload for your custom classes! You can overload the + operator to concatenate your program-defined string class, or add two Fraction class objects together. You can overload the << operator to make it easy to print your class to the screen (or a file). You can overload the equality operator (==) to compare two class objects. This makes operator overloading one of the most useful features in C++ -- simply because it allows you to work with your classes in a more intuitive way.

overloading using friend functions

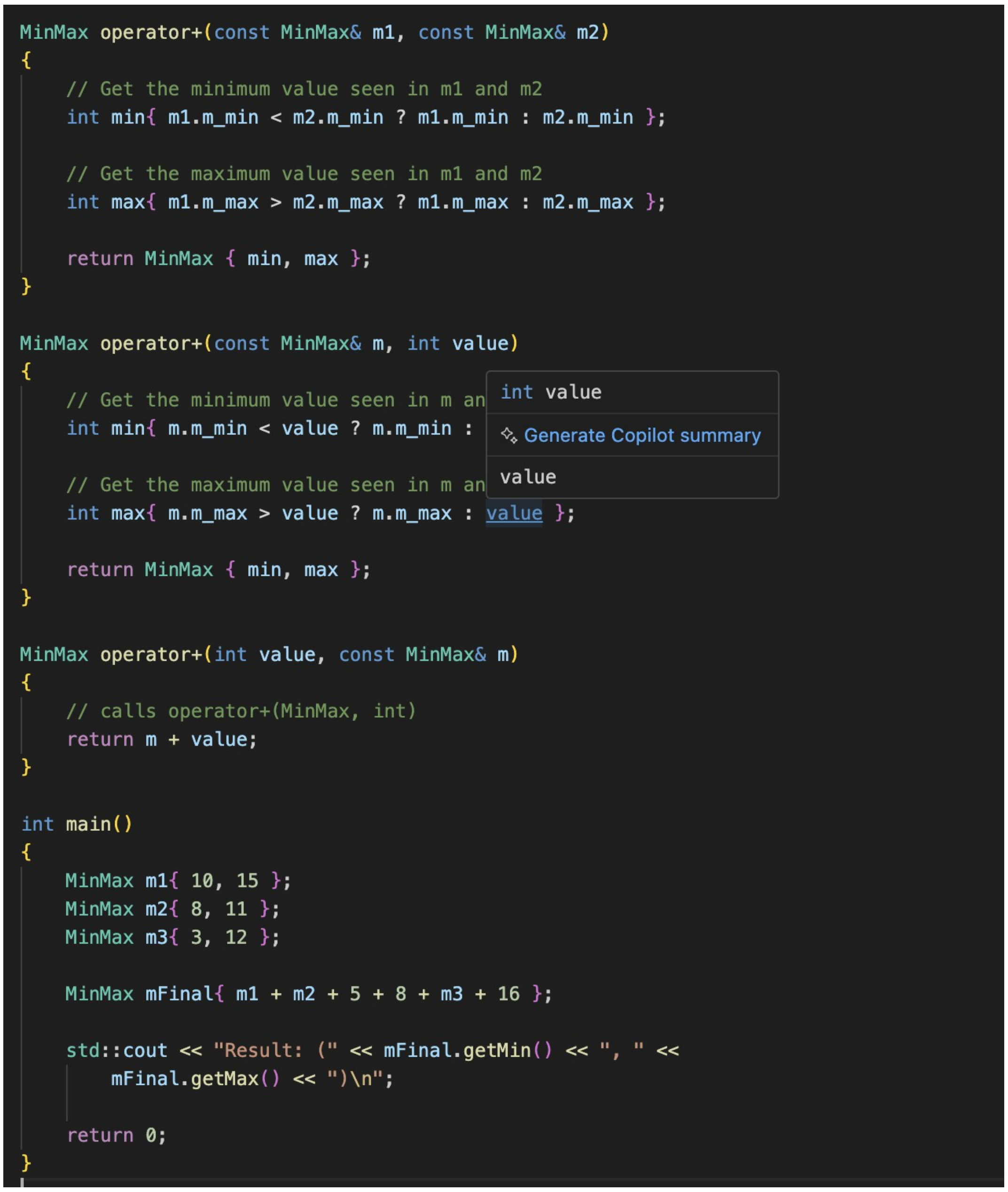
```cpp
#include <iostream>

class Cents
{
private:
    int m_cents {};

public:
    Cents(int cents) : m_cents{ cents } { }

    // add Cents + Cents using a friend function
    friend Cents operator+(const Cents& c1, const Cents& c2);

    int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents& c1, const Cents& c2)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return c1.m_cents + c2.m_cents;
}

int main()
{
    Cents cents1{ 6 };
    Cents cents2{ 8 };
    Cents centsSum{ cents1 + cents2 };
    std::cout << "I have " << centsSum.getCents() << " cents.\n";

    return 0;
}
```

