OOP C++ part2

Moatasem Elsayed

Content(long session 😁)

operator overloading

- sort

-functor

- post/pre

-conversion

-friend operator

-explicit

- explicit operator

- copy constructor

- -fno-elide-constructors

-Operator =

- value category

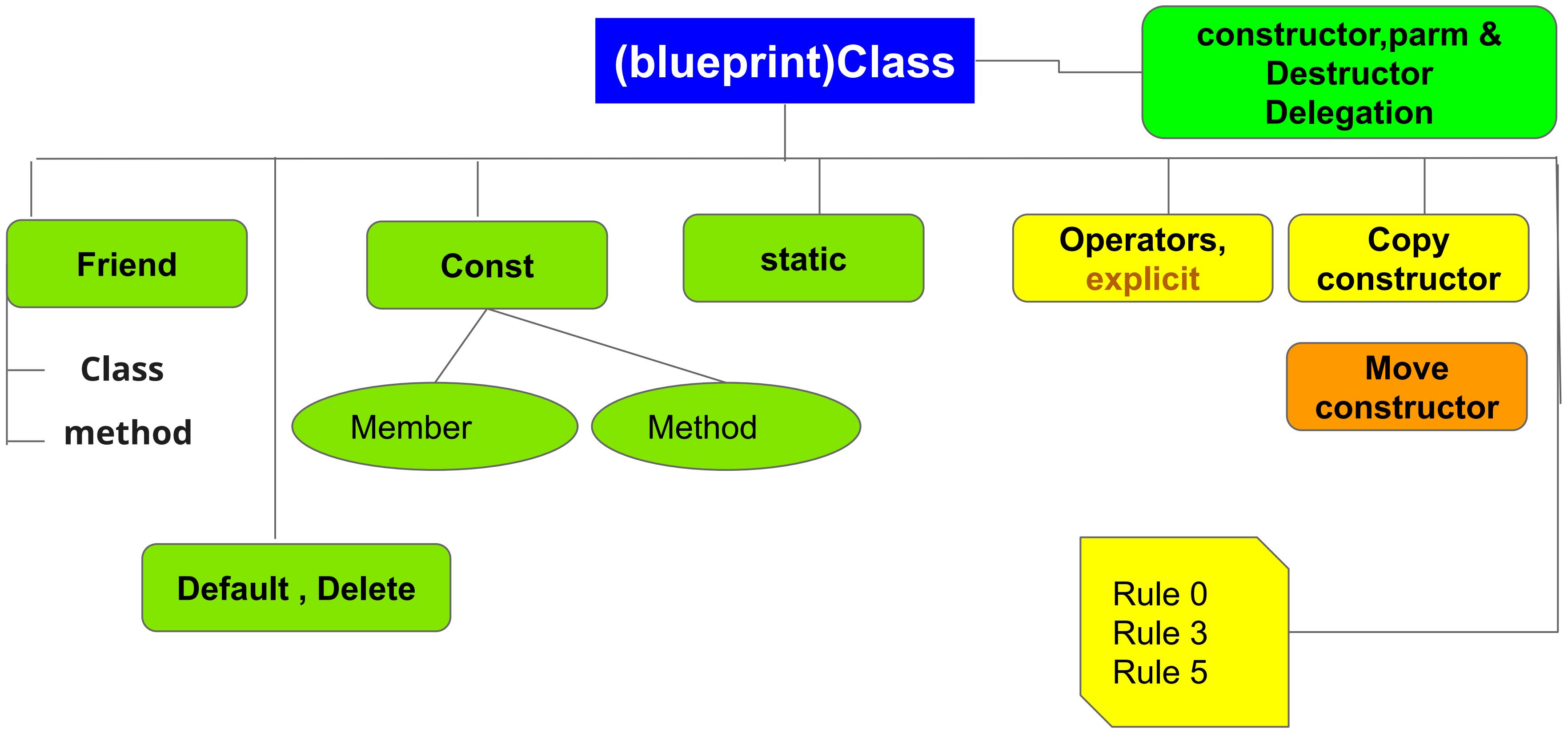
- Rvalue Reference

- rvalue ref Constructor

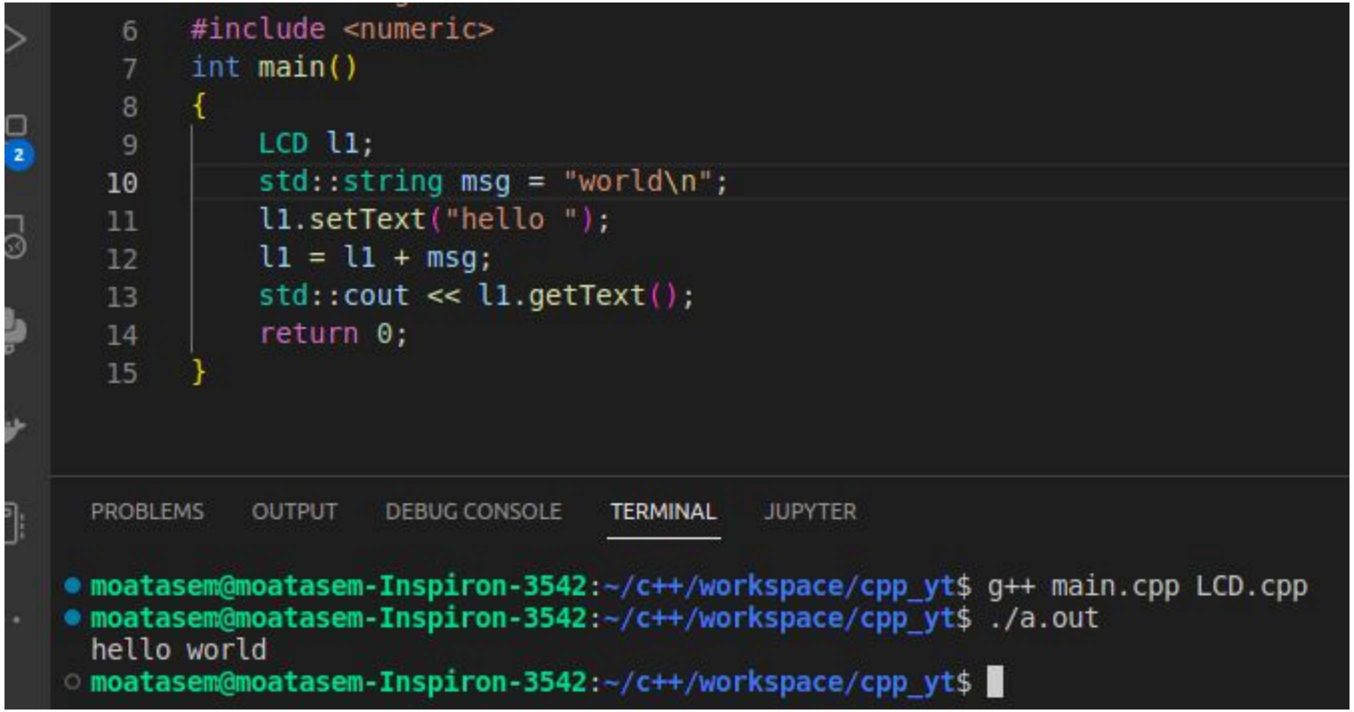
- operator rvalue Reference

-Rule 5 / Rule 3 / Rule 0

Features of class



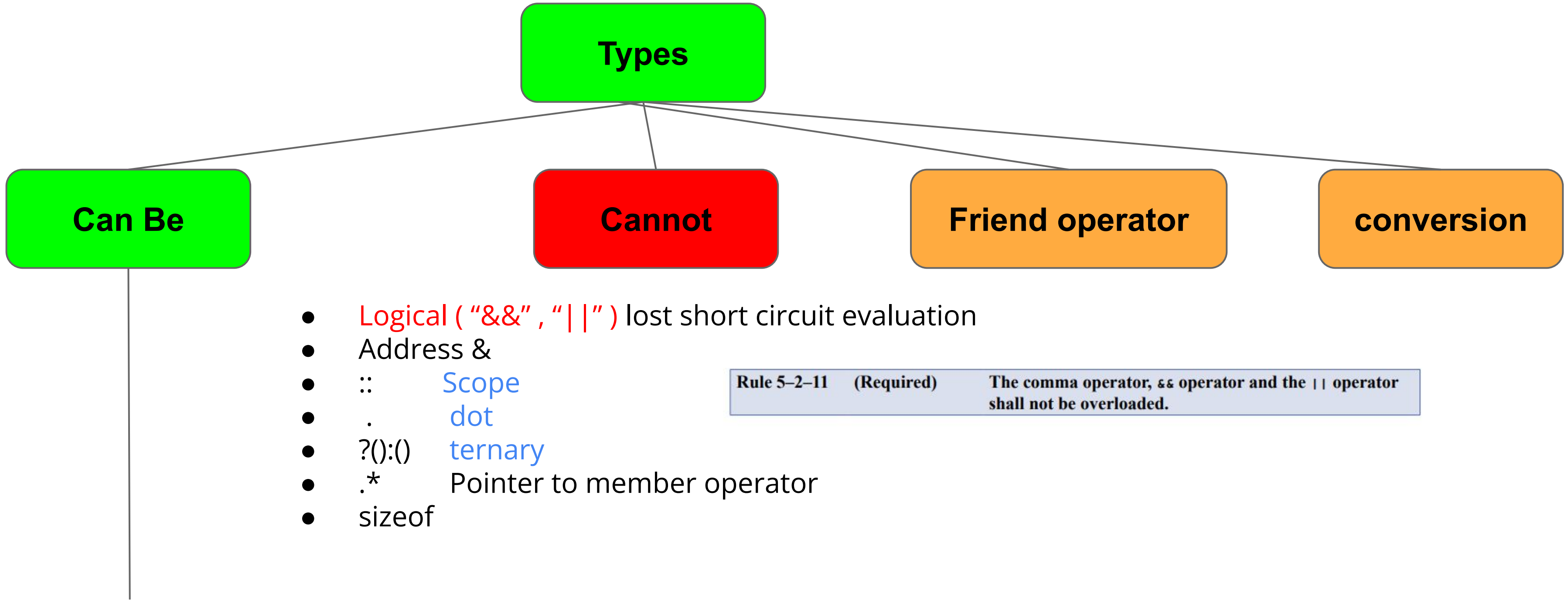
What is meant by operator overloading



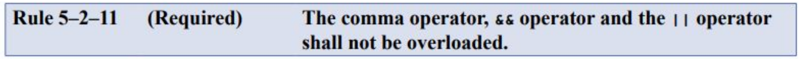
```cpp
#include <numeric>
int main()
{
    LCD l1;
    std::string msg = "world\n";
    l1.setText("hello ");
    l1 = l1 + msg;
    std::cout << l1.getText();
    return 0;
}
```

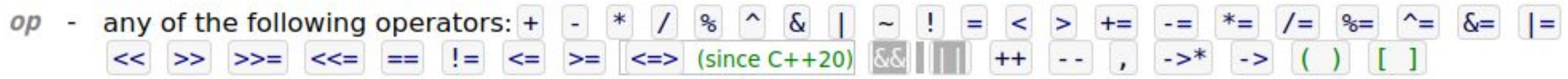
```
moatasem@moatasem-Inspiron-3542:~/c++/workspace/cpp_yt$ g++ main.cpp LCD.cpp
moatasem@moatasem-Inspiron-3542:~/c++/workspace/cpp_yt$ ./a.out
hello world
moatasem@moatasem-Inspiron-3542:~/c++/workspace/cpp_yt$ 
```

Operator overloading



- Logical (“&&” , “||”) lost short circuit evaluation
- Address &
- :: Scope
- . dot
- ?():() ternary
- .* Pointer to member operator
- sizeof

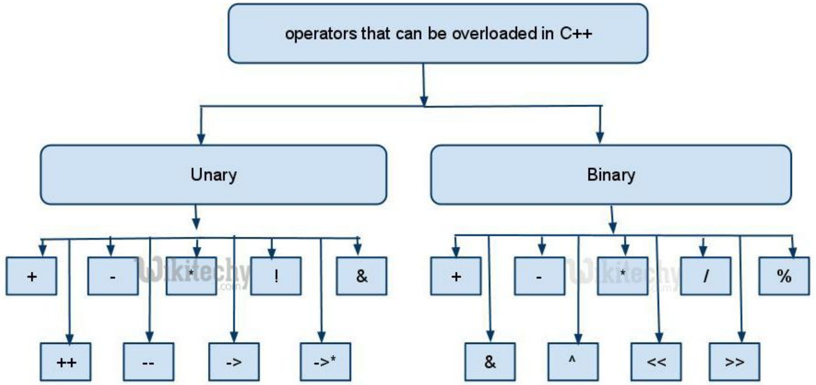

Rule 5–2–11 (Required) The comma operator, && operator and the || operator shall not be overloaded.


op - any of the following operators: + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> (since C++20) && || ++ -- , ->* -> ( ) [ ]

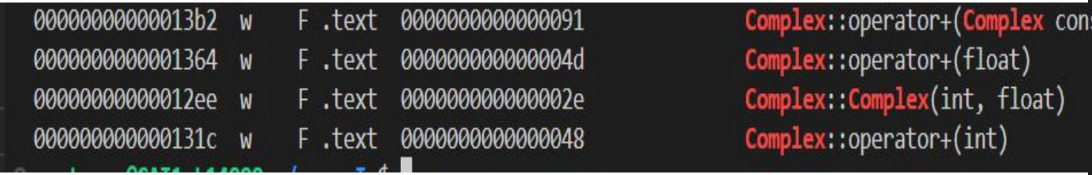
Operator overloading

keyword operator to be overloaded

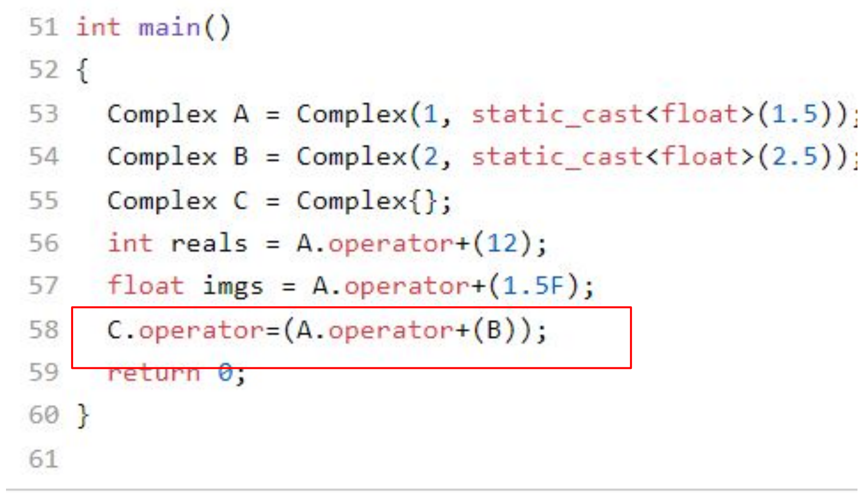
```
ReturnType classname :: Operator OperatorSymbol(argument list)
        {
            //Function Body
        }
```



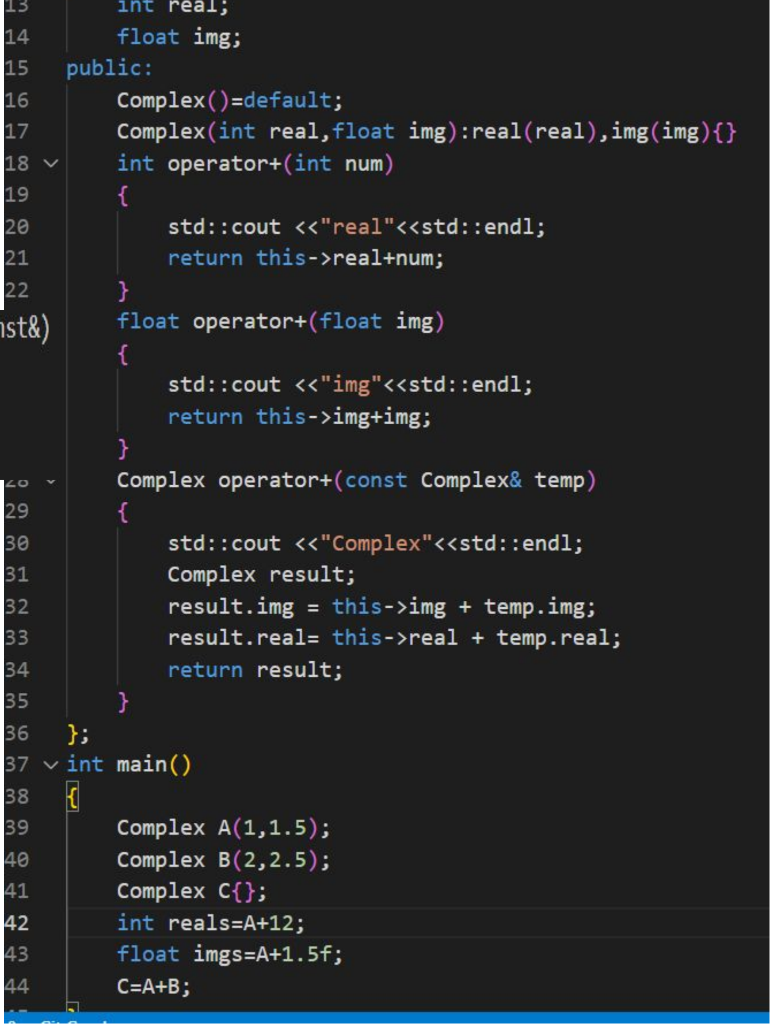
Operator +



```
00000000000013b2  w    F .text  0000000000000091           Complex::operator+(Complex const&)
0000000000001364  w    F .text  000000000000004d           Complex::operator+(float)
00000000000012ee  w    F .text  000000000000002e           Complex::Complex(int, float)
000000000000131c  w    F .text  0000000000000048           Complex::operator+(int)
```



