Intermediate Programming

Day 30

Outline

- Exercise 29
- Initialization and assignment
- Rule of 3
- Review questions

Exercise 29

Overload the << operator for the **Complex** class.

complex.cpp

```
...
std::ostream &operator << ( std::ostream &os , const Complex &c )
{
	return os << c.rel << " + " << c.img << "i";
}
...
```

complex.h

```
...
class Complex
{
	...
	friend std::ostream &operator << ( std::ostream & , const Complex & );
};
```

Exercise 29

Define the copy constructor and overloaded arithmetic operators.

complex.cpp

```
...
Complex::Complex( const Complex &rhs ) : rel( rhs.rel ) , img( rhs.img ) {}
Complex &Complex::operator = ( const Complex &rhs ){ rel=rhs.rel ; img = rhs.img ; return *this; }
Complex Complex::operator + ( const Complex &rhs ) const { return Complex( rel+rhs.rel , img+rhs.img ); }
Complex Complex::operator - ( const Complex &rhs ) const { return Complex( rel-rhs.rel , img-rhs.img ); }
Complex Complex::operator * ( const Complex &rhs ) const
{
	return Complex( rel*rhs.rel - img*rhs.img , rel*rhs.img + img*rhs.rel );
}
Complex Complex::operator * ( const float &rhs ) const { return Complex( rel*rhs , img*rhs ); }
Complex Complex::operator / ( const Complex &rhs ) const
{
	return Complex( rel , -img ) * ( 1. / ( rel*rel + img*img ) );
}
```

Exercise 29

Declare and define the **float** times **Complex** operator.

complex.cpp

```
...
Complex operator * ( const float &s , const Complex &c ){ return c*s; }
```

complex.h

```
...
class Complex
{
        ...
};
Complex operator * ( const float &s , const Complex &c );
```

Outline

- Exercise 29
- Initialization and assignment
- Rule of 3
- Review questions

Initialization and assignment

- In C++ we:
  - **Initialize** a variable if we set its value when we declare it
  - **Assign** a variable if we set its value after we declare it

main.cpp

```
#include "complex.h"

int main(void)
{
	Complex c1;
	Complex c2(c1);
	Complex c3=c1;
	c3 = c2;
	return 0;
}
```

```
>> ./a.out
Default <ctor>
Copy <ctor>
Copy <ctor>
Assign
>>
```

complex.h

```
#ifndef COMPLEX_INCLUDED
#define COMPLEX_INCLUDED
#include <iostream>
class Complex
{
	float rel , img;
public:
	Complex( void ){ std::cout << "Default <ctor>" << std::endl; }
	Complex( const Complex &c ) { std::cout << "Copy <ctor>" << std::endl; }
	Complex &operator = ( const Complex &c ){ std::cout << "Assign" << std::endl ; return *this; }
};
#endif // COMPLEX_INCLUDED
```

Initialization and assignment

Note:
Even though the = operator is used, the value is still set through the copy constructor, not the assignment operator.

complex.h

```
#ifndef COMPLEX_INCLUDED
#define COMPLEX_INCLUDED
#include <iostream>
class Complex
{
	float rel , img;
public:
	Complex( void ){ std::cout << "Default <ctor>" << std::endl; }
	Complex( const Complex &c ) { std::cout << "Copy <ctor>" << std::endl; }
	Complex &operator = ( const Complex &c ){ std::cout << "Assign" << std::endl ; return *this; }
};
#endif // COMPLEX_INCLUDED
```

```
#include "complex.h"

int main(void)
{
	Complex c1;
	Complex c2(c1);
	Complex c3=c1;
	c3 = c2;
	return 0;
}
```

```
>> ./a.out
Default <ctor>
Copy <ctor>
Copy <ctor>
Assign
>>
```

Outline

- Exercise 29
- Initialization and assignment
- Rule of 3
- Review questions

Rule of 3

What is wrong with this array class for dynamic memory allocation?