In overloading functions including one operand as built in and other as user defined, we would need to write two functions for example (obj,int) and (int,obj)

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents {};

public:
    explicit Cents(int cents) : m_cents{ cents } { }

    // add Cents + int using a friend function
    friend Cents operator+(const Cents& c1, int value);

    // add int + Cents using a friend function
    friend Cents operator+(int value, const Cents& c1);


    int getCents() const { return m_cents; }
};

// note: this function is not a member function!
Cents operator+(const Cents& c1, int value)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents { c1.m_cents + value };
}

// note: this function is not a member function!
Cents operator+(int value, const Cents& c1)
{
    // use the Cents constructor and operator+(int, int)
    // we can access m_cents directly because this is a friend function
    return Cents { c1.m_cents + value };
}

int main()
{
    Cents c1{ Cents{ 4 } + 6 };
    Cents c2{ 6 + Cents{ 4 } };

    std::cout << "I have " << c1.getCents() << " cents.\n";
    std::cout << "I have " << c2.getCents() << " cents.\n";

    return 0;
}
```



```cpp
#include <iostream>

class MinMax
{
private:
    int m_min {}; // The min value seen so far
    int m_max {}; // The max value seen so far

public:
    MinMax(int min, int max)
        : m_min { min }, m_max { max }
    { }

    int getMin() const { return m_min; }
    int getMax() const { return m_max; }

    friend MinMax operator+(const MinMax& m1, const MinMax& m2);
    friend MinMax operator+(const MinMax& m, int value);
    friend MinMax operator+(int value, const MinMax& m);
};
```



```
MinMax operator+(const MinMax& m1, const MinMax& m2)
{
    // Get the minimum value seen in m1 and m2
    int min{ m1.m_min < m2.m_min ? m1.m_min : m2.m_min };

    // Get the maximum value seen in m1 and m2
    int max{ m1.m_max > m2.m_max ? m1.m_max : m2.m_max };

    return MinMax { min, max };
}

MinMax operator+(const MinMax& m, int value)
{
    // Get the minimum value seen in m an
    int min{ m.m_min < value ? m.m_min :

    // Get the maximum value seen in m an
    int max{ m.m_max > value ? m.m_max : value };

    return MinMax { min, max };
}

MinMax operator+(int value, const MinMax& m)
{
    // calls operator+(MinMax, int)
    return m + value;
}

int main()
{
    MinMax m1{ 10, 15 };
    MinMax m2{ 8, 11 };
    MinMax m3{ 3, 12 };

    MinMax mFinal{ m1 + m2 + 5 + 8 + m3 + 16 };

    std::cout << "Result: (" << mFinal.getMin() << ", " <<
        mFinal.getMax() << ")\n";

    return 0;
}
```

since operator+ evaluates from left to right, m1 + m2 evaluates first and so on. this expression evaluates as “MinMax mFinal = (((((m1 + m2) + 5) + 8) + m3) + 16)”, with each successive operation returning a MinMax object that becomes the left-hand operand for the following operaton.

as you can see we defined operator+(int, MinMax) **by calling operator+(MinMax, int)** (which produces the same result). This allows us to reduce the implementation of operator+(int, MinMax) to a single line, making our code easier to maintain by minimizing redundancy and making the function simpler to understand.

overloading using normal functions

```cpp
#include <iostream>

class Cents
{
private:
  int m_cents{};

public:
  Cents(int cents)
    : m_cents{ cents }
  {}

  int getCents() const { return m_cents; }
};