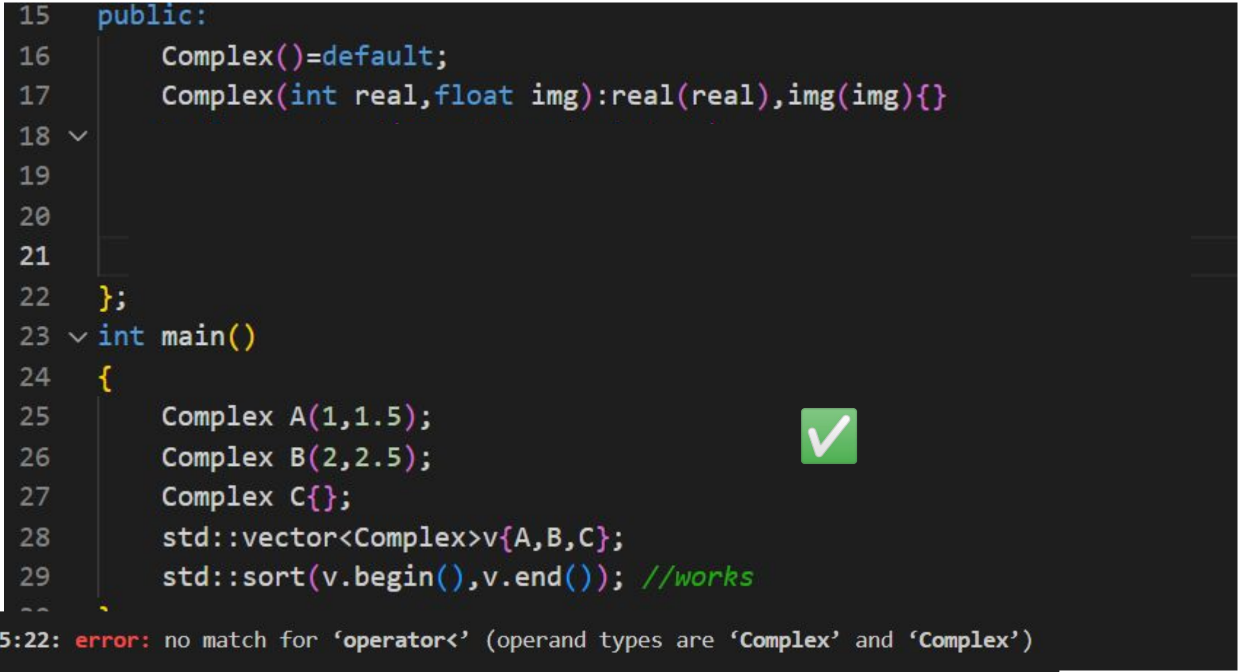
```cpp
int main()
{
  Complex A = Complex(1, static_cast<float>(1.5));
  Complex B = Complex(2, static_cast<float>(2.5));
  Complex C = Complex{};
  int reals = A.operator+(12);
  float imgs = A.operator+(1.5F);
  C.operator=(A.operator+(B));
  return 0;
}
```



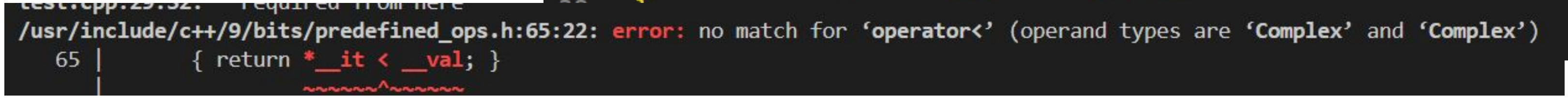
```cpp
    int real;
    float img;
public:
    Complex()=default;
    Complex(int real,float img):real(real),img(img){}
    int operator+(int num)
    {
        std::cout <<"real"<<std::endl;
        return this->real+num;
    }
    float operator+(float img)
    {
        std::cout <<"img"<<std::endl;
        return this->img+img;
    }
    Complex operator+(const Complex& temp)
    {
        std::cout <<"Complex"<<std::endl;
        Complex result;
        result.img = this->img + temp.img;
        result.real= this->real + temp.real;
        return result;
    }
};
int main()
{
    Complex A(1,1.5);
    Complex B(2,2.5);
    Complex C{};
    int reals=A+12;
    float imgs=A+1.5f;
    C=A+B;
```

Standard Lib

The Standard Library uses the less-than operator for sorting and ordering

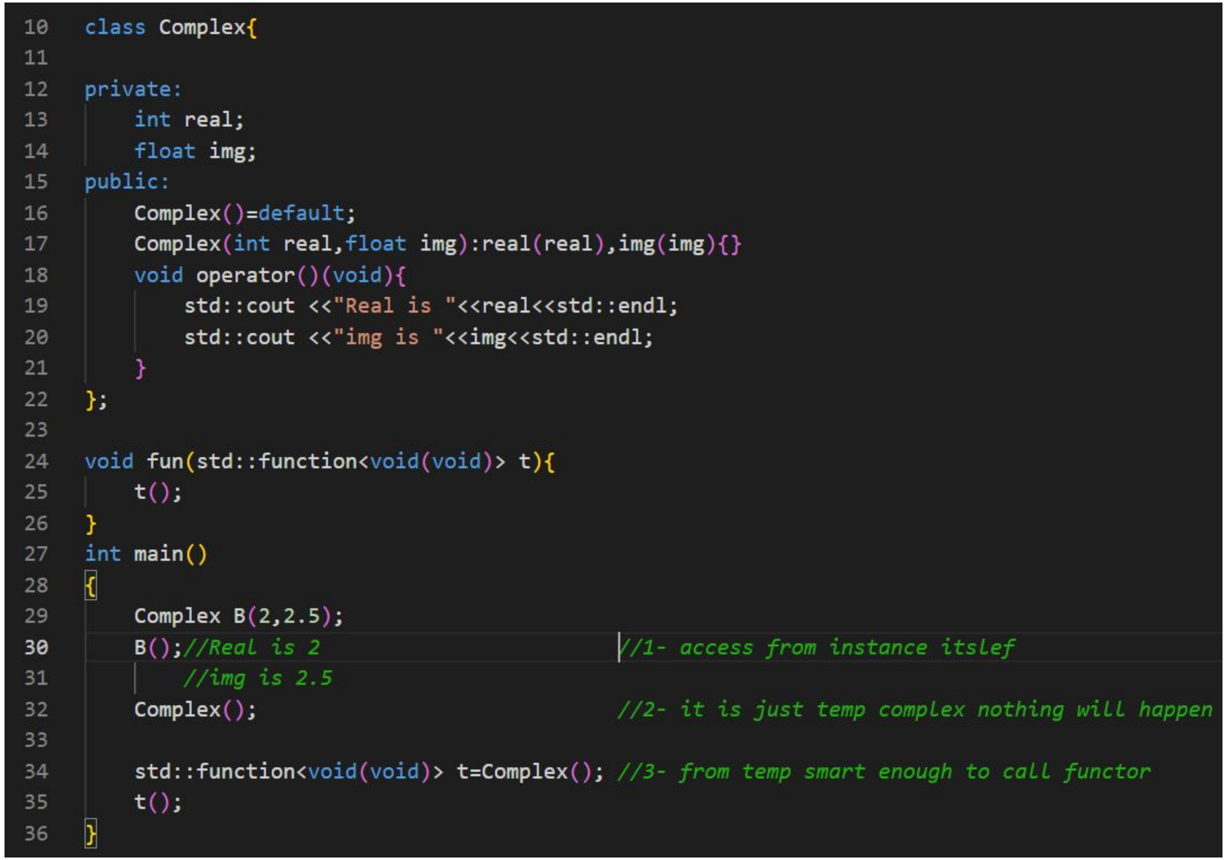


```cpp
public:
    Complex()=default;
    Complex(int real,float img):real(real),img(img){}



};
int main()
{
    Complex A(1,1.5);
    Complex B(2,2.5);
    Complex C{};
    std::vector<Complex>v{A,B,C};
    std::sort(v.begin(),v.end()); //works
```



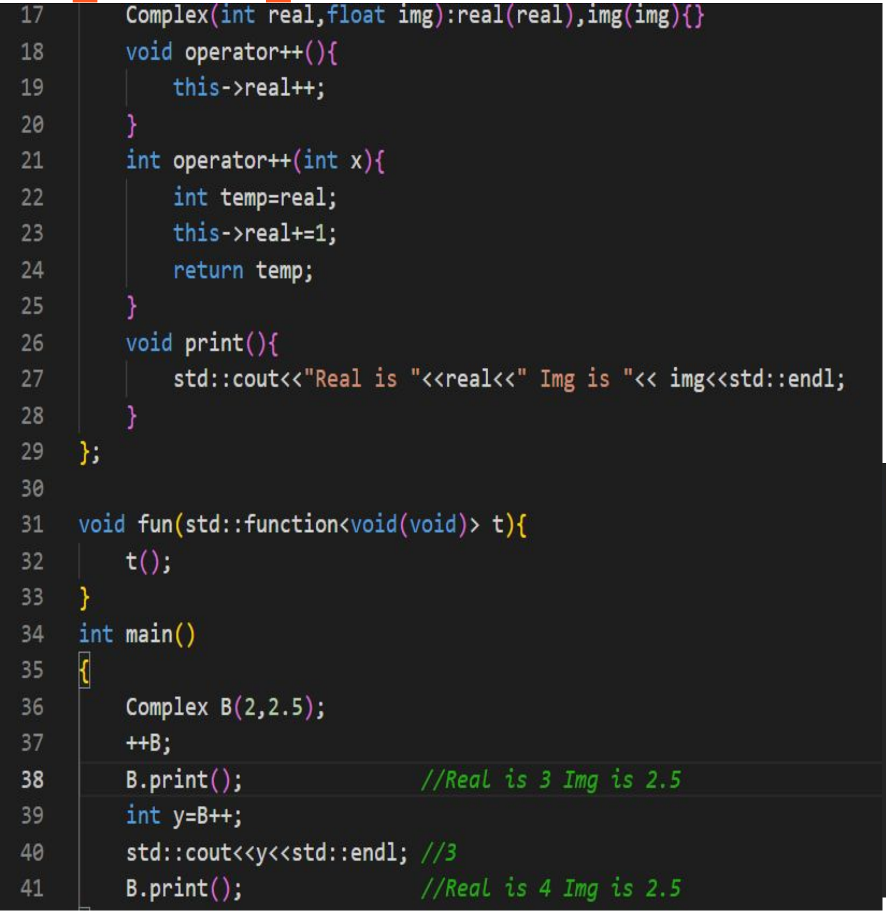
```
test.cpp:29:32:   required from here
/usr/include/c++/9/bits/predefined_ops.h:65:22: error: no match for ‘operator<’ (operand types are ‘Complex’ and ‘Complex’)
   65 |       { return *__it < __val; }
      |                ~~~~~~^~~~~~~
```

Functor

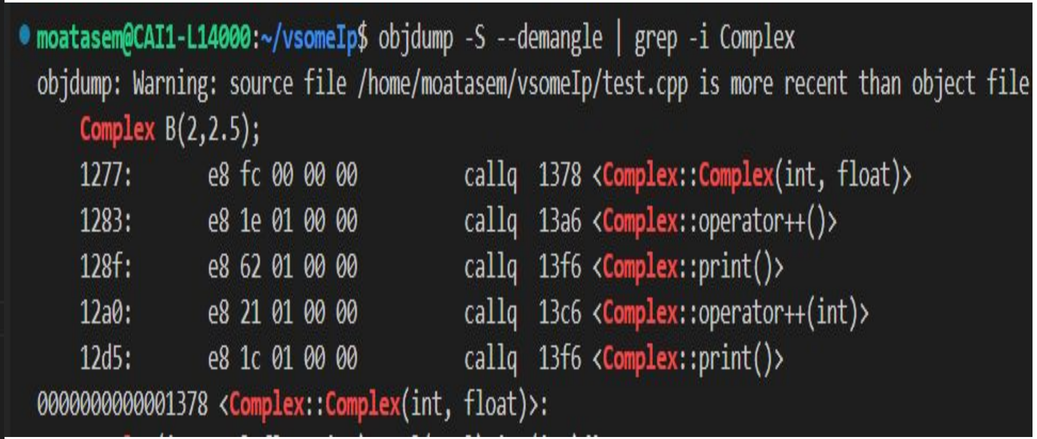


```cpp
class Complex{

private:
    int real;
    float img;
public:
    Complex()=default;
    Complex(int real,float img):real(real),img(img){}
    void operator()(void){
        std::cout <<"Real is "<<real<<std::endl;
        std::cout <<"img is "<<img<<std::endl;
    }
};

void fun(std::function<void(void)> t){
    t();
}
int main()
{
    Complex B(2,2.5);
    B();//Real is 2                                  //1- access from instance itslef
        //img is 2.5
    Complex();                                       //2- it is just temp complex nothing will happen

    std::function<void(void)> t=Complex(); //3- from temp smart enough to call functor
    t();
}
```

post/pre

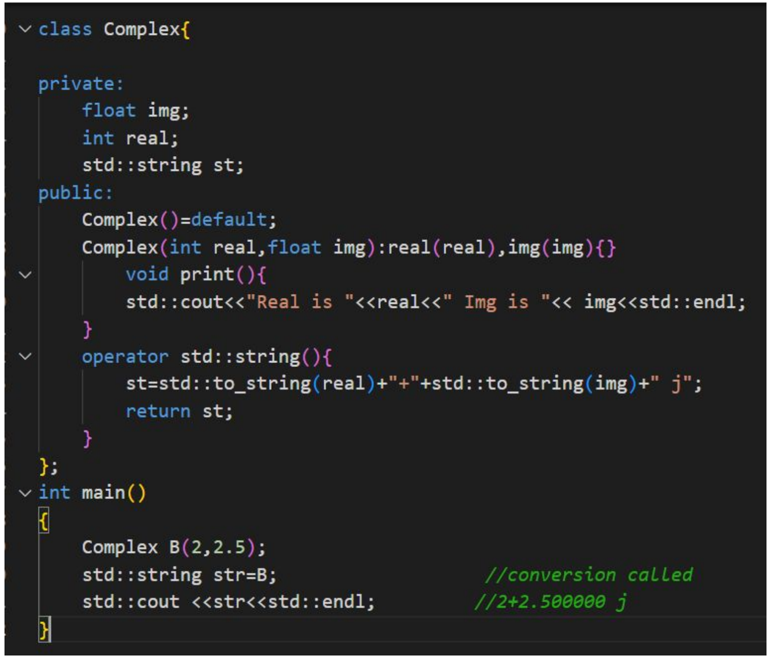


```cpp
    Complex(int real,float img):real(real),img(img){}
    void operator++(){
        this->real++;
    }
    int operator++(int x){
        int temp=real;
        this->real+=1;
        return temp;
    }
    void print(){
        std::cout<<"Real is "<<real<<" Img is "<< img<<std::endl;
    }
};

void fun(std::function<void(void)> t){
    t();
}
int main()
{
    Complex B(2,2.5);
    ++B;
    B.print();              //Real is 3 Img is 2.5
    int y=B++;
    std::cout<<y<<std::endl; //3
    B.print();              //Real is 4 Img is 2.5
```

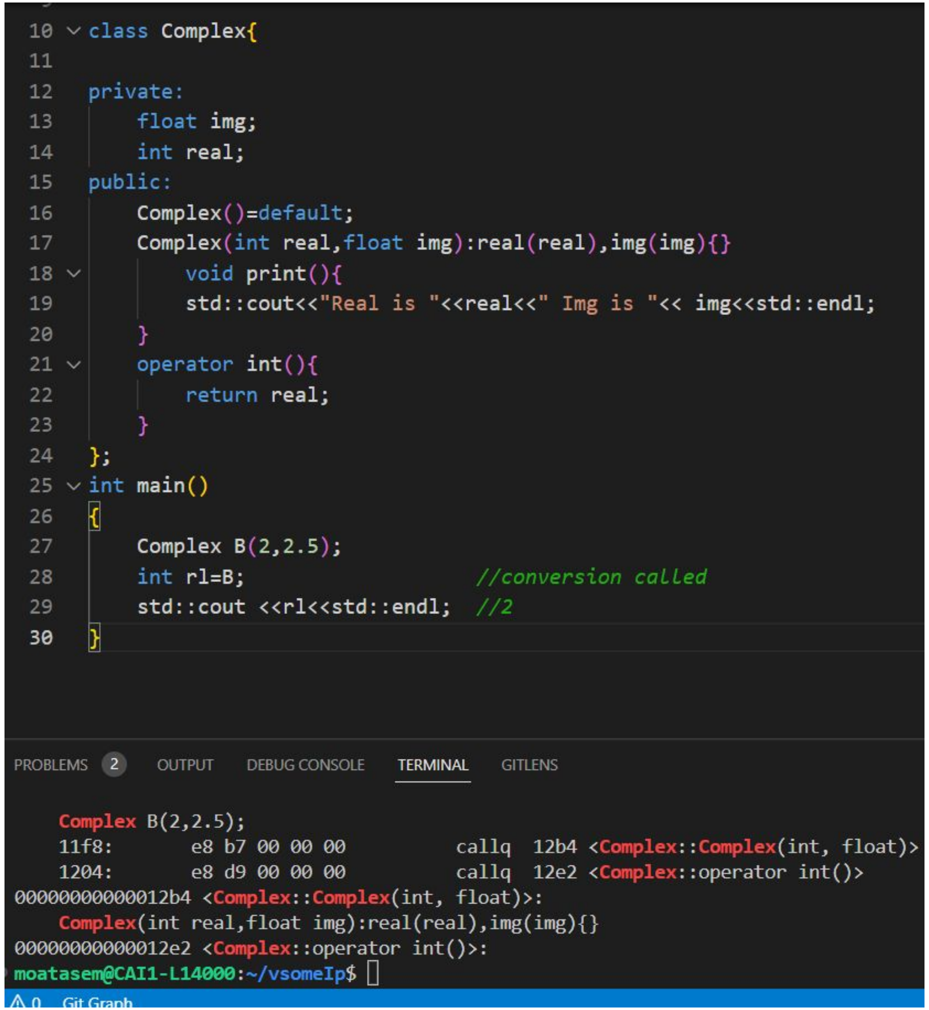


```
moatasem@CAI1-L14000:~/vsomeIp$ objdump -S --demangle | grep -i Complex
objdump: Warning: source file /home/moatasem/vsomeIp/test.cpp is more recent than object file
    Complex B(2,2.5);
    1277:       e8 fc 00 00 00          callq  1378 <Complex::Complex(int, float)>
    1283:       e8 1e 01 00 00          callq  13a6 <Complex::operator++()>
    128f:       e8 62 01 00 00          callq  13f6 <Complex::print()>
    12a0:       e8 21 01 00 00          callq  13c6 <Complex::operator++(int)>
    12d5:       e8 1c 01 00 00          callq  13f6 <Complex::print()>
0000000000001378 <Complex::Complex(int, float)>:
```

Conversion



```cpp
class Complex{

private:
    float img;
    int real;
    std::string st;
public:
    Complex()=default;
    Complex(int real,float img):real(real),img(img){}
        void print(){
        std::cout<<"Real is "<<real<<" Img is "<< img<<std::endl;
    }
    operator std::string(){
        st=std::to_string(real)+"+"+std::to_string(img)+" j";
        return st;
    }
};
int main()
{

    Complex B(2,2.5);
    std::string str=B;                  //conversion called
    std::cout <<str<<std::endl;        //2+2.500000 j
}
```