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
                                    _size(s) , _values(new float[s]) {}
                                    t { return _size; }
                                    i ){ return _values[i]; }

                                    NCLUDED
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      return 0;
}
```

```
>> valgrind --leak-check=full ./main1
...
==1403445== HEAP SUMMARY:
==1403445==     in use at exit: 40 bytes in 1 blocks
==1403445==   total heap usage: 2 allocs, 1 frees, 72,744 bytes allocated
==1403445==
==1403445== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1403445==    at 0x483A582: operator new[](unsigned long) (vg_replace_malloc.c:431)
==1403445==    by 0x40121A: MyArray::MyArray(unsigned long) (my_array.h:9)
==1403445==    by 0x40117E: main (main.cpp:6)
==1403445==
==1403445== LEAK SUMMARY:
==1403445==    definitely lost: 40 bytes in 1 blocks
==1403445==    indirectly lost: 0 bytes in 0 blocks
==1403445==      possibly lost: 0 bytes in 0 blocks
==1403445==    still reachable: 0 bytes in 0 blocks
==1403445==         suppressed: 0 bytes in 0 blocks
==1403445==
==1403445== For lists of detected and suppressed errors, rerun with: -s
==1403445== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      return 0;
}
```

```
>> valgrind --leak-check=full ./main1
...
==1398180== HEAP SUMMARY:
==1398180==     in use at exit: 0 bytes in 0 blocks
==1398180==   total heap usage: 2 allocs, 2 frees, 72,744 bytes allocated
==1398180==
==1398180== All heap blocks were freed -- no leaks are possible
==1398180==
==1398180== For lists of detected and suppressed errors, rerun with: -s
==1398180== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

main.cpp

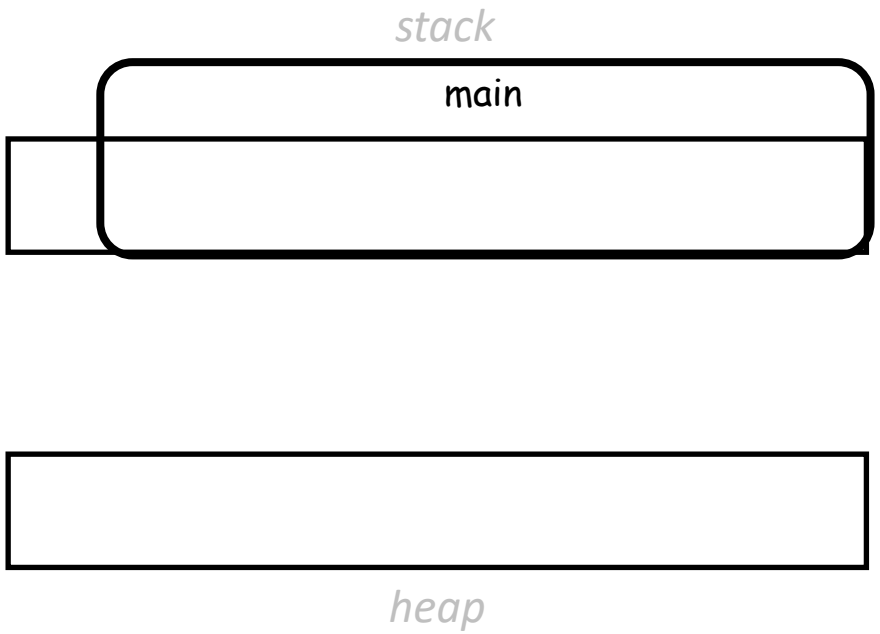
```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

```
>> valgrind --leak-check=full ./main1
...
==1796672== Invalid free() / delete / delete[] / realloc()
==1796672==    at 0x483B59C: operator delete[](void*) (vg_replace_malloc.c:649)
==1796672==    by 0x401290: MyArray::~MyArray() (my_array.h:10)
==1796672==    by 0x4011BC: main (main.cpp:6)
==1796672==  Address 0x4dafc80 is 0 bytes inside a block of size 40 free'd
==1796672==    at 0x483B59C: operator delete[](void*) (vg_replace_malloc.c:649)
==1796672==    by 0x401290: MyArray::~MyArray() (my_array.h:10)
==1796672==    by 0x4011B0: main (main.cpp:7)
==1796672==  Block was alloc'd at
==1796672==    at 0x483A582: operator new[](unsigned long) (vg_replace_malloc.c:431)
==1796672==    by 0x401258: MyArray::MyArray(unsigned long) (my_array.h:9)
==1796672==    by 0x40118F: main (main.cpp:6)
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