// note: this function is not a member function nor a friend function!
Cents operator+(const Cents& c1, const Cents& c2)
{
  // use the Cents constructor and operator+(int, int)
  // we don't need direct access to private members here
  return Cents{ c1.getCents() + c2.getCents() };
}

int main()
{
  Cents cents1{ 6 };
  Cents cents2{ 8 };
  Cents centsSum{ cents1 + cents2 };
  std::cout << "I have " << centsSum.getCents() << " cents.\n";

  return 0;
}
```

we can use the getter functions to overload operators without using friendly functions, in this method; if the class is in a header file then the overload function needs to be declared after the whole class definition as a function prototype declaration because then the normal function code can be written in the main file for use.

normal functions should be preferred over friend functions; especially if implemented with already existing member functions and to avoid creating extra friend functions to access the object.

overloading using member functions

Converting a friend overloaded operator to a member overloaded operator is easy:

1) The overloaded operator is defined as a member instead of a friend (Cents::operator+ instead of friend operator+)
2) The left parameter is removed, because that parameter now becomes the implicit *this object.
3) Inside the function body, all references to the left parameter can be removed (e.g. cents.m_cents becomes m_cents, which implicitly references the *this object).

```cpp
#include <iostream>

class Cents
{
private:
    int m_cents {};

public:
    Cents(int cents)
        : m_cents { cents } { }

    // Overload Cents + int
    Cents operator+(int value) const;

    int getCents() const { return m_cents; }
};

// note: this function is a member function!
// the cents parameter in the friend version is now the implicit *this parameter
Cents Cents::operator+ (int value) const
{
    return Cents { m_cents + value };
}
```

not all operators can be overloaded as a friend function

assignment (=), subscript ([]), function call (()), and member selection (->) operators must be overloaded as member functions, because the language requires them to be.

not all operators can be overloaded as a member function

we cannot overload operator<< as a member function because the overloaded operator must be added as a member of the left operand which in this case,the left operand is an object of type std::ostream. std::ostream is fixed as part of the standard library and the class declaration can't be modified to add the overload as a member function of std::ostream. this necessitates that operator<< be overloaded as a **normal function (preferred)** or a friend.

similarly, although we can overload operator+(Cents, int) as a member function (as we did above), we can't overload operator+(int, Cents) as a member function, because int isn't a class we can add members to, typically, we won't be able to use a member overload if the left operand is either not a class (e.g. int), or it is a class that we can't modify (e.g. std::ostream).

The following rules of thumb can help you determine which form is best for a given situation:

- If you're overloading assignment (=), subscript ([]), function call (()), or member selection (->), do so as a member function

- If you're overloading a unary operator, do so as a member function

- If you're overloading a binary operator that does not modify its left operand (e.g. operator+), do so as a normal function (preferred) or friend function.

- If you're overloading a binary operator that modifies its left operand, but you can't add members to the class definition of the left operand (e.g. operator<<, which has a left operand of type ostream), do so as a normal function (preferred) or friend function.

- If you're overloading a binary operator that modifies its left operand (e.g. operator+=), and you can modify the definition of the left operand, do so as a member function.

overloading <<

suppose we wanted to std::cout all or some data members of an object with **<<**, we would need to overload the operator clearly.

```
#include <iostream>

class Point
{
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0)
      : m_x{x}, m_y{y}, m_z{z}
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Point& point);
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')'; // actual output done here

    return out; // return std::ostream so we can chain calls to operator<<
}

int main()
{
    const Point point1 { 2.0, 3.0, 4.0 };

    std::cout << point1 << '\n';

    return 0;
}
```

the left operand is a std::cout object and the right operand is a point class object.
The most notable difference is that std::cout has become parameter out (which will be a **reference** to std::cout when the function is called) this is done because unlike normal operator overloads where the return type can simply be the data type returned, we cannot return std::ostream by value because it is disallowed in c++ to be copied, we must return as a reference to make it work, especially because this also makes chain output possible.

(If our operator<< returned void instead. When the compiler evaluates std::cout << point << '\n', due to the precedence/associativity rules, it evaluates this expression as (std::cout << point) << '\n';. std::cout << point would call our void-returning overloaded operator<< function, which returns void. Then the partially evaluated expression becomes: void << '\n';, which makes no sense! By returning the out parameter as the return type instead, (std::cout << point) returns std::cout. Then our partially evaluated expression becomes: std::cout << '\n';, which then gets evaluated itself! Any time we want our overloaded binary operators to be chainable in such a manner, the left operand should be returned (by reference). Returning the left-hand parameter by reference is okay in this case -- since the left-hand parameter was passed in by the calling function, it must still exist when the called function returns. Therefore, we don't have to worry about referencing something that will go out of scope and get destroyed when the operator returns.)

overloading >>