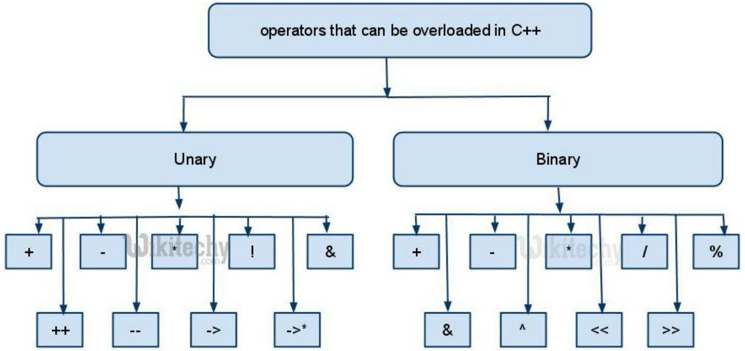


```cpp
class Complex{

private:
    float img;
    int real;
public:
    Complex()=default;
    Complex(int real,float img):real(real),img(img){}
        void print(){
        std::cout<<"Real is "<<real<<" Img is "<< img<<std::endl;
    }
    operator int(){
        return real;
    }
};
int main()
{

    Complex B(2,2.5);
    int rl=B;                     //conversion called
    std::cout <<rl<<std::endl;   //2
}
```

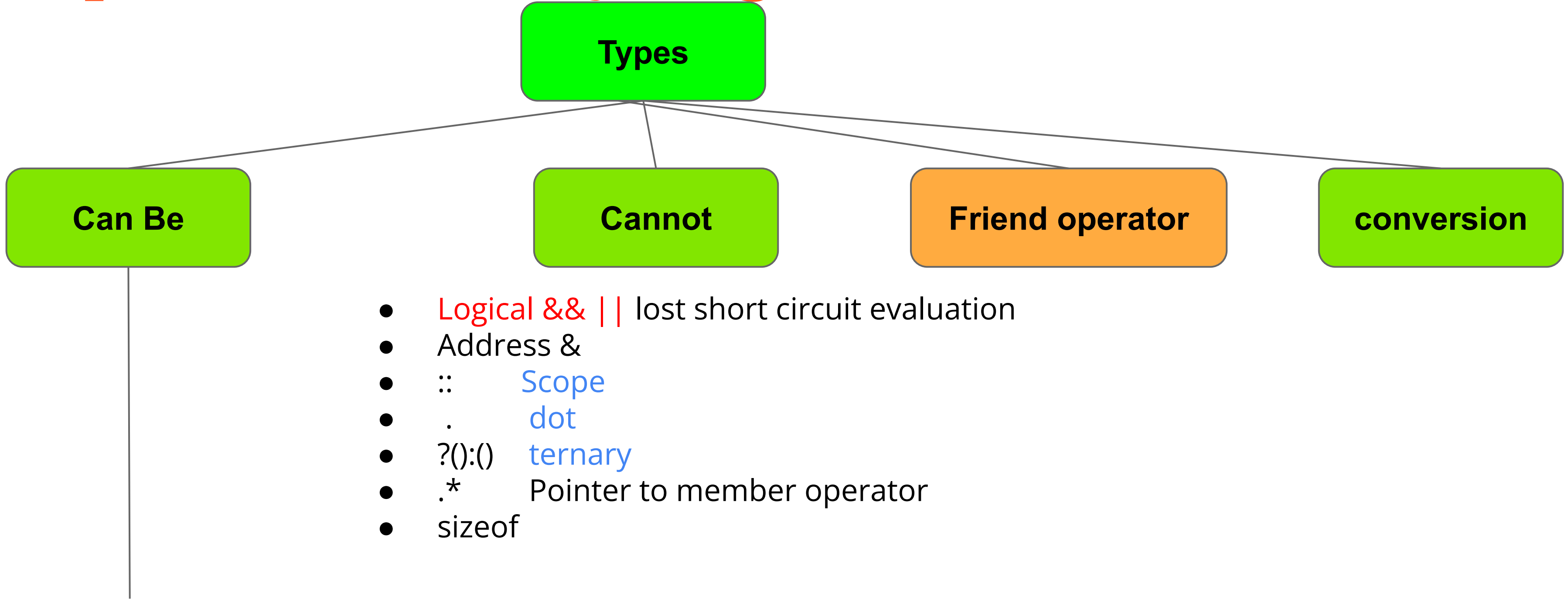
```
    Complex B(2,2.5);
    11f8:       e8 b7 00 00 00          callq  12b4 <Complex::Complex(int, float)>
    1204:       e8 d9 00 00 00          callq  12e2 <Complex::operator int()>
00000000000012b4 <Complex::Complex(int, float)>:
    Complex(int real,float img):real(real),img(img){}
00000000000012e2 <Complex::operator int()>:
moatasem@CAI1-L14000:~/vsomeIp$
```

Operator overloading

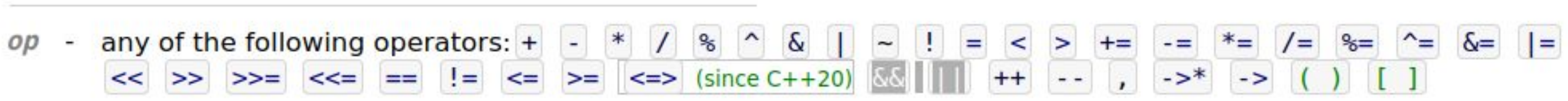
1. Binary Arithmetic : +, -, *, /, %
2. Unary Arithmetic : +, -, ++,
3. Assignment: =, +=,*=, /=,-=, %= (3/5 rule)
4. Bitwise: & , | , << , >> , ~ , ^ (Stream)
5. De-referencing: (->)
6. Dynamic memory allocation : New, delete (memory)
7. Subscript: [ ]
8. Function call: ()
9. Logical: &, | |, !
10. Relational: >, < , = =, <=, >=



Operator overloading task:please check everything



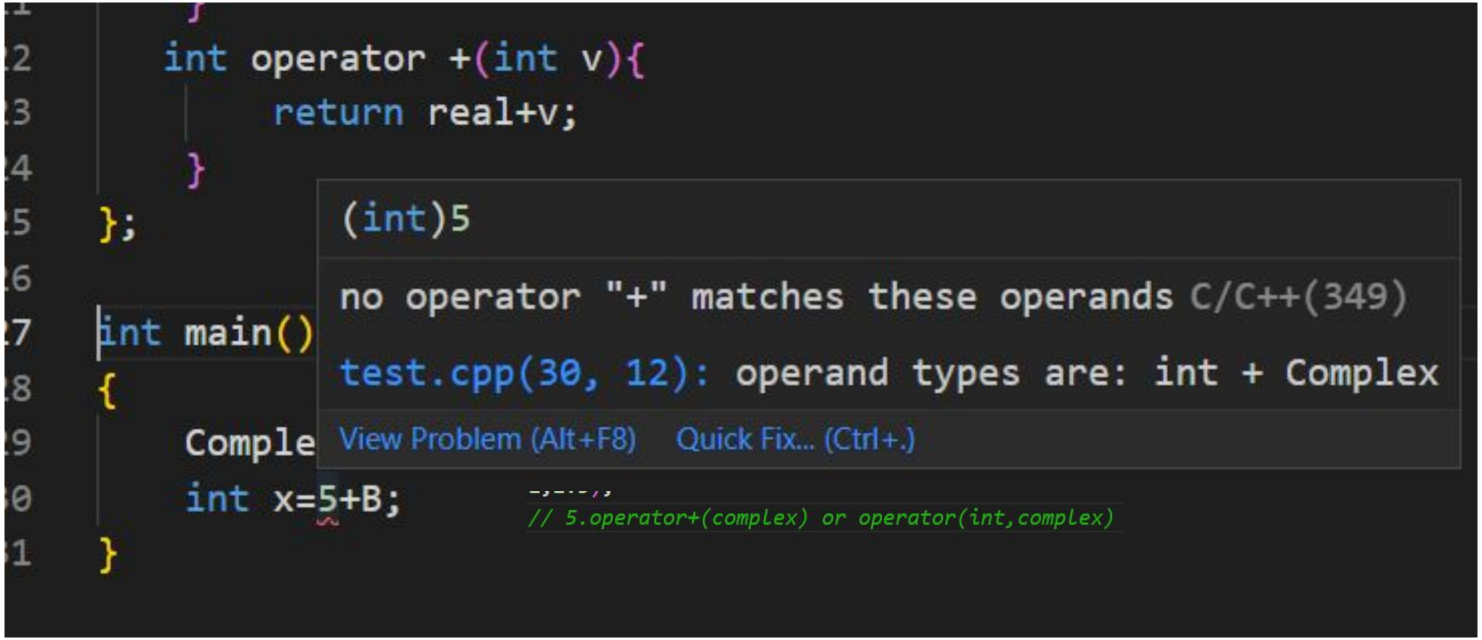
- Logical && || lost short circuit evaluation
- Address &
- :: Scope
- . dot
- ?():() ternary
- .* Pointer to member operator
- sizeof



op - any of the following operators: + - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> (since C++20) && || ++ -- , ->* -> ( ) [ ]

Friend operator

Issue

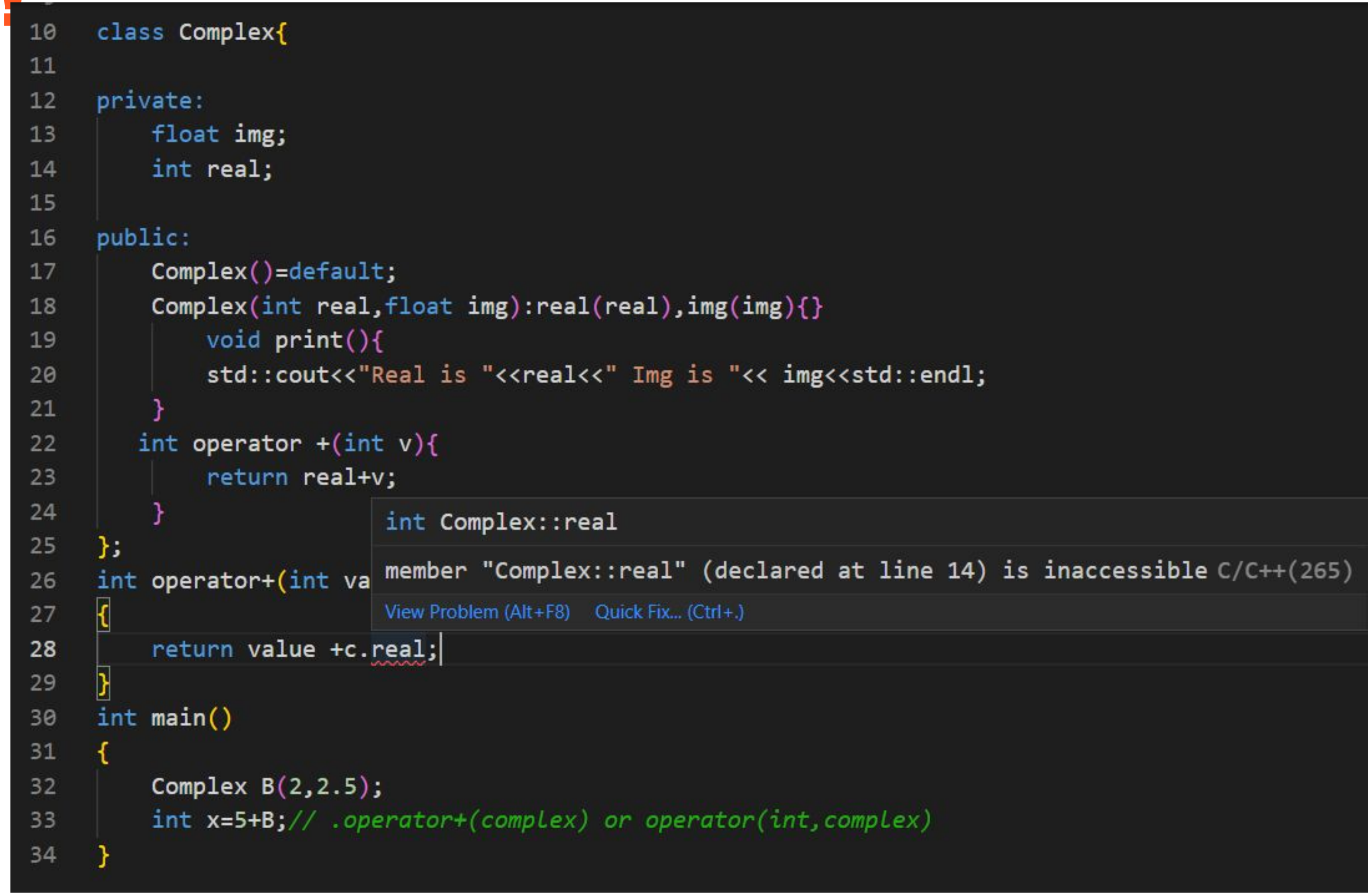


```
    int operator +(int v){
        return real+v;
    }
};

int main()
{
    Comple
    int x=5+B;
}
```

```
(int)5

no operator "+" matches these operands C/C++(349)
test.cpp(30, 12): operand types are: int + Complex
```

```
// 5.operator+(complex) or operator(int,complex)
```

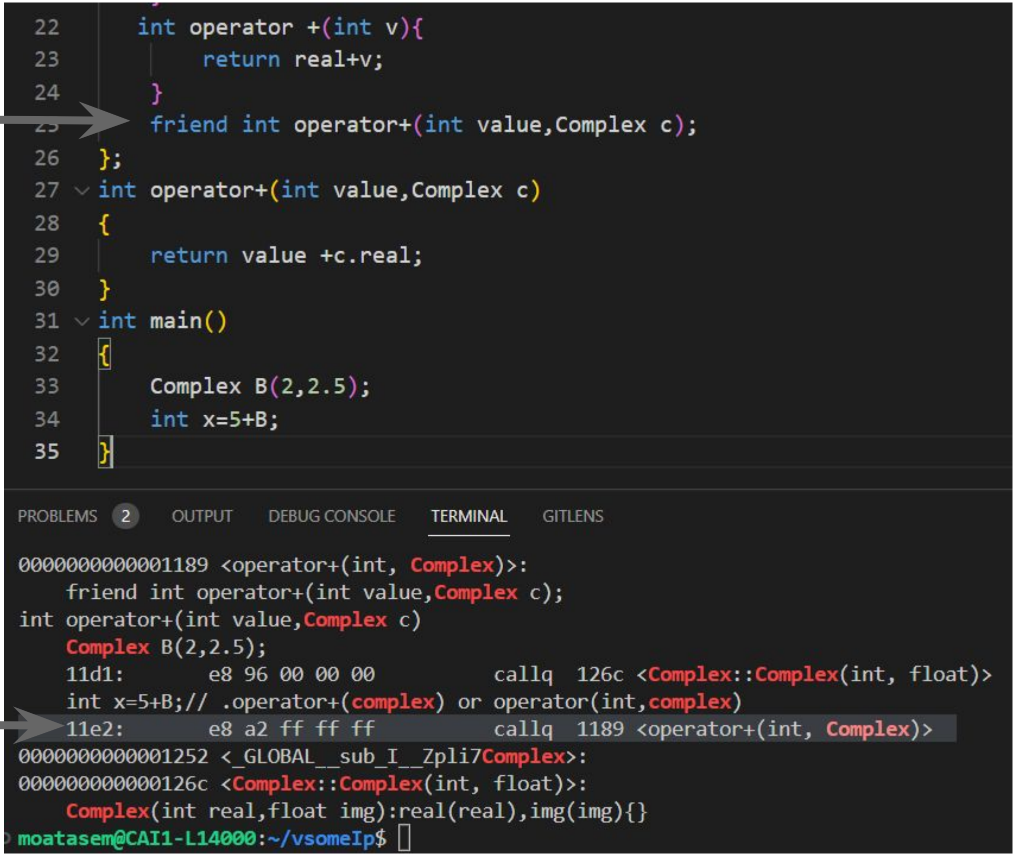
Private !



```
class Complex{

private:
    float img;
    int real;

public:
    Complex()=default;
    Complex(int real,float img):real(real),img(img){}
        void print(){
        std::cout<<"Real is "<<real<<" Img is "<< img<<std::endl;
    }
   int operator +(int v){
        return real+v;
    }
};
int operator+(int va
{
    return value +c.real;
}
int main()
{
    Complex B(2,2.5);
    int x=5+B;// .operator+(complex) or operator(int,complex)
}
```

```
int Complex::real

member "Complex::real" (declared at line 14) is inaccessible C/C++(265)
```