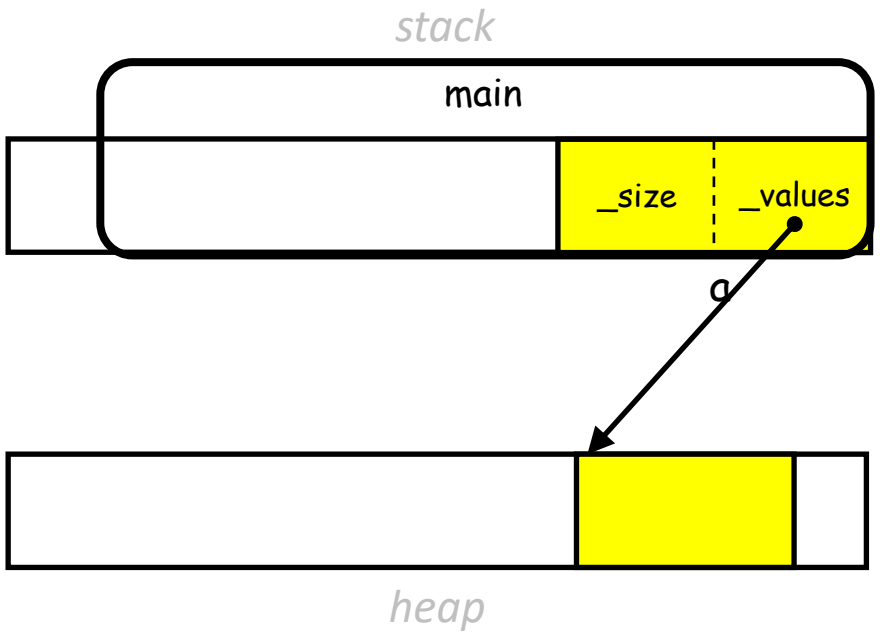
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

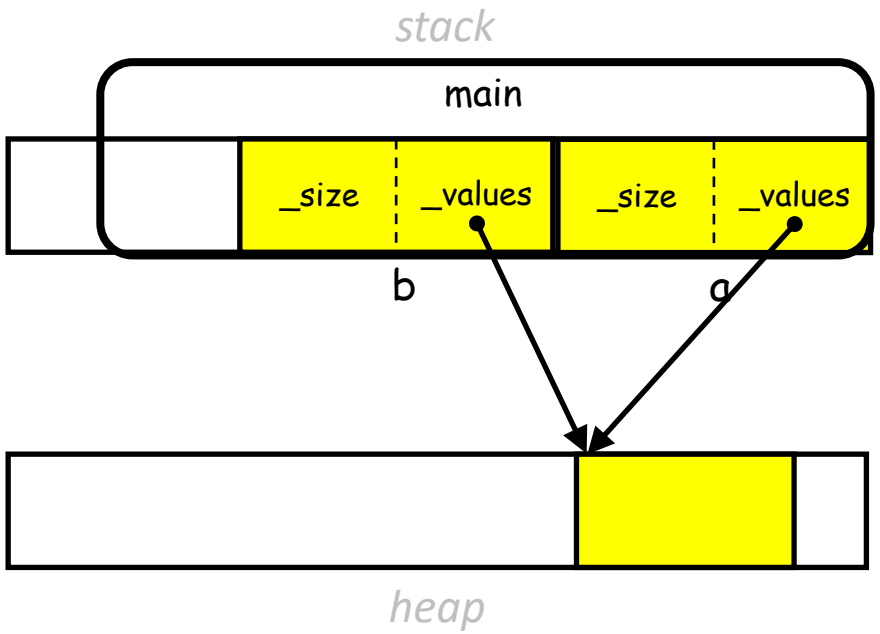
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