```
#include <iostream>

class Point
{
private:
    double m_x{};
    double m_y{};
    double m_z{};

public:
    Point(double x=0.0, double y=0.0, double z=0.0)
      : m_x{x}, m_y{y}, m_z{z}
    {
    }

    friend std::ostream& operator<< (std::ostream& out, const Point& point);
    friend std::istream& operator>> (std::istream& out, Point& point);
};

std::ostream& operator<< (std::ostream& out, const Point& point)
{
    // Since operator<< is a friend of the Point class, we can access Point's members directly.
    out << "Point(" << point.m_x << ", " << point.m_y << ", " << point.m_z << ')';

    return out;
}

// note that point must be non-const so we can modify the object
std::istream& operator>> (std::istream& in, Point& point)
{
    // This version subject to partial extraction issues (see below)
    in >> point.m_x >> point.m_y >> point.m_z;

    return in;
}

int main()
{
    std::cout << "Enter a point: ";

    Point point{ 1.0, 2.0, 3.0 }; // non-zero test data
    std::cin >> point;

    std::cout << "You entered: " << point << '\n';

    return 0;
}
```

Assuming the user enters `4.0 5.6 7.26` as input, the program produces the following result:

```
You entered: Point(4, 5.6, 7.26)
```

Now let's see what happens when the user enters `4.0b 5.6 7.26` as input (notice the `b` after the `4.0`):

```
You entered: Point(4, 0, 3)
```

Our point is now a weird hybrid consisting of one value from the user's input (`4.0`), one value that has been zero-initialized (`0.0`), and one value that was untouched by the input function (`3.0`). That's… not great!

When we're extracting a single value, there are only two possible outcomes: the extraction fails, or it is successful. However, when we're extracting more than one value as part of an input operation, things get a bit more complicated. The above implementation of operator>> can result in a partial extraction. And this is exactly what we're seeing with input 4.0b 5.6 7.26. The extraction to x_y successfully extracts 4.0 from the user's input, leaving b 5.6 7.26 in the input stream. The extraction to m_y fails to extract b, so m_y is copy-assigned the value 0.0 and the input stream is set to failure mode. Since we haven't cleared failure mode, the extraction to m_z aborts immediately, and the value that m_z had before the extraction attempt remains (3.0).

A **transactional operation** must either completely succeed or completely fail -- no partial successes or failures are allowed. This is sometimes known as “all or nothing”. If a failure occurs at any point during the transaction, prior changes made by the operation must be undone.

```
// note that point must be non-const so we can modify the object
// note that this implementation is a non-friend
std::istream& operator>> (std::istream& in, Point& point)
{
    double x{};
    double y{};
    double z{};

    if (in >> x >> y >> z)      // if all extractions succeeded
        point = Point{x, y, z}; // overwrite our existing point

    return in;
}
```

While the above `operator>>` prevents partial extractions, it is inconsistent with how `operator>>` works for fundamental types. When extraction to an object with a fundamental type fails, the object isn’t left unaltered -- it is copy assigned the value `0` (this ensures the object has some consistent value in case it wasn’t initialized before the extraction attempt). Therefore, for consistency, you may wish to have a failed extraction reset the object to its default state (at least in cases where such a thing exists).

```
// note that point must be non-const so we can modify the object
// note that this implementation is a non-friend
std::istream& operator>> (std::istream& in, Point& point)
{
    double x{};
    double y{};
    double z{};

    in >> x >> y >> z;
    point = in ? Point{x, y, z} : Point{};

    return in;
}
```

we can also add our own rules in accepting valid inputs for values that we may deem semantically invalid like a denominator being zero.