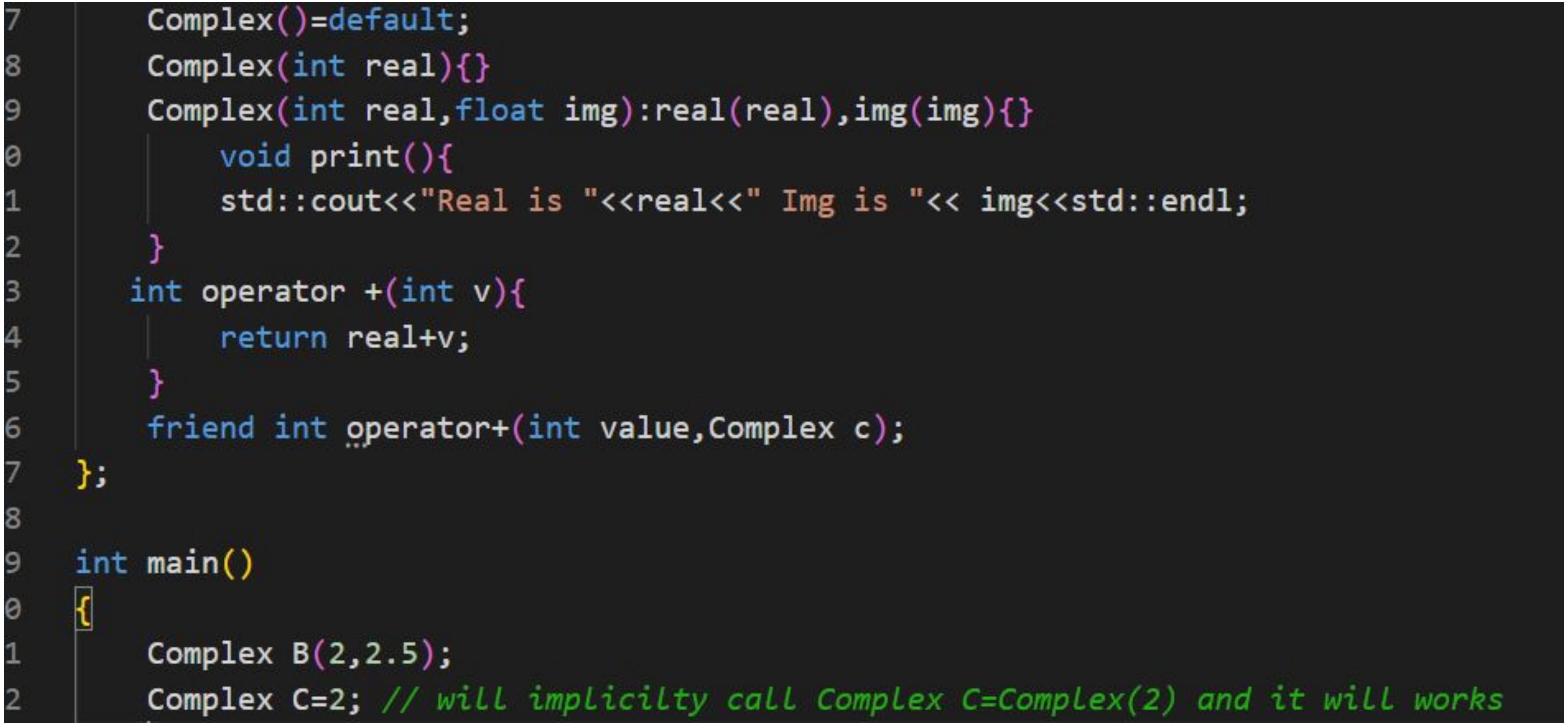
Solution



```cpp
    int operator +(int v){
        return real+v;
    }
    friend int operator+(int value,Complex c);
};
int operator+(int value,Complex c)
{
    return value +c.real;
}
int main()
{
    Complex B(2,2.5);
    int x=5+B;
}
```

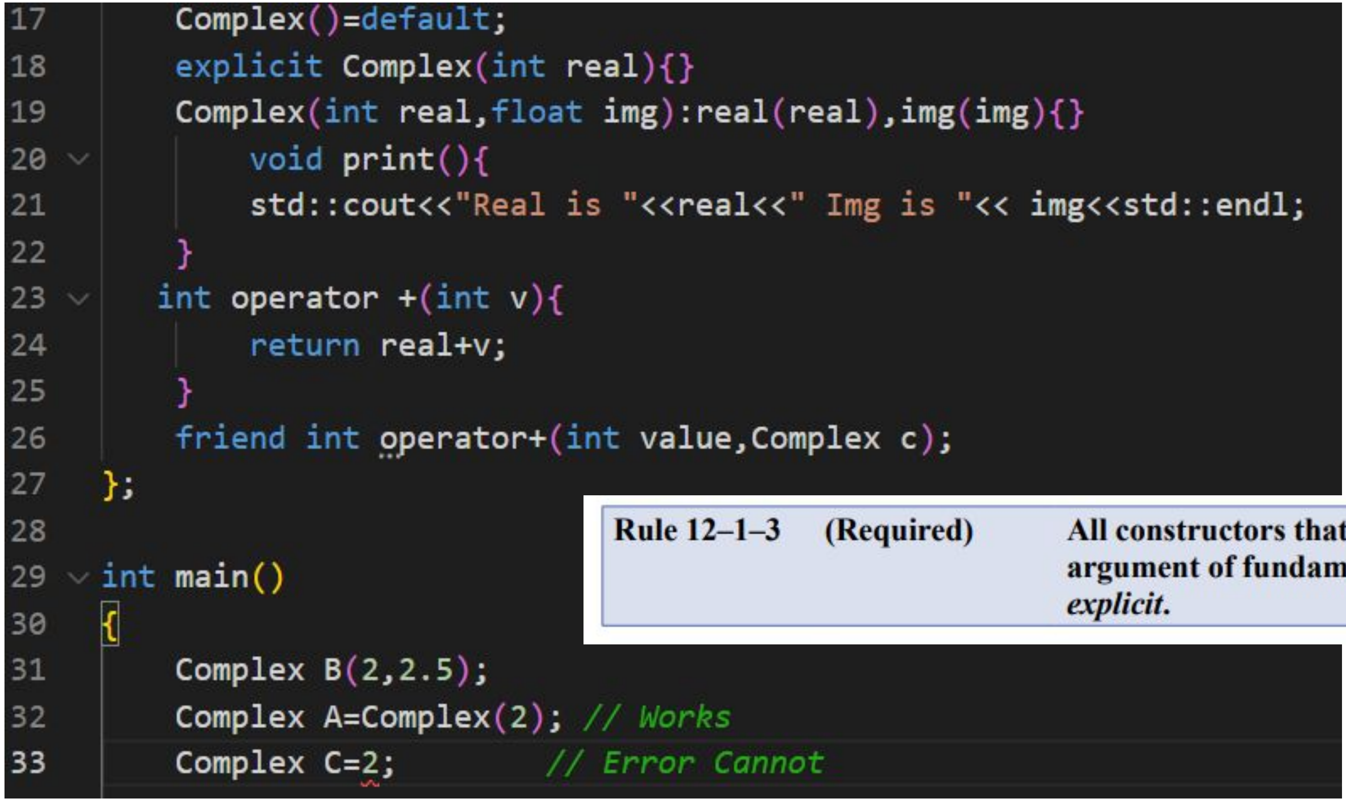
```
0000000000001189 <operator+(int, Complex)>:
    friend int operator+(int value,Complex c);
int operator+(int value,Complex c)
    Complex B(2,2.5);
    11d1:       e8 96 00 00 00          callq  126c <Complex::Complex(int, float)>
    int x=5+B;// .operator+(complex) or operator(int,complex)
    11e2:       e8 a2 ff ff ff          callq  1189 <operator+(int, Complex)>
0000000000001252 <_GLOBAL__sub_I__Zpli7Complex>:
000000000000126c <Complex::Complex(int, float)>:
    Complex(int real,float img):real(real),img(img){}
moatasem@CAI1-L14000:~/vsomeIp$
```

Implicit Conversion

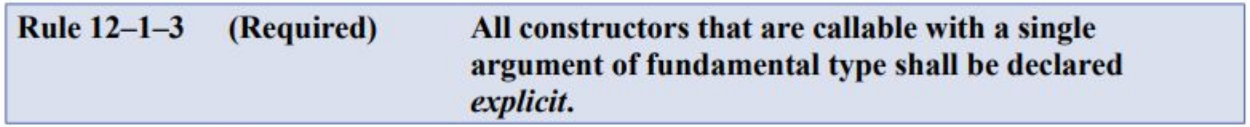


```
        Complex()=default;
        Complex(int real){}
        Complex(int real,float img):real(real),img(img){}
            void print(){
            std::cout<<"Real is "<<real<<" Img is "<< img<<std::endl;
        }
       int operator +(int v){
            return real+v;
        }
        friend int operator+(int value,Complex c);
    };

    int main()
    {
        Complex B(2,2.5);
        Complex C=2; // will implicilty call Complex C=Complex(2) and it will works
```

explicit

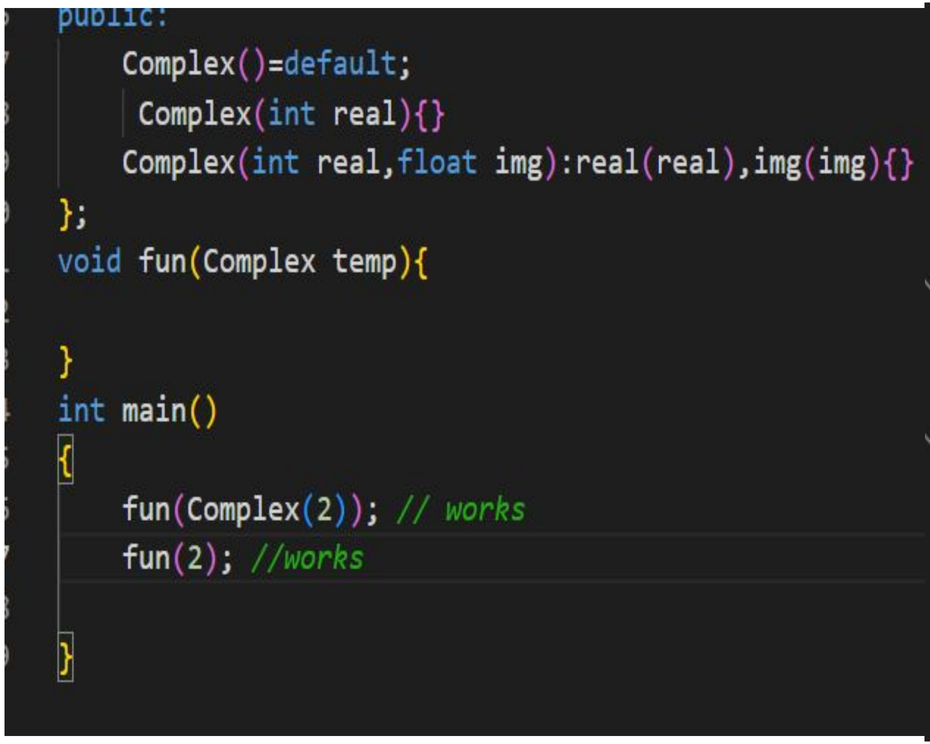


```
    Complex()=default;
    explicit Complex(int real){}
    Complex(int real,float img):real(real),img(img){}
        void print(){
        std::cout<<"Real is "<<real<<" Img is "<< img<<std::endl;
    }
   int operator +(int v){
        return real+v;
    }
    friend int operator+(int value,Complex c);
};

int main()
{
    Complex B(2,2.5);
    Complex A=Complex(2); // Works
    Complex C=2;          // Error Cannot
```

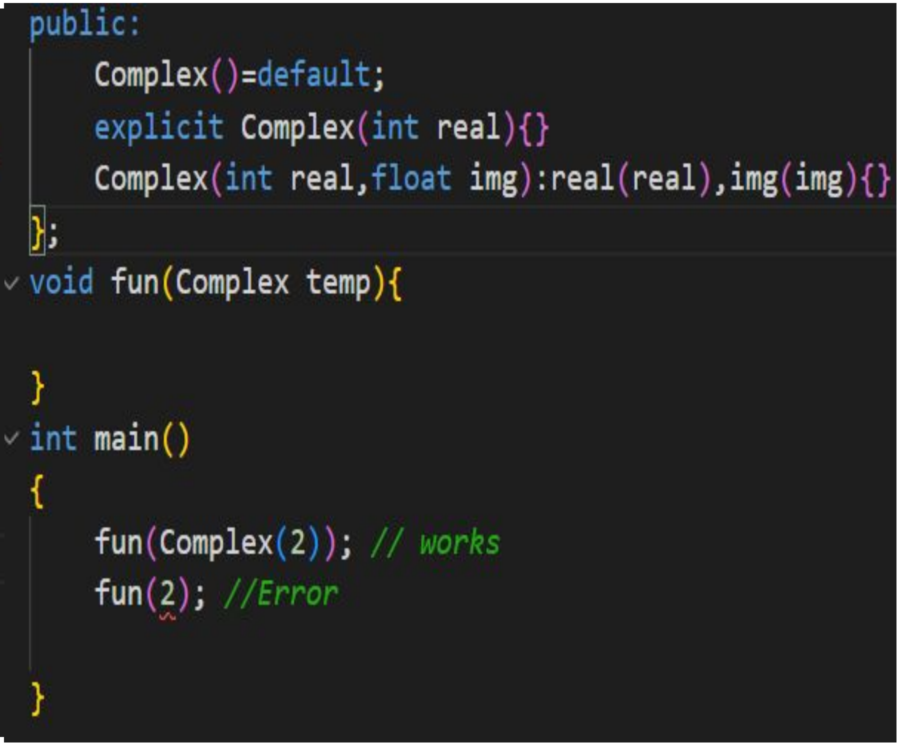


Rule 12–1–3 (Required) **All constructors that are callable with a single argument of fundamental type shall be declared *explicit*.**

Usage

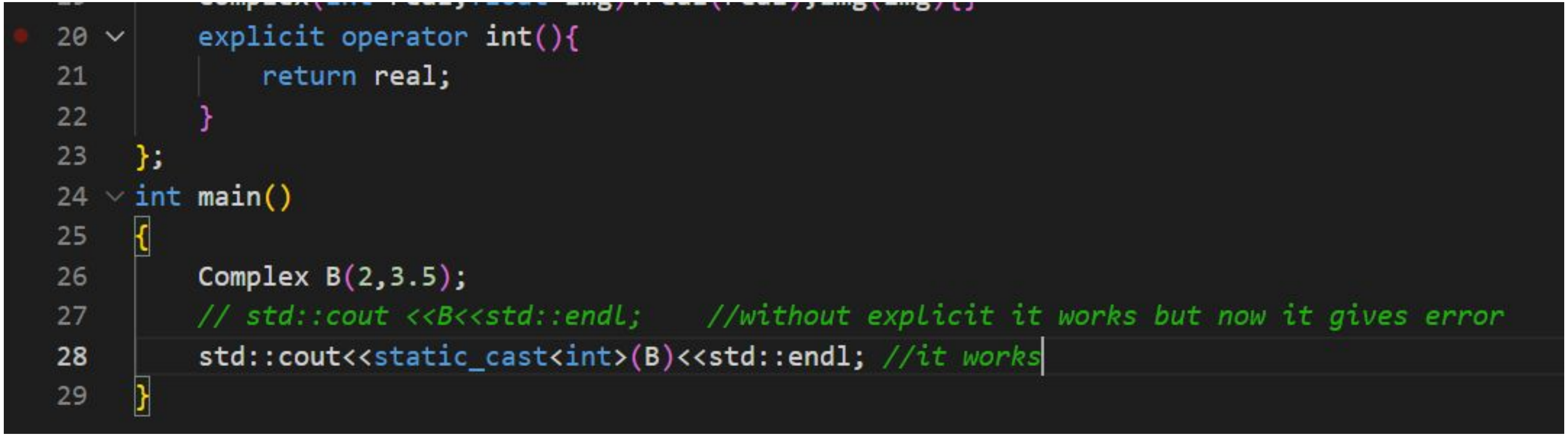


```
public:
    Complex()=default;
     Complex(int real){}
    Complex(int real,float img):real(real),img(img){}
};
void fun(Complex temp){

}
int main()
{
    fun(Complex(2)); // works
    fun(2); //works


}
```



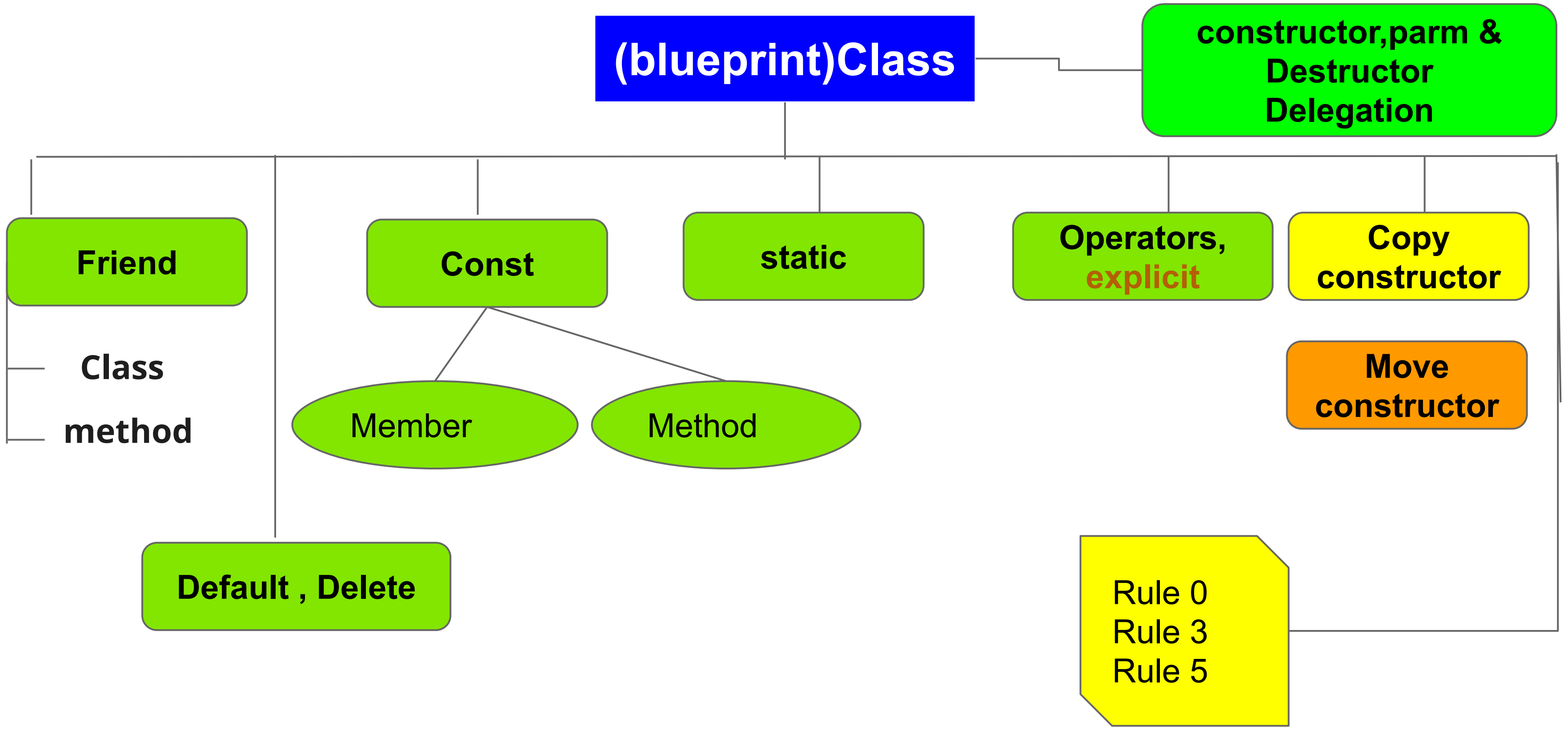
```
public:
    Complex()=default;
    explicit Complex(int real){}
    Complex(int real,float img):real(real),img(img){}
};
void fun(Complex temp){

}
int main()
{
    fun(Complex(2)); // works
    fun(2); //Error

}
```