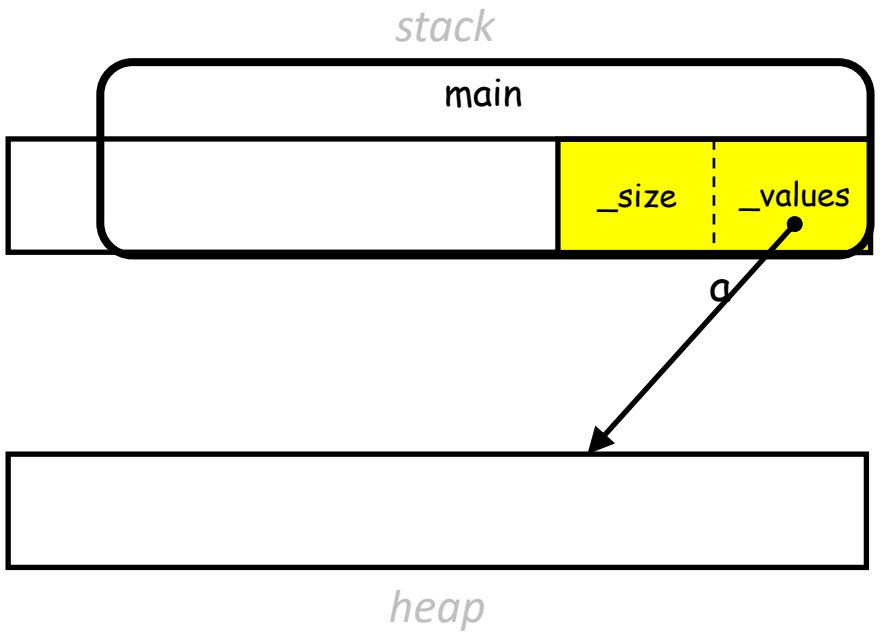
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_INCLUDED
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) :
   {
      _values = new float[ a._size ];
      _size = a._size;
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { dele
   size_t size( void ) cons
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

```
>> valgrind --leak-check=full ./main1
...
==1415557== HEAP SUMMARY:
==1415557==     in use at exit: 0 bytes in 0 blocks
==1415557==   total heap usage: 3 allocs, 3 frees, 72,784 bytes allocated
==1415557==
==1415557== All heap blocks were freed -- no leaks are possible
==1415557==
==1415557== For lists of detected and suppressed errors, rerun with: -s
==1415557== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) cons
   float &operator[] ( int
};
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 );
      MyArray b( a );
      return 0;
}
```

```
>> valgrind --leak-check=full ./main1
...
==1415557== HEAP SUMMARY:
==1415557==     in use at exit: 0 bytes in 0 blocks
==1415557==   total heap usage: 3 allocs, 3 frees, 72,784 bytes allocated
==1415557==
==1415557== All heap blocks were freed -- no leaks are possible
==1415557==
==1415557== For lists of detected and suppressed errors, rerun with: -s
==1415557== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_I
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t { return _size; }
   i ){ return _values[i]; }
```

```
>> valgrind --leak-check=full ./main1
...
==1418592== Invalid free() / delete / delete[] / realloc()
==1418592==    at 0x483B59C: operator delete[](void*) (vg_replace_malloc.c:649)
==1418592==    by 0x4012DE: MyArray::~MyArray() (my_array.h:14)
==1418592==    by 0x4011ED: main (main.cpp:6)
==1418592==  Address 0x4dafc80 is 0 bytes inside a block of size 40 free'd
==1418592==    at 0x483B59C: operator delete[](void*) (vg_replace_malloc.c:649)
==1418592==    by 0x4012DE: MyArray::~MyArray() (my_array.h:14)
==1418592==    by 0x4011E1: main (main.cpp:6)
==1418592==  Block was alloc'd at
==1418592==    at 0x483A582: operator new[](unsigned long) (vg_replace_malloc.c:431)
==1418592==    by 0x4012A6: MyArray::MyArray(unsigned long) (my_array.h:9)
==1418592==    by 0x4011AF: main (main.cpp:6)
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_I
```