```
std::istream& operator>> (std::istream& in, Point& point)
{
    double x{};
    double y{};
    double z{};

    in >> x >> y >> z;
    if (x < 0.0 || y < 0.0 || z < 0.0)       // if any extractable input is negative
        in.setstate(std::ios_base::failbit); // set failure mode manually
    point = in ? Point{x, y, z} : Point{};

    return in;
}
```

resource material for operator overloads

https://www.learncpp.com/cpp-tutorial/overloading-the-io-operators/

https://www.learncpp.com/cpp-tutorial/overloading-unary-operators/

https://www.learncpp.com/cpp-tutorial/overloading-the-comparison-operators/

https://www.learncpp.com/cpp-tutorial/overloading-the-increment-and-decrement-operators/

https://www.learncpp.com/cpp-tutorial/overloading-the-subscript-operator/

https://www.learncpp.com/cpp-tutorial/overloading-the-parenthesis-operator/

https://www.learncpp.com/cpp-tutorial/overloading-typecasts/

overloading assignment operator

the difference between use case for a copy constructor & overloaded assignment operator

- If a new object has to be created before the copying can occur, the copy constructor is used (note: this includes passing or returning objects by value).
- If a new object does not have to be created before the copying can occur, the assignment operator is used.

```
// A simplistic implementation of operator= (see better implementation below)
Fraction& Fraction::operator= (const Fraction& fraction)
{
    // do the copy
    m_numerator = fraction.m_numerator;
    m_denominator = fraction.m_denominator;

    // return the existing object so we can chain this operator
    return *this;
}

int main()
{
    Fraction fiveThirds { 5, 3 };
    Fraction f;
    f = fiveThirds; // calls overloaded assignment
    std::cout << f;

    return 0;
}
```

there is a implicit copy assignment operator provided by c++ which can be deleted as usual

```
    // Overloaded assignment
    Fraction& operator= (const Fraction& fraction) = delete; // no copies through assignment!
```

issue with self assignment

if we did something like fiveThirds = fiveThirds, it will work fine for the most part and all the members will be assigned accordingly but in cases where an operator needs to assign memory dynamically, it gets quite dangerous because the object we wanted to copy to another object gets deleted to prevent memory leak before we can do the copy process and the this* pointer is now just dangling pointing to garbage which ends up getting copied to the new object. a better way to deal with this issue is copy and swap idiom & self-assignment guards.

shallow vs deep copy

copy assignment operator uses shallow copy which means the new variables are simply pointing to the address of the older variable; this can cause lots of problems like in cases of dynamically allocated objects but even if lets say

```
#include <iostream>

int main()
{
    MyString hello{ "Hello, world!" };
    {
        MyString copy{ hello }; // use default copy constructor
    } // copy is a local variable, so it gets destroyed here.  The destructor deletes copy's string, which leaves
hello with a dangling pointer

    std::cout << hello.getString() << '\n'; // this will have undefined behavior

    return 0;
}
```

the copy variable gets destroyed here but its pointing to the address of hello variable, making it dangling even if its scope has not ended yet.

```
// assumes m_data is initialized
void MyString::deepCopy(const MyString& source)
{
    // first we need to deallocate any value that this string is holding!
    delete[] m_data;

    // because m_length is not a pointer, we can shallow copy it
    m_length = source.m_length;

    // m_data is a pointer, so we need to deep copy it if it is non-null
    if (source.m_data)
    {
        // allocate memory for our copy
        m_data = new char[m_length];

        // do the copy
        for (int i{ 0 }; i < m_length; ++i)
            m_data[i] = source.m_data[i];
    }
    else
        m_data = nullptr;
}

// Copy constructor
MyString::MyString(const MyString& source)
{
    deepCopy(source);
}
```

```
// Assignment operator
MyString& MyString::operator=(const MyString& source)
{
    // check for self-assignment
    if (this != &source)
    {
        // now do the deep copy
        deepCopy(source);
    }

    return *this;
}
```

- We added a self-assignment check.
- We return *this so we can chain the assignment operator.
- We need to explicitly deallocate any value that the string is already holding (so we don’t have a memory leak when m_data is reallocated later) and create new memory to save the new data to have separate scopes.

class template

```cpp
#ifndef ARRAY_H
#define ARRAY_H

#include <cassert>

template <typename T> // added
class Array
{
private:
    int m_length{};
    T* m_data{}; // changed type to T

public:

    Array(int length)
    {
        assert(length > 0);
        m_data = new T[length]{}; // allocated an array of objects of type T
        m_length = length;
    }

    Array(const Array&) = delete;
    Array& operator=(const Array&) = delete;

    ~Array()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

    // templated operator[] function defined below
    T& operator[](int index); // now returns a T&

    int getLength() const { return m_length; }
};

// member functions defined outside the class need their own template declaration
template <typename T>
T& Array<T>::operator[](int index) // now returns a T&
{
    assert(index >= 0 && index < m_length);
    return m_data[index];
}