Explicit with operator



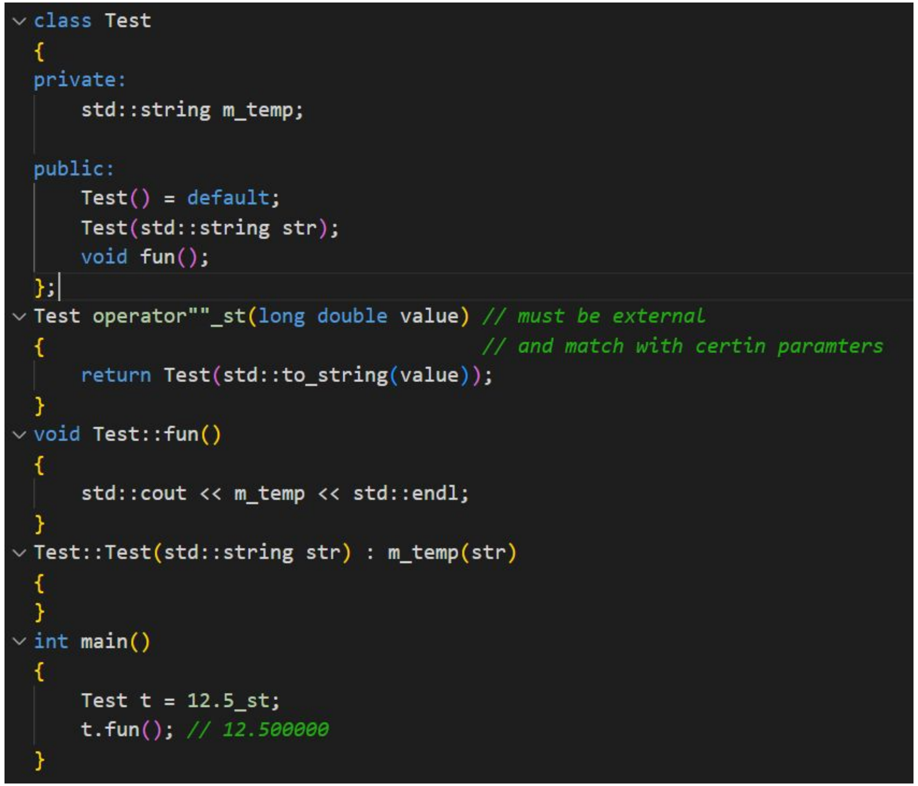
```
    explicit operator int(){
        return real;
    }
};
int main()
{
    Complex B(2,3.5);
    // std::cout <<B<<std::endl;    //without explicit it works but now it gives error
    std::cout<<static_cast<int>(B)<<std::endl; //it works
}
```

Features of class



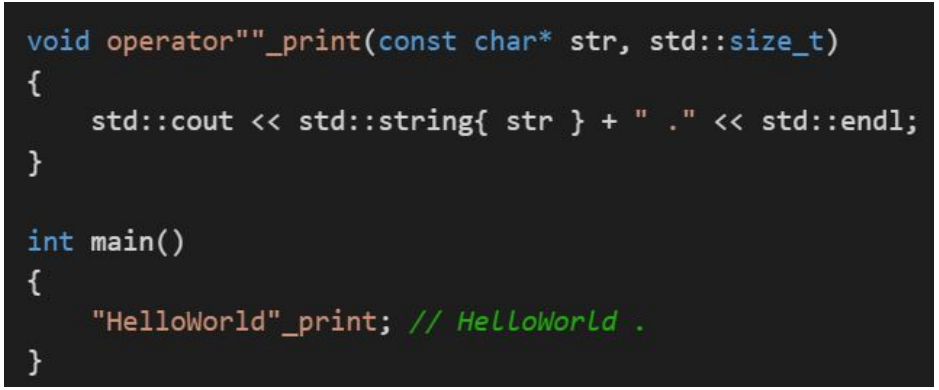
Q&A

Custom literal



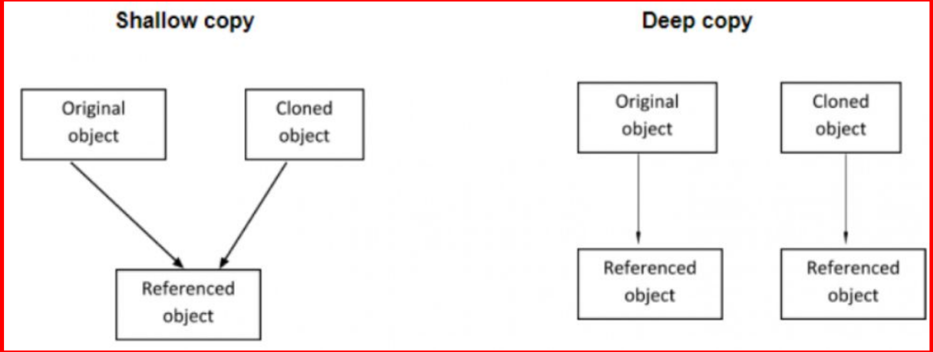
```cpp
class Test
{
private:
    std::string m_temp;

public:
    Test() = default;
    Test(std::string str);
    void fun();
};
Test operator""_st(long double value) // must be external
{                                     // and match with certin paramters
    return Test(std::to_string(value));
}
void Test::fun()
{
    std::cout << m_temp << std::endl;
}
Test::Test(std::string str) : m_temp(str)
{
}
int main()
{
    Test t = 12.5_st;
    t.fun(); // 12.500000
}
```

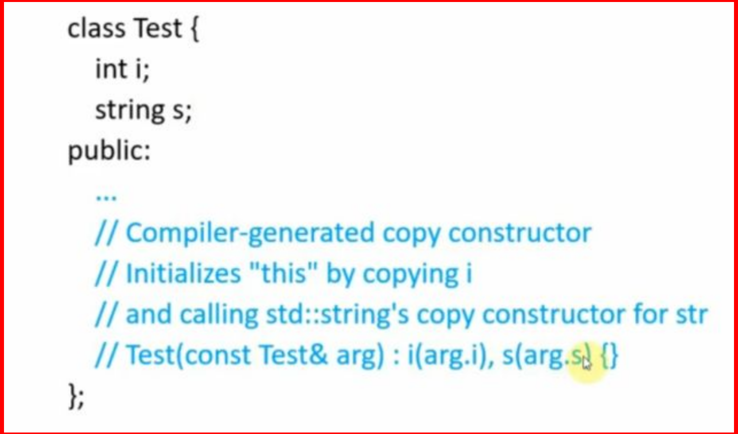
User-defined literals



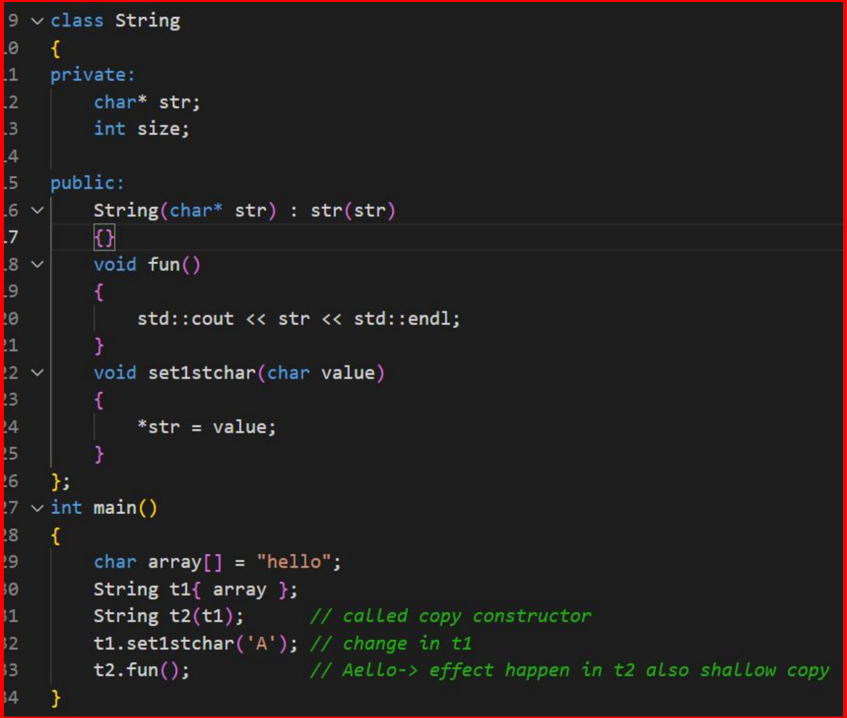
```cpp
void operator""_print(const char* str, std::size_t)
{
    std::cout << std::string{ str } + " ." << std::endl;
}

int main()
{
    "HelloWorld"_print; // HelloWorld .
}
```

Copy constructor





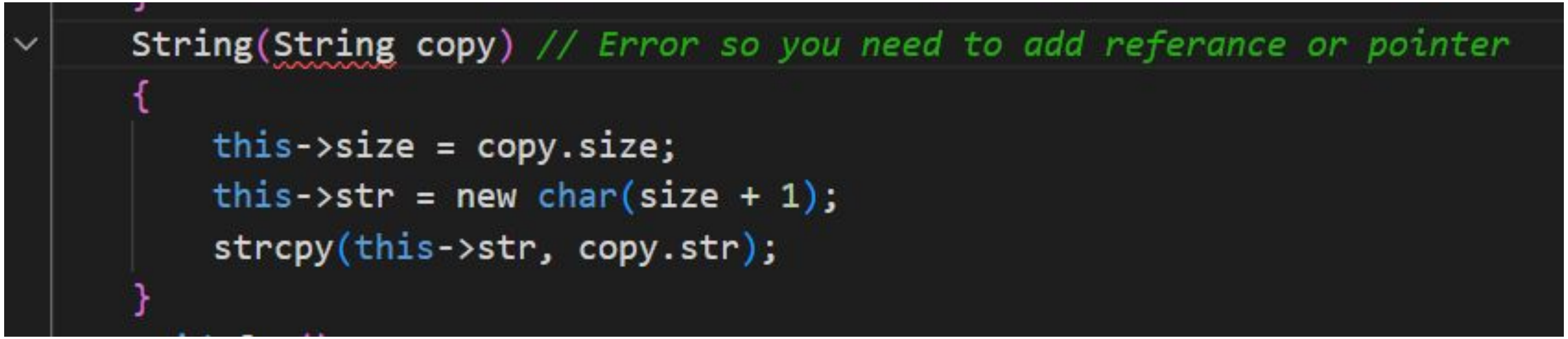
```
class Test {
   int i;
   string s;
public:
   ...
   // Compiler-generated copy constructor
   // Initializes "this" by copying i
   // and calling std::string's copy constructor for str
   // Test(const Test& arg) : i(arg.i), s(arg.s) {}
};
```



```
class String
{
private:
    char* str;
    int size;

public:
    String(char* str) : str(str)
    {}
    void fun()
    {
        std::cout << str << std::endl;
    }
    void set1stchar(char value)
    {
        *str = value;
    }
};
int main()
{
    char array[] = "hello";
    String t1{ array };
    String t2(t1);      // called copy constructor
    t1.set1stchar('A'); // change in t1
    t2.fun();           // Aello-> effect happen in t2 also shallow copy
}
```

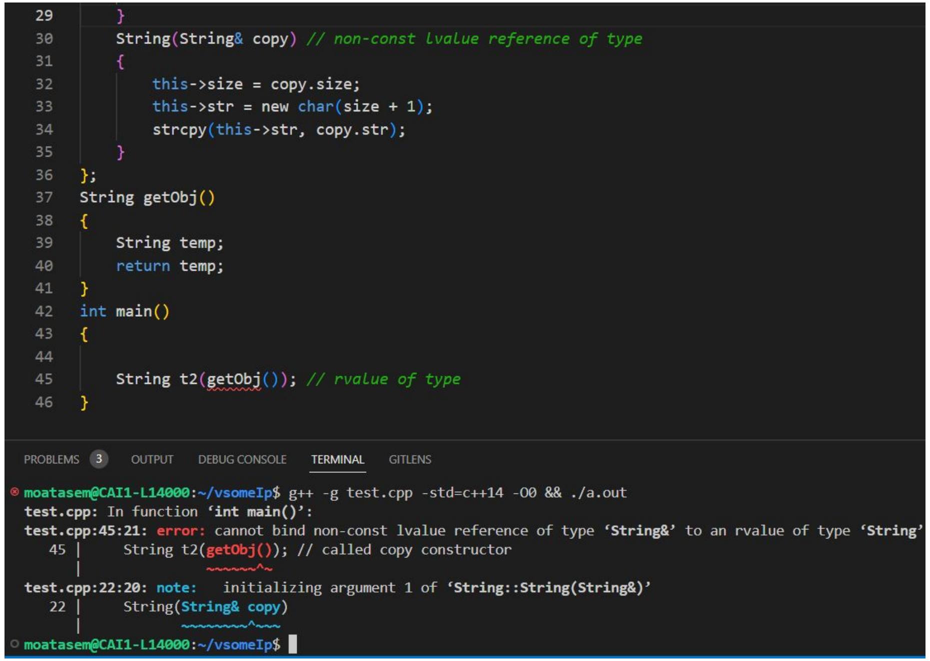
Syntax of copy constructor

```
classname (const classname &obj) {
   // body of constructor
}
```



```
    String(String copy) // Error so you need to add referance or pointer
    {
        this->size = copy.size;
        this->str = new char(size + 1);
        strcpy(this->str, copy.str);
    }
```

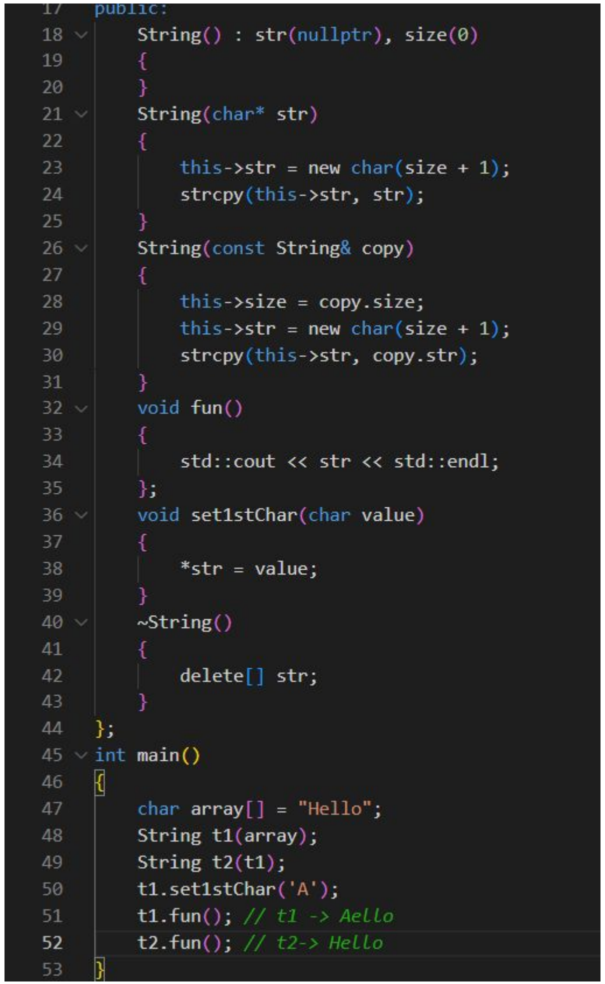
Deep Copy but there is an issue (rvalue)



```cpp
    }
    String(String& copy) // non-const lvalue reference of type
    {
        this->size = copy.size;
        this->str = new char(size + 1);
        strcpy(this->str, copy.str);
    }
};
String getObj()
{
    String temp;
    return temp;
}
int main()
{

    String t2(getObj()); // rvalue of type
}
```

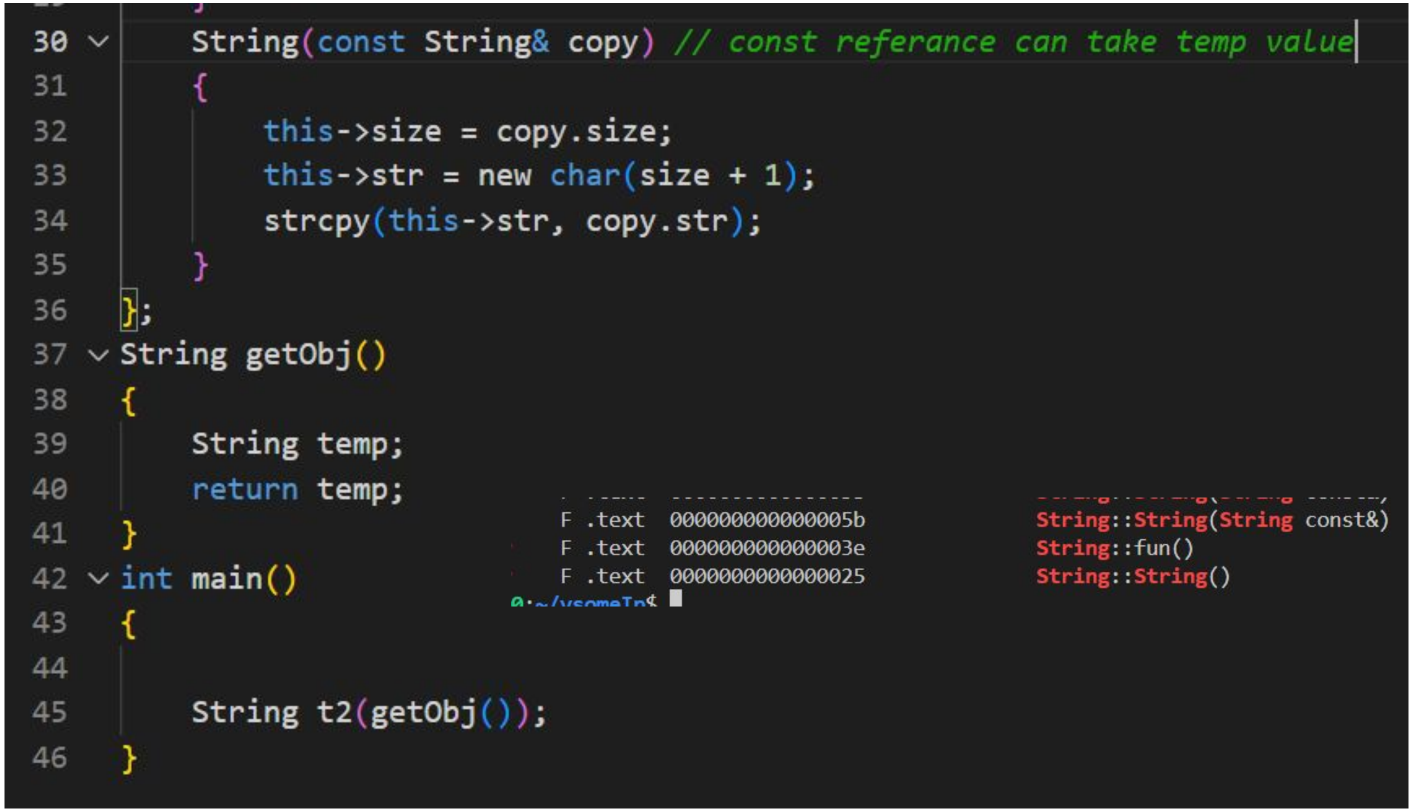
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL   GITLENS

```
⊗ moatasem@CAI1-L14000:~/vsomeIp$ g++ -g test.cpp -std=c++14 -O0 && ./a.out
test.cpp: In function ‘int main()’:
test.cpp:45:21: error: cannot bind non-const lvalue reference of type ‘String&’ to an rvalue of type ‘String’
   45 |     String t2(getObj()); // called copy constructor
      |               ~~~~~~^~
test.cpp:22:20: note:   initializing argument 1 of ‘String::String(String&)’
   22 |     String(String& copy)
      |            ~~~~~~~~^~~~
○ moatasem@CAI1-L14000:~/vsomeIp$ 
```



```cpp
public:
    String() : str(nullptr), size(0)
    {
    }
    String(char* str)
    {
        this->str = new char(size + 1);
        strcpy(this->str, str);
    }
    String(const String& copy)
    {
        this->size = copy.size;
        this->str = new char(size + 1);
        strcpy(this->str, copy.str);
    }
    void fun()
    {
        std::cout << str << std::endl;
    };
    void set1stChar(char value)
    {
        *str = value;
    }
    ~String()
    {
        delete[] str;
    }
};
int main()
{
    char array[] = "Hello";
    String t1(array);
    String t2(t1);
    t1.set1stChar('A');
    t1.fun(); // t1 -> Aello
    t2.fun(); // t2-> Hello
}
```

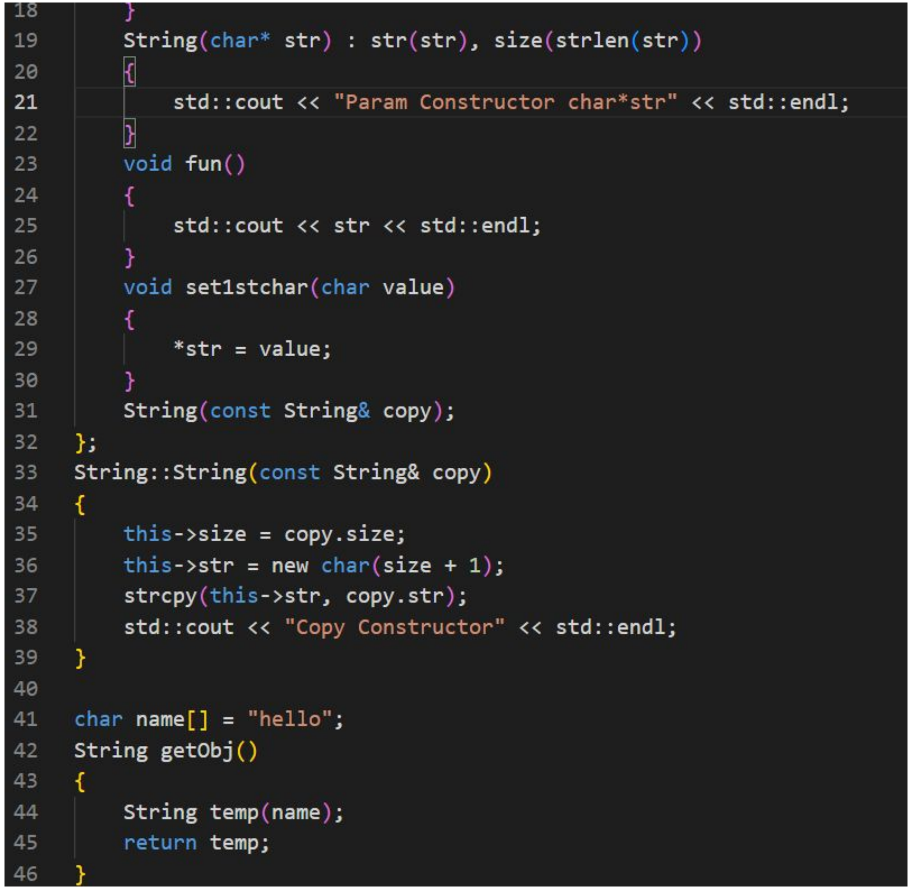
The overall syntex of deep copy



```
    String(const String& copy) // const referance can take temp value
    {
        this->size = copy.size;
        this->str = new char(size + 1);
        strcpy(this->str, copy.str);
    }
};
String getObj()
{
    String temp;
    return temp;
}
int main()
{

    String t2(getObj());
}
```

```
F .text  000000000000005b          String::String(String const&)
F .text  000000000000003e          String::fun()
F .text  0000000000000025          String::String()
```

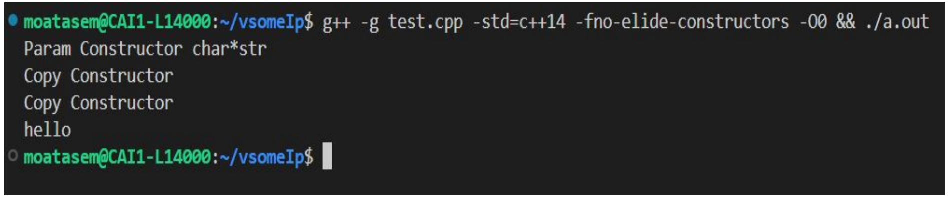
Elide-constructors



```
    }
    String(char* str) : str(str), size(strlen(str))
    {
        std::cout << "Param Constructor char*str" << std::endl;
    }
    void fun()
    {
        std::cout << str << std::endl;
    }
    void set1stchar(char value)
    {
        *str = value;
    }
    String(const String& copy);
};
String::String(const String& copy)
{
    this->size = copy.size;
    this->str = new char(size + 1);
    strcpy(this->str, copy.str);
    std::cout << "Copy Constructor" << std::endl;
}

char name[] = "hello";
String getObj()
{
    String temp(name);
    return temp;
}
```

-fno-elide-constructors

The C++ standard allows an implementation to **omit creating a temporary which is only used to initialize another object of the same type**. Specifying this option disables that optimization, and forces G++ to call the copy constructor in all cases.



```
moatasem@CAI1-L14000:~/vsomeIp$ g++ -g test.cpp -std=c++14 -fno-elide-constructors -O0 && ./a.out
Param Constructor char*str
Copy Constructor
Copy Constructor
hello
moatasem@CAI1-L14000:~/vsomeIp$
```

Operator = copy overload

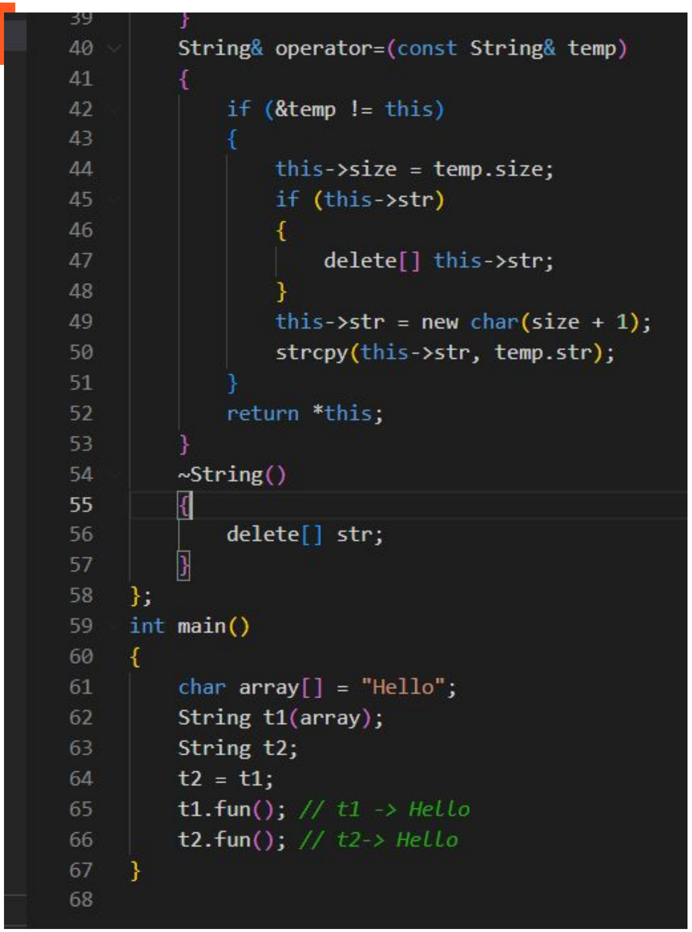
The Rule of Three

The Rule of Three states that if a type ever needs one of the following, then it must have all three.

- copy constructor
- copy assignment
- destructor

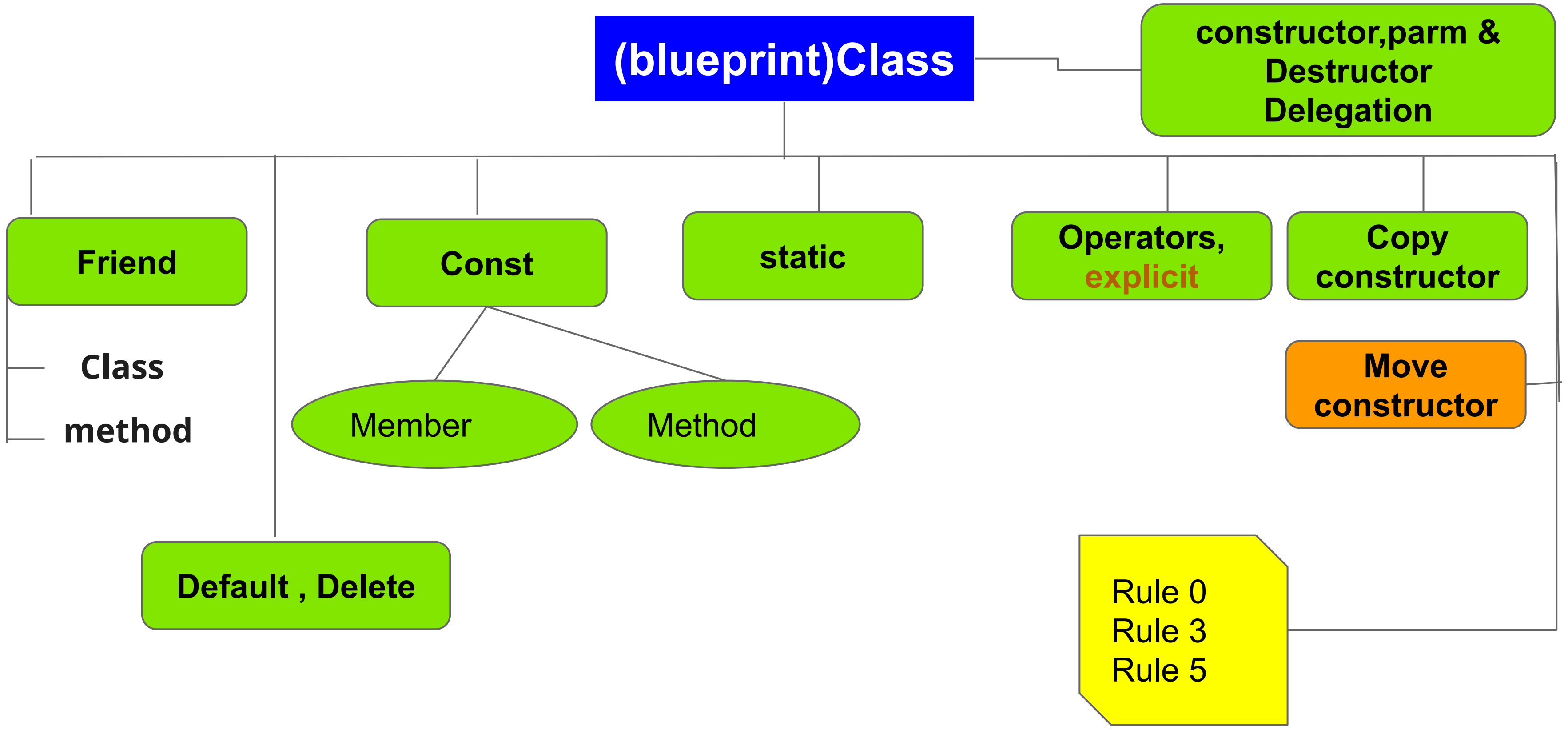
Resource Acquisition Is Initialization

In accordance with RAII principles, the aforementioned functions are usually required when a class is manually managing at least one dynamically allocated resource.



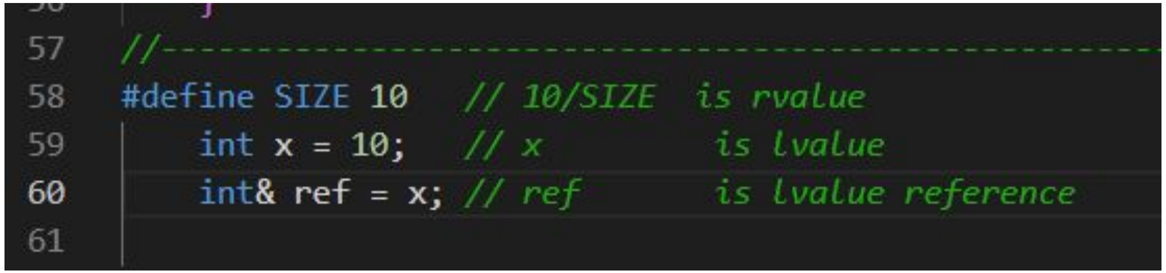
```
    }
    String& operator=(const String& temp)
    {
        if (&temp != this)
        {
            this->size = temp.size;
            if (this->str)
            {
                delete[] this->str;
            }
            this->str = new char(size + 1);
            strcpy(this->str, temp.str);
        }
        return *this;
    }
    ~String()
    {
        delete[] str;
    }
};
int main()
{
    char array[] = "Hello";
    String t1(array);
    String t2;
    t2 = t1;
    t1.fun(); // t1 -> Hello
    t2.fun(); // t2-> Hello
}
```

Features of class

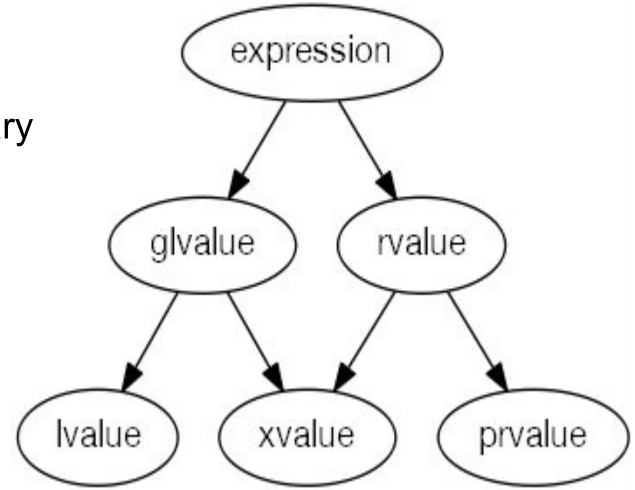


Value Category

Lvalue = has Name / Has Address   Rvalue = No Name / temporary



```
//--------------------------------------------------------------
#define SIZE 10    // 10/SIZE  is rvalue
    int x = 10;    // x         is lvalue
    int& ref = x;  // ref       is lvalue reference

```

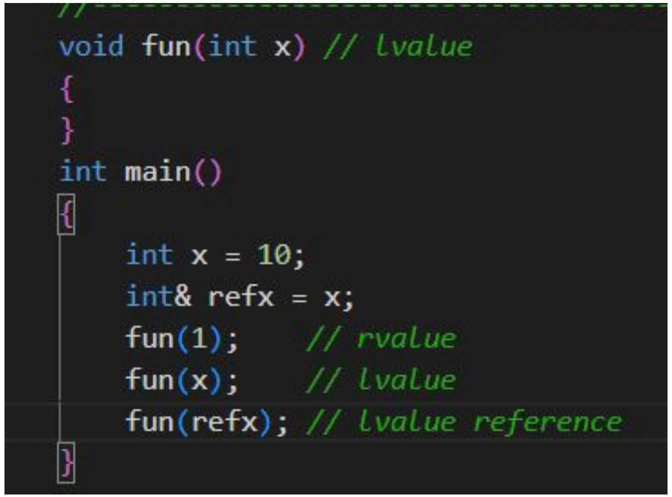


Glvalue : Generalized lValue

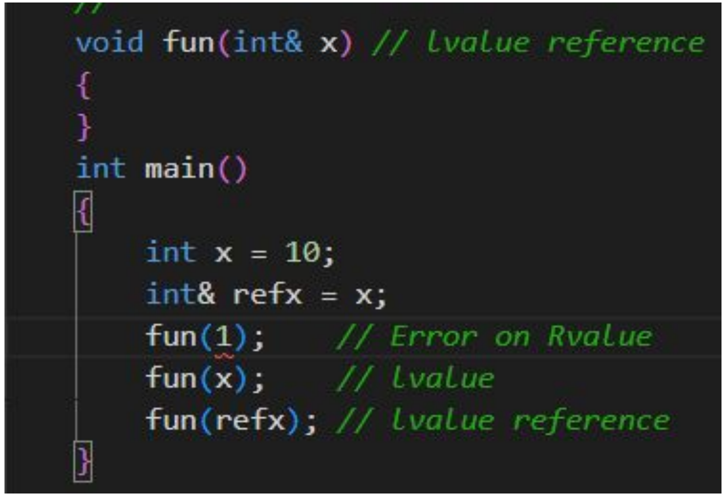
Prvalue : Pure Rvalue

Xvalue : Expiring Lvalue

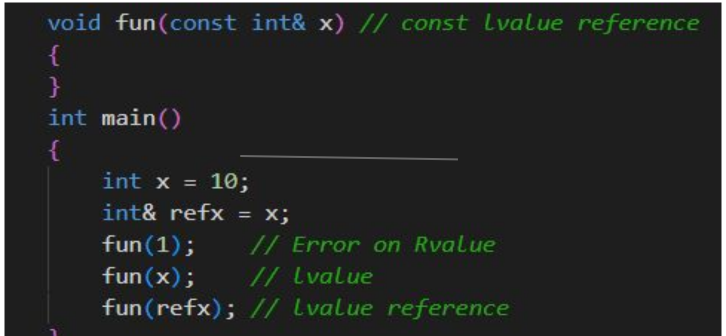
Lvalue Vs Rvalue Reference Vs const Lvalue Reference



```cpp
//------------------------------------
void fun(int x) // lvalue
{
}
int main()
{
    int x = 10;
    int& refx = x;
    fun(1);    // rvalue
    fun(x);    // lvalue
    fun(refx); // lvalue reference
}
```