#endif
```

template member methods must be declared in same file as class template to avoid linking error

we can specialise function templates and class templates to ensure that there exists a template for all data types but if we used a specific data type like int or double then we can write extra code for that specific case. https://www.learncpp.com/cpp-tutorial/function-template-specialization/

```
template <typename T>
void print(T value) {
    std::cout << "General template: " << value << '\n';
}

template <>
void print<int>(int value) {
    std::cout << "Specialized for int: " << value << '\n';
}
```

```
template <typename T>
class Printer {
public:
    void print() {
        std::cout << "General printer\n";
    }
};

// Specialization for int
template <>
class Printer<int> {
public:
    void print() {
        std::cout << "Integer printer\n";
    }
};
```

https://www.learncpp.com/cpp-tutorial/class-template-specialization/

we can even partially specialise that if one of the data types is present; we can utilise the specialised function/class code.

https://www.learncpp.com/cpp-tutorial/partial-template-specialization/

https://www.learncpp.com/cpp-tutorial/partial-template-specialization-for-pointers/

```cpp
template <typename T1, typename T2>
class Pair {
public:
    void display() {
        std::cout << "Generic Pair\n";
    }
};

// Specialize when second type is int
template <typename T1>
class Pair<T1, int> {
public:
    void display() {
        std::cout << "Partial specialization where T2 = int\n";
    }
};
```

smart pointers

smart pointers is a composition class that is designed to manage dynamically allocated memory and ensure that memory gets deleted when the smart pointer object goes out of scope. (Relatedly, built-in pointers are sometimes called “dumb pointers” because they can’t clean up after themselves).

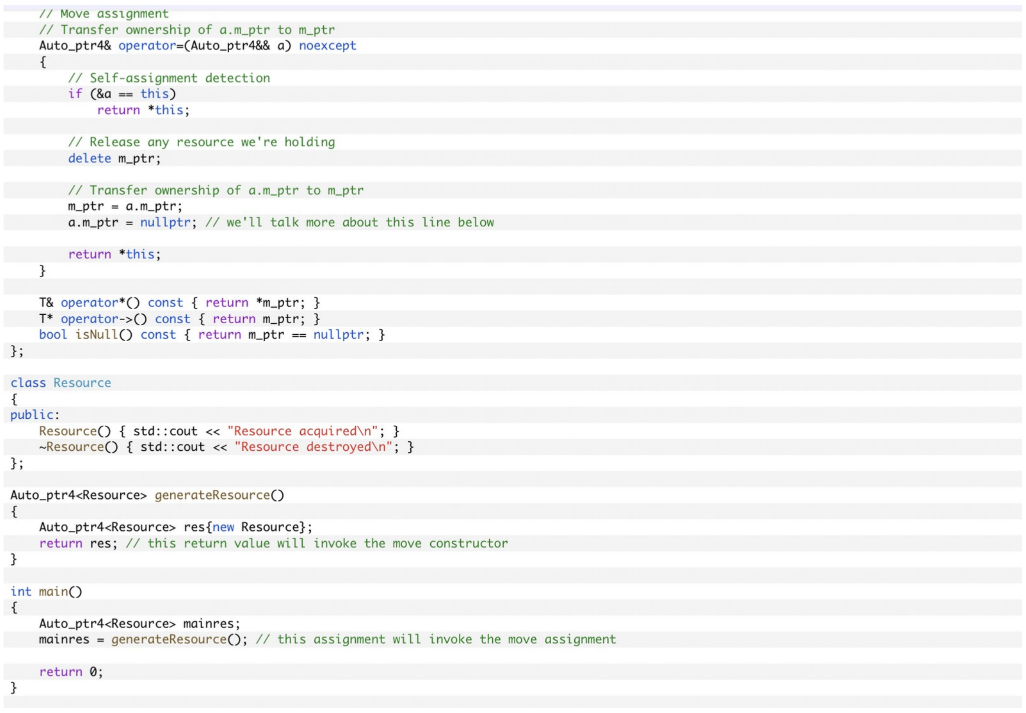
```cpp
#include <iostream>

template <typename T>
class Auto_ptr1
{
    T* m_ptr {};
public:
    // Pass in a pointer to "own" via the constructor
    Auto_ptr1(T* ptr=nullptr)
        :m_ptr(ptr)
    {
    }