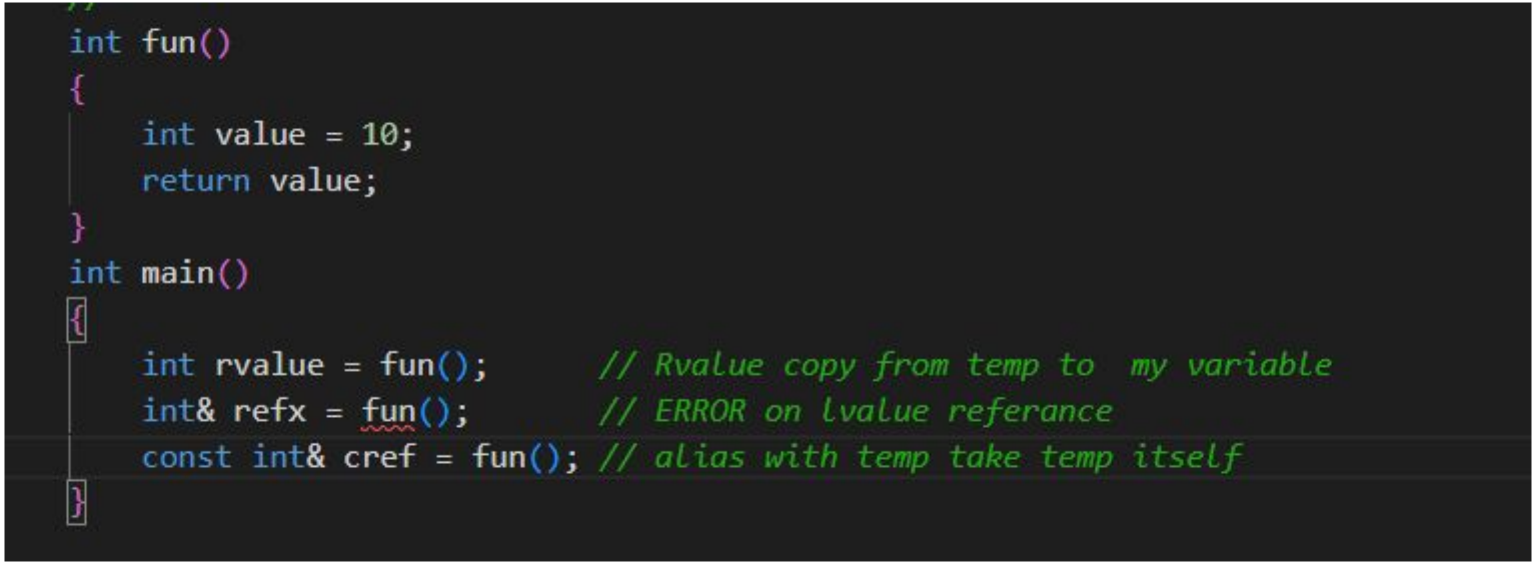


```cpp
//
void fun(int& x) // lvalue reference
{
}
int main()
{
    int x = 10;
    int& refx = x;
    fun(1);    // Error on Rvalue
    fun(x);    // lvalue
    fun(refx); // lvalue reference
}
```



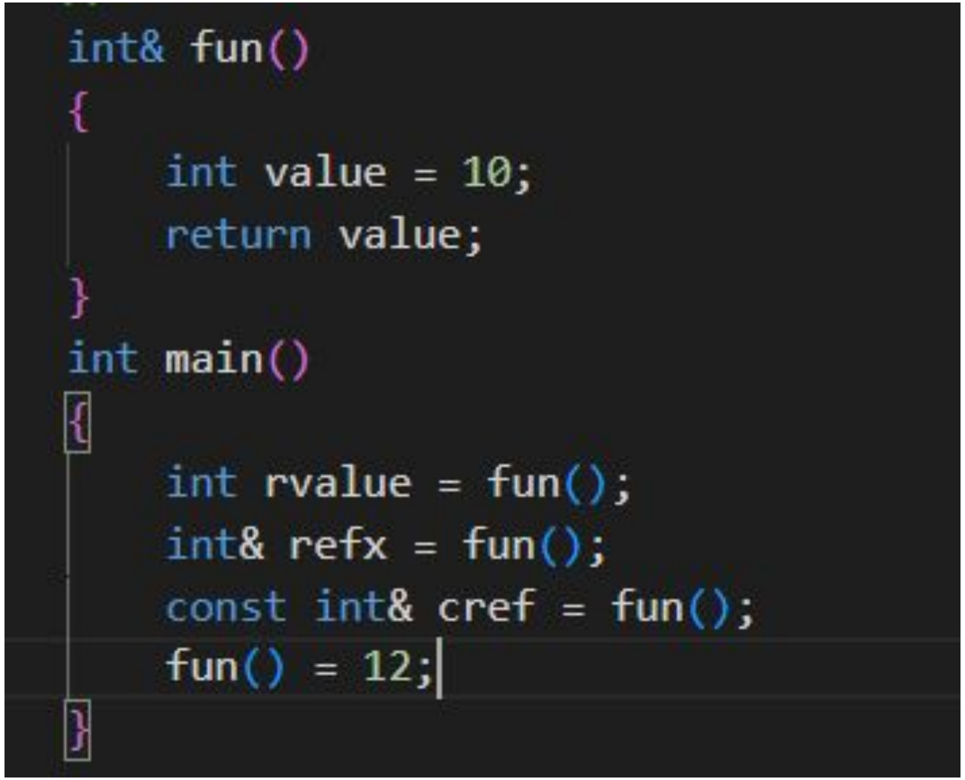
```cpp
void fun(const int& x) // const lvalue reference
{
}
int main()
{
    int x = 10;
    int& refx = x;
    fun(1);    // Error on Rvalue
    fun(x);    // lvalue
    fun(refx); // lvalue reference
```

Return from Function is Rvalue



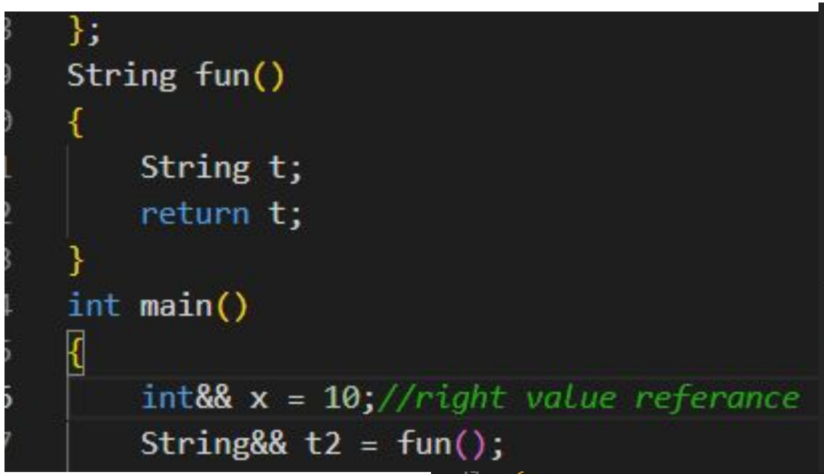
```
int fun()
{
    int value = 10;
    return value;
}
int main()
{
    int rvalue = fun();      // Rvalue copy from temp to  my variable
    int& refx = fun();       // ERROR on lvalue referance
    const int& cref = fun(); // alias with temp take temp itself
}
```

Return Lvalue from Function

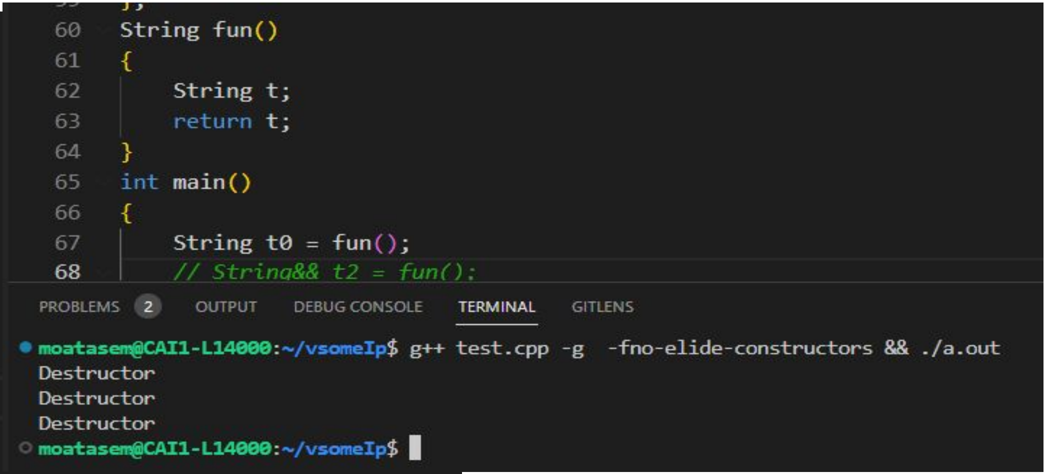


```
int& fun()
{
    int value = 10;
    return value;
}
int main()
{
    int rvalue = fun();
    int& refx = fun();
    const int& cref = fun();
    fun() = 12;
}
```

Right Value Reference

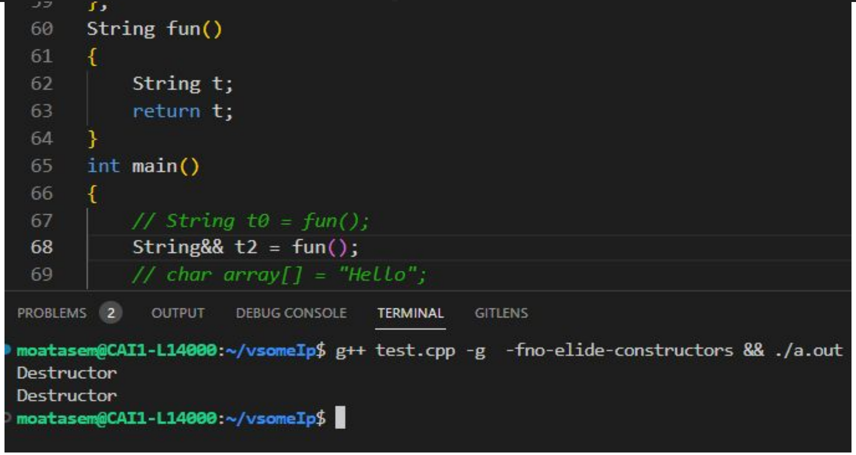


```cpp
};
String fun()
{
    String t;
    return t;
}
int main()
{
    int&& x = 10;//right value referance
    String&& t2 = fun();
```



```cpp
String fun()
{
    String t;
    return t;
}
int main()
{
    String t0 = fun();
    // String&& t2 = fun();
```

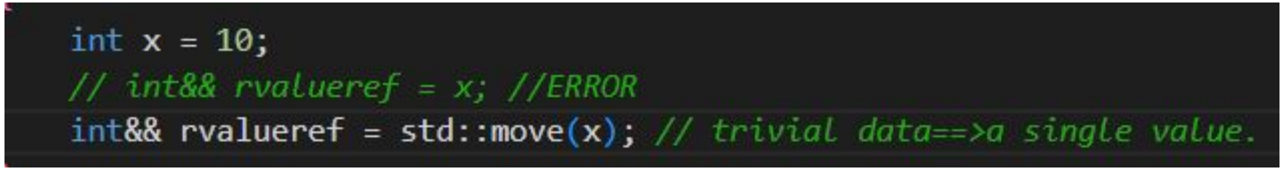
```
moatasem@CAI1-L14000:~/vsomeIp$ g++ test.cpp -g  -fno-elide-constructors && ./a.out
Destructor
Destructor
Destructor
moatasem@CAI1-L14000:~/vsomeIp$
```



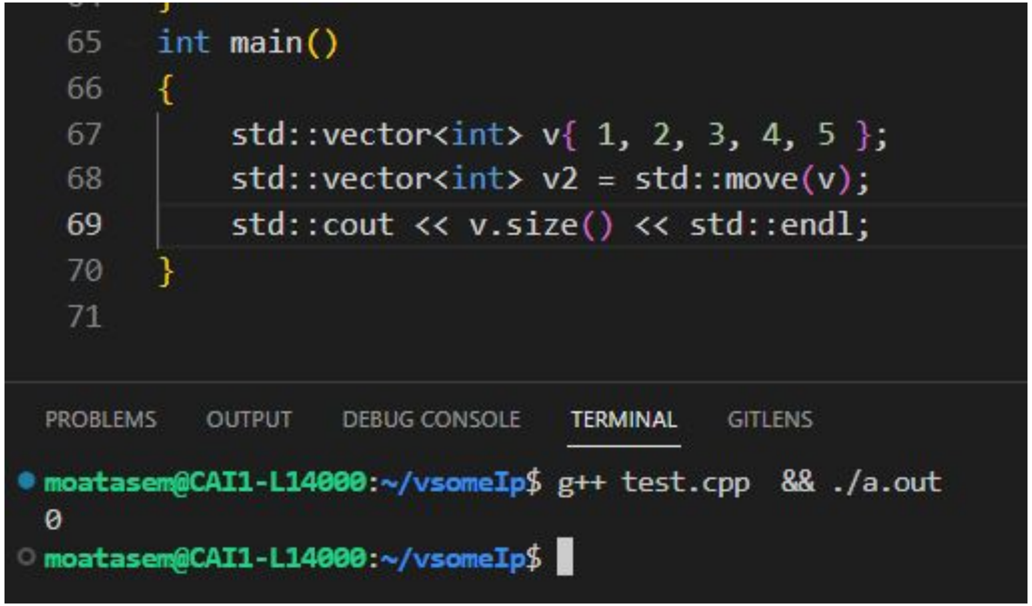
```cpp
String fun()
{
    String t;
    return t;
}
int main()
{
    // String t0 = fun();
    String&& t2 = fun();
    // char array[] = "Hello";
```

```
moatasem@CAI1-L14000:~/vsomeIp$ g++ test.cpp -g  -fno-elide-constructors && ./a.out
Destructor
Destructor
moatasem@CAI1-L14000:~/vsomeIp$
```

How to assign lvalue on Rvalue Reference ? Std::move



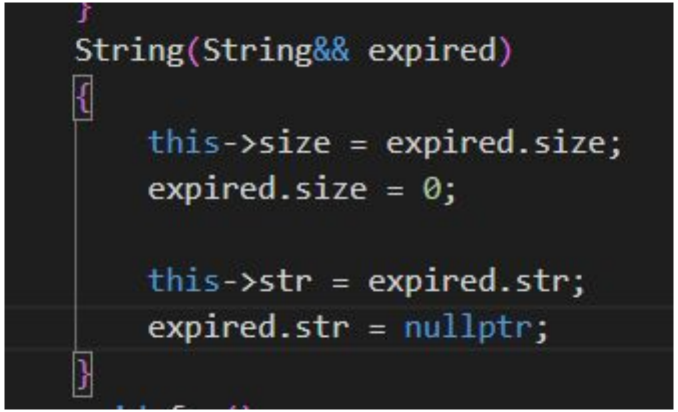
```cpp
int x = 10;
// int&& rvalueref = x; //ERROR
int&& rvalueref = std::move(x); // trivial data==>a single value.
```



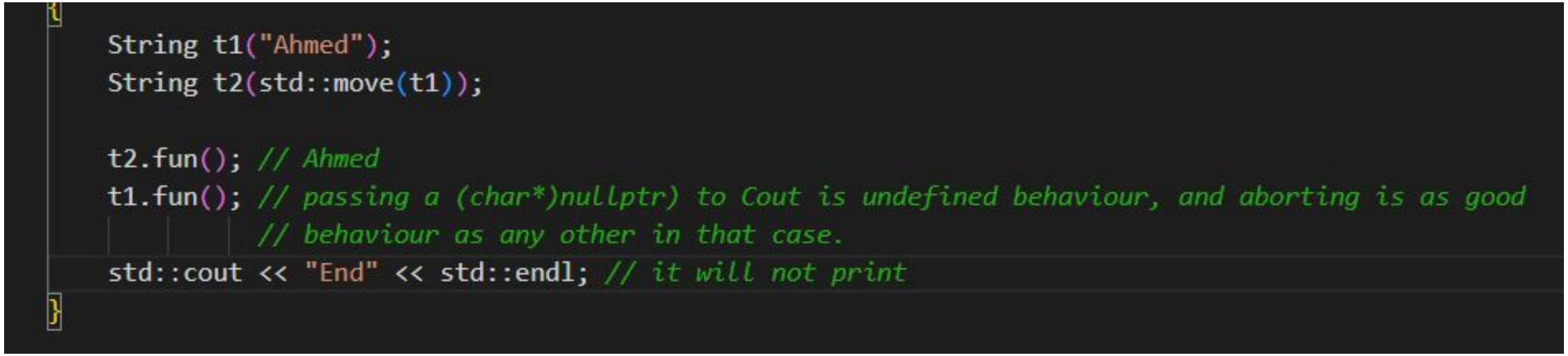
```cpp
int main()
{
    std::vector<int> v{ 1, 2, 3, 4, 5 };
    std::vector<int> v2 = std::move(v);
    std::cout << v.size() << std::endl;
}
```

```
moatasem@CAI1-L14000:~/vsomeIp$ g++ test.cpp  && ./a.out
0
moatasem@CAI1-L14000:~/vsomeIp$
```

Move constructor

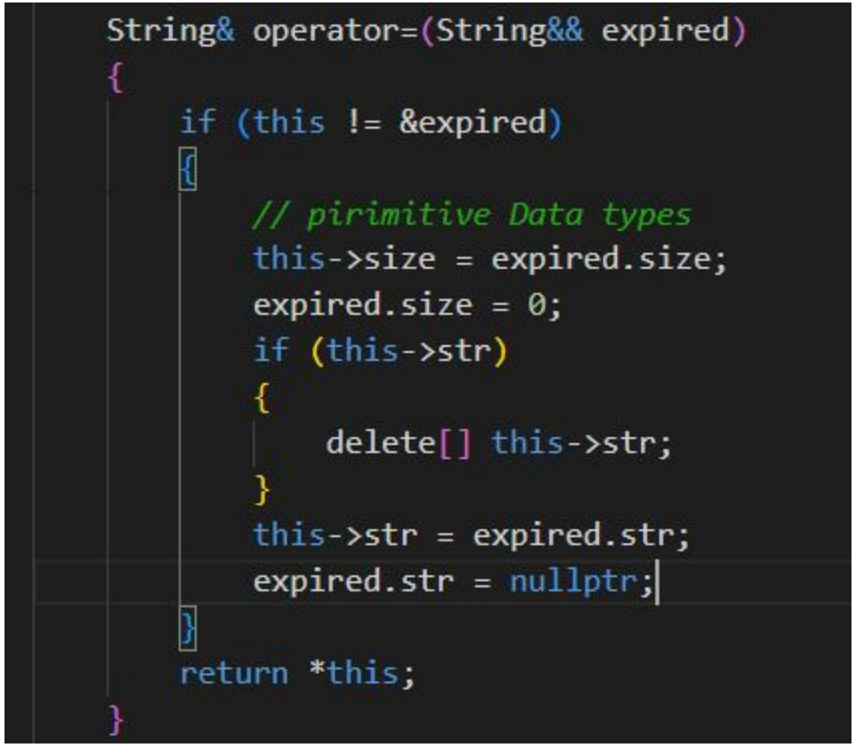


```
    String(String&& expired)
    {
        this->size = expired.size;
        expired.size = 0;

        this->str = expired.str;
        expired.str = nullptr;
    }
```

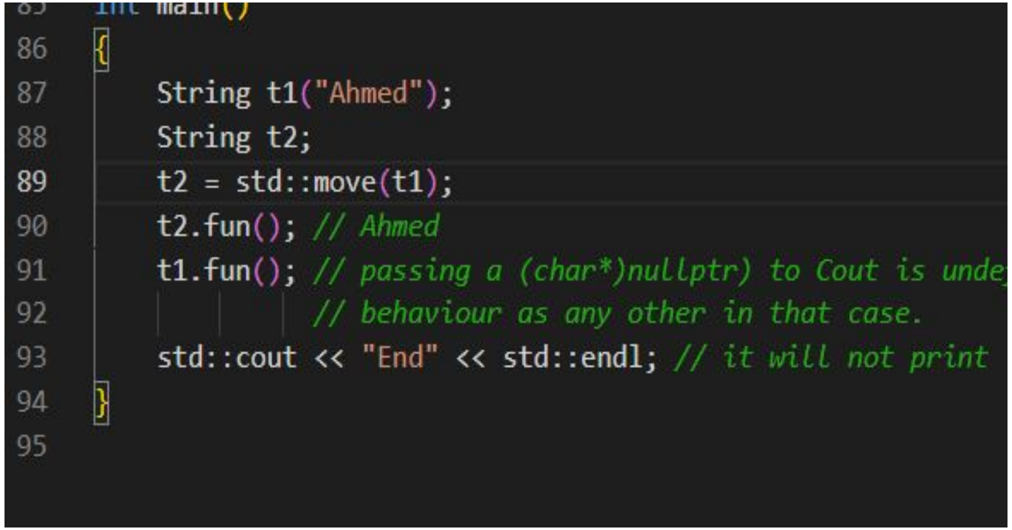


```
{
    String t1("Ahmed");
    String t2(std::move(t1));

    t2.fun(); // Ahmed
    t1.fun(); // passing a (char*)nullptr) to Cout is undefined behaviour, and aborting is as good
              // behaviour as any other in that case.
    std::cout << "End" << std::endl; // it will not print
}
```

operator=



```
String& operator=(String&& expired)
{
    if (this != &expired)
    {
        // pirimitive Data types
        this->size = expired.size;
        expired.size = 0;
        if (this->str)
        {
            delete[] this->str;
        }
        this->str = expired.str;
        expired.str = nullptr;
    }
    return *this;
}
```



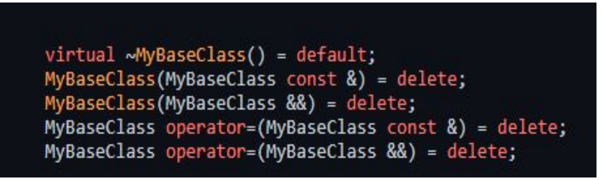
```
int main()
{
    String t1("Ahmed");
    String t2;
    t2 = std::move(t1);
    t2.fun(); // Ahmed
    t1.fun(); // passing a (char*)nullptr to Cout is unde
                // behaviour as any other in that case.
    std::cout << "End" << std::endl; // it will not print
}
```

The Rule of Five

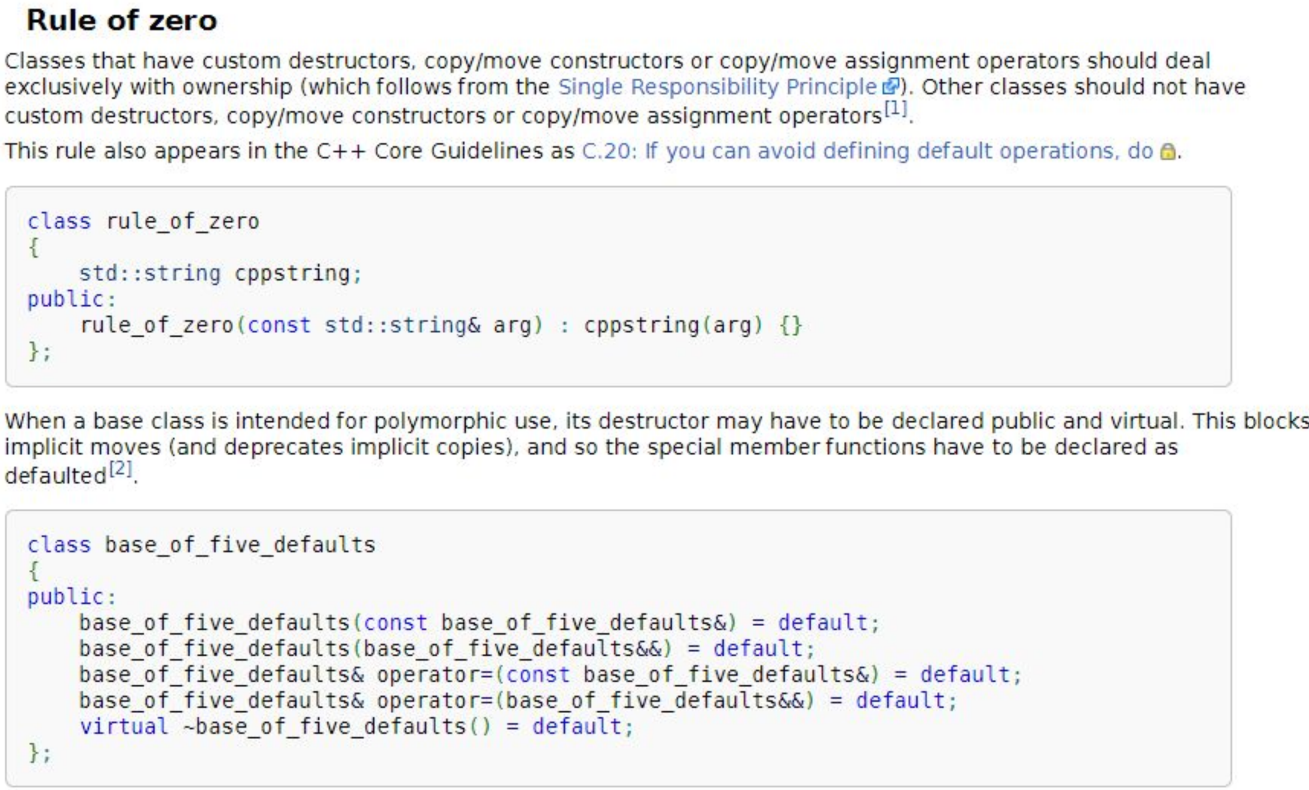
The Rule of Five is a modern extension to the Rule of Three. The Rule of Five states that if a type ever needs one of the following, then it must have all five.

- copy constructor
- copy assignment
- destructor
- move constructor
- move assignment

Rule of Zero



```
virtual ~MyBaseClass() = default;
MyBaseClass(MyBaseClass const &) = delete;
MyBaseClass(MyBaseClass &&) = delete;
MyBaseClass operator=(MyBaseClass const &) = delete;
MyBaseClass operator=(MyBaseClass &&) = delete;
```



Rule of zero

Classes that have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators[1].

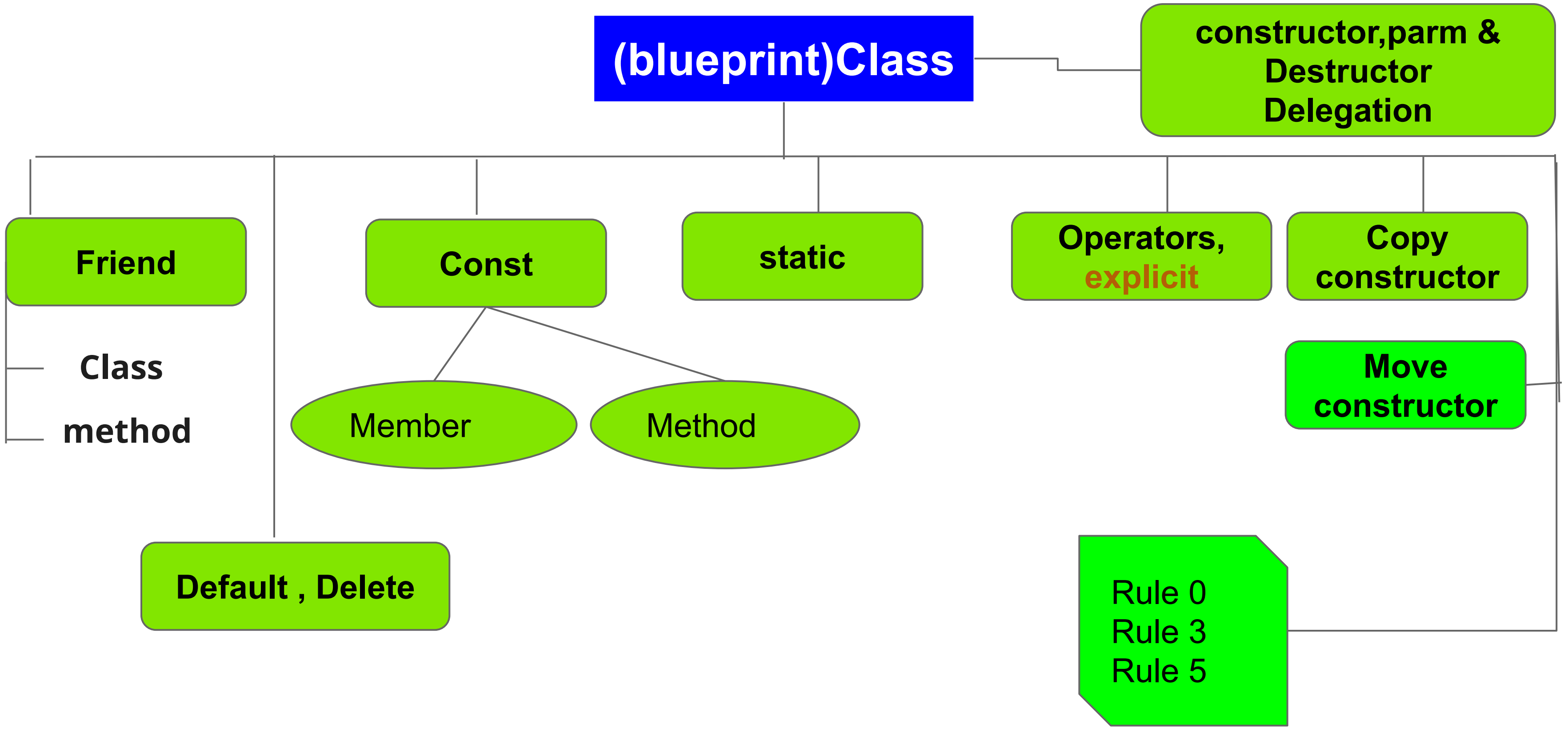
This rule also appears in the C++ Core Guidelines as C.20: If you can avoid defining default operations, do.

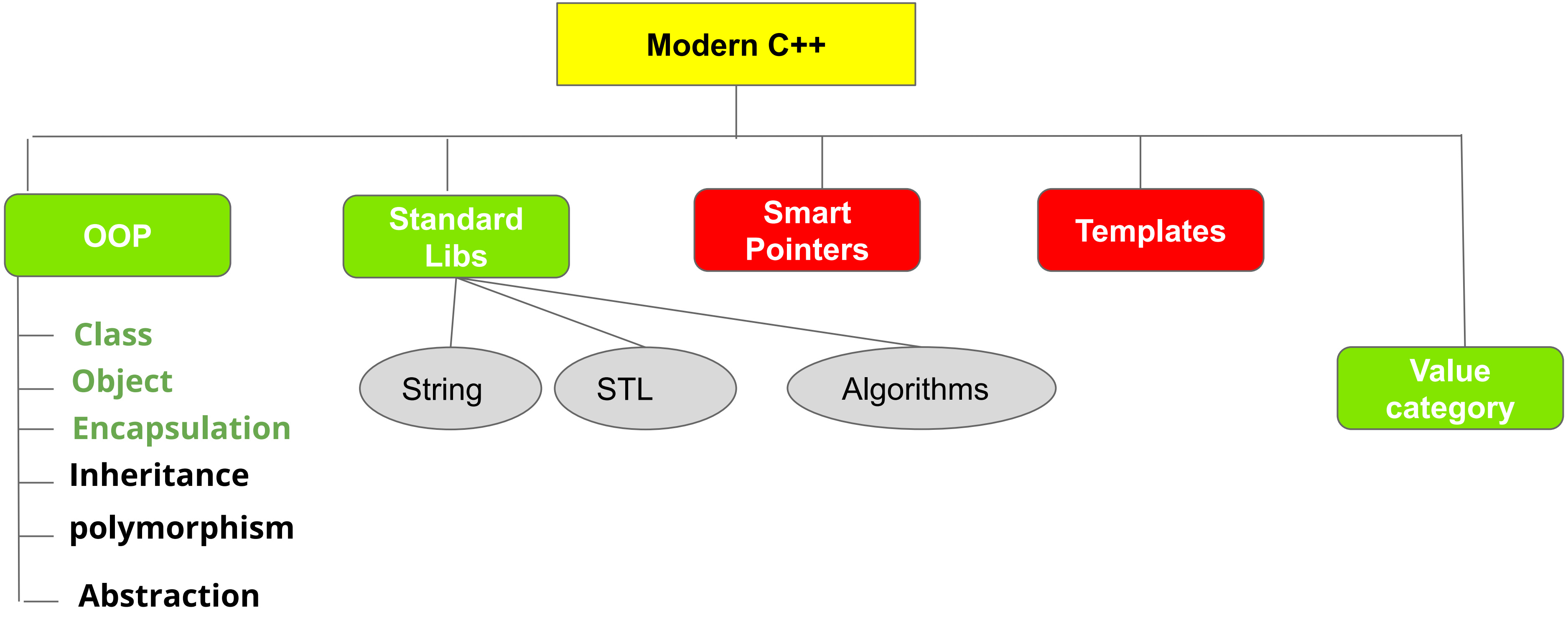
```
class rule_of_zero
{
    std::string cppstring;
public:
    rule_of_zero(const std::string& arg) : cppstring(arg) {}
};
```

When a base class is intended for polymorphic use, its destructor may have to be declared public and virtual. This blocks implicit moves (and deprecates implicit copies), and so the special member functions have to be declared as defaulted[2].

```
class base_of_five_defaults
{
public:
    base_of_five_defaults(const base_of_five_defaults&) = default;
    base_of_five_defaults(base_of_five_defaults&&) = default;
    base_of_five_defaults& operator=(const base_of_five_defaults&) = default;
    base_of_five_defaults& operator=(base_of_five_defaults&&) = default;
    virtual ~base_of_five_defaults() = default;
};
```

Features of class




Modern C++
OOP
Standard Libs
Smart Pointers
Templates
Value category
Class
Object
Encapsulation
Inheritance
polymorphism
Abstraction
String
STL
Algorithms

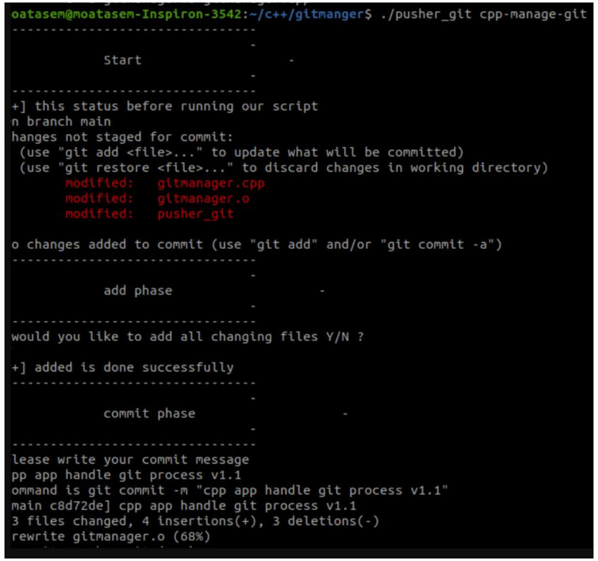
Tasks

1- Create Class behave like string totally

2-Create Class to handle Logs with different Levels and store Msgs as well for dump and clear the buffer

```
LOG::Level(level::warn)<<" forget to close file " ;
LOG::Dump();
LOG::Clear
```

3-git Manager
https://github.com/Moatasem-Elsayed/cpp-manage-git/tree/main



```
oatasem@moatasem-Inspiron-3542:~/c++/gitmanger$ ./pusher_git cpp-manage-git
--------------------------------
                               -
           Start                    -
                               -
--------------------------------
+] this status before running our script
n branch main
hanges not staged for commit:
 (use "git add <file>..." to update what will be committed)
 (use "git restore <file>..." to discard changes in working directory)
       modified:   gitmanager.cpp
       modified:   gitmanager.o
       modified:   pusher_git

o changes added to commit (use "git add" and/or "git commit -a")
--------------------------------
                               -
           add phase                -
                               -
--------------------------------
would you like to add all changing files Y/N ?

+] added is done successfully
--------------------------------
                               -
           commit phase                -
                               -
--------------------------------
lease write your commit message
pp app handle git process v1.1
ommand is git commit -m "cpp app handle git process v1.1"
main c8d72de] cpp app handle git process v1.1
3 files changed, 4 insertions(+), 3 deletions(-)
rewrite gitmanager.o (68%)
```

references

1-http://www.vishalchovatiya.com/

2-MISRA-CPP-2008-STANDARD.pdf

3-https://lefticus.gitbooks.io/cpp-best-practices/content/05-Considering_Maintainability.html

4-www.fluentcpp.com