    // The destructor will make sure it gets deallocated
    ~Auto_ptr1()
    {
        delete m_ptr;
    }

    // Overload dereference and operator-> so we can use Auto_ptr1 like m_ptr.
    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
};

// A sample class to prove the above works
class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

int main()
{
    Auto_ptr1<Resource> res(new Resource()); // Note the allocation of memory here

        // ... but no explicit delete needed

    // Also note that we use <Resource>, not <Resource*>
        // This is because we've defined m_ptr to have type T* (not T)

    return 0;
} // res goes out of scope here, and destroys the allocated Resource for us
```

only problem with this class is that the implicit copy constructors and assignment overloads will perform shadow copies with references pointing to the same address, we can either delete both these implicit things to prevent them from happening or just overload them ourselves explicitly to perform deep copy

move semantics

move semantics means transferring the ownership of object rather than making the copy, so we will first create the new object variable referencing the address and point the first object's pointer to null. we use to have **std::auto_ptr** which standard implementation of a smart pointer in c++ using move semantics but it had a lot of problems like if its passed by value in a function then; it will copied to a function parameter and then get destroyed so if you want to access the argument later; well its dangling now & dynamic memory is not properly deallocated because it uses non-array delete for its delete operation. **move constructors** & **move assignment operator** were introduced in C++11. copy constructors and copy assignment operator makes the copy of one object to another, move transfers the ownership of resource and deletes the previous ownership record.

```
template<typename T>
class Auto_ptr4
{
    T* m_ptr {};
public:
    Auto_ptr4(T* ptr = nullptr)
        : m_ptr { ptr }
    {
    }

    ~Auto_ptr4()
    {
        delete m_ptr;
    }

    // Copy constructor
    // Do deep copy of a.m_ptr to m_ptr
    Auto_ptr4(const Auto_ptr4& a)
    {
        m_ptr = new T;
        *m_ptr = *a.m_ptr;
    }

    // Move constructor
    // Transfer ownership of a.m_ptr to m_ptr
    Auto_ptr4(Auto_ptr4&& a) noexcept
        : m_ptr { a.m_ptr }
    {
        a.m_ptr = nullptr; // we'll talk more about this line below
    }

    // Copy assignment
    // Do deep copy of a.m_ptr to m_ptr
    Auto_ptr4& operator=(const Auto_ptr4& a)
    {
        // Self-assignment detection
        if (&a == this)
            return *this;

        // Release any resource we're holding
        delete m_ptr;

        // Copy the resource
        m_ptr = new T;
        *m_ptr = *a.m_ptr;

        return *this;
    }
```



```
    // Move assignment
    // Transfer ownership of a.m_ptr to m_ptr
    Auto_ptr4& operator=(Auto_ptr4&& a) noexcept
    {
        // Self-assignment detection
        if (&a == this)
            return *this;

        // Release any resource we're holding
        delete m_ptr;

        // Transfer ownership of a.m_ptr to m_ptr
        m_ptr = a.m_ptr;
        a.m_ptr = nullptr; // we'll talk more about this line below

        return *this;
    }

    T& operator*() const { return *m_ptr; }
    T* operator->() const { return m_ptr; }
    bool isNull() const { return m_ptr == nullptr; }
};

class Resource
{
public:
    Resource() { std::cout << "Resource acquired\n"; }
    ~Resource() { std::cout << "Resource destroyed\n"; }
};

Auto_ptr4<Resource> generateResource()
{
    Auto_ptr4<Resource> res{new Resource};
    return res; // this return value will invoke the move constructor
}

int main()
{
    Auto_ptr4<Resource> mainres;
    mainres = generateResource(); // this assignment will invoke the move assignment

    return 0;
}
```

the && here is for r-value references (temporary objects); which solves all issues because r-value objects are going to be destroyed anyways so we don't have to be worried about them later.
move constructors and assignments are provided implicitly and they can be deleted like usual.
move semantics has some issues but they are advanced problems and there are more subtopics

https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/
https://www.learncpp.com/cpp-tutorial/stdmove/
https://www.learncpp.com/cpp-tutorial/stdunique_ptr/
https://www.learncpp.com/cpp-tutorial/circular-dependency-issues-with-stdshared_ptr-and-stdweak_ptr/