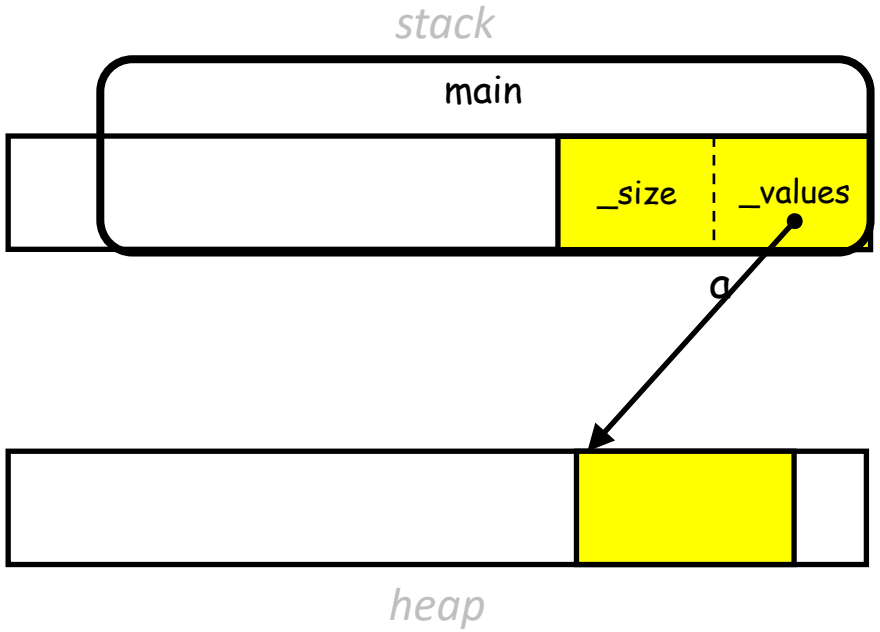
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_I
```

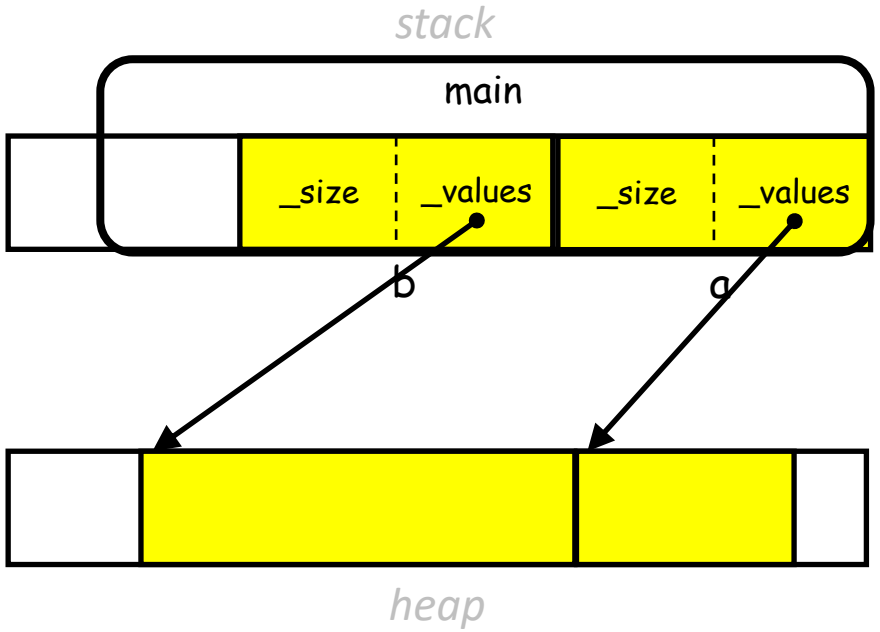
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_I
```

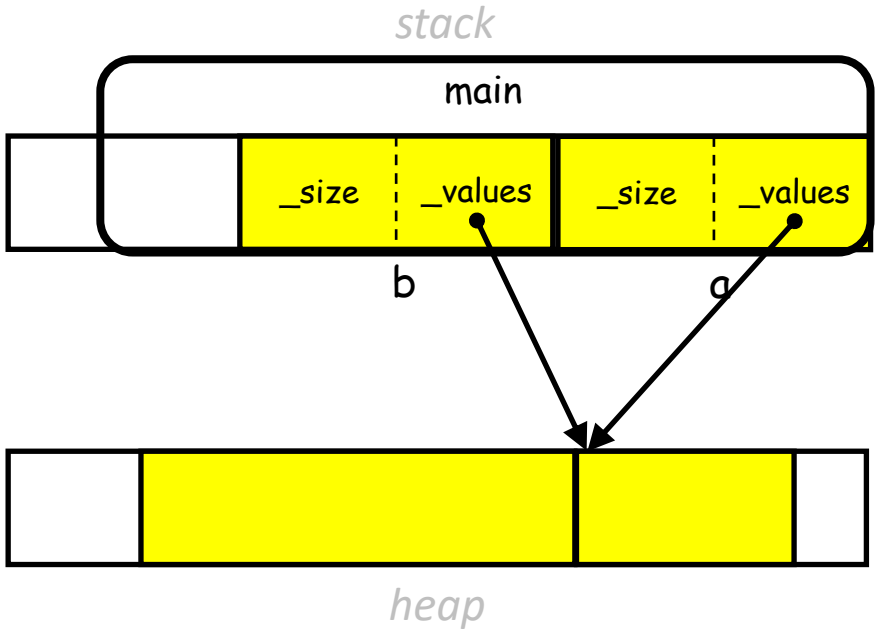
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_I
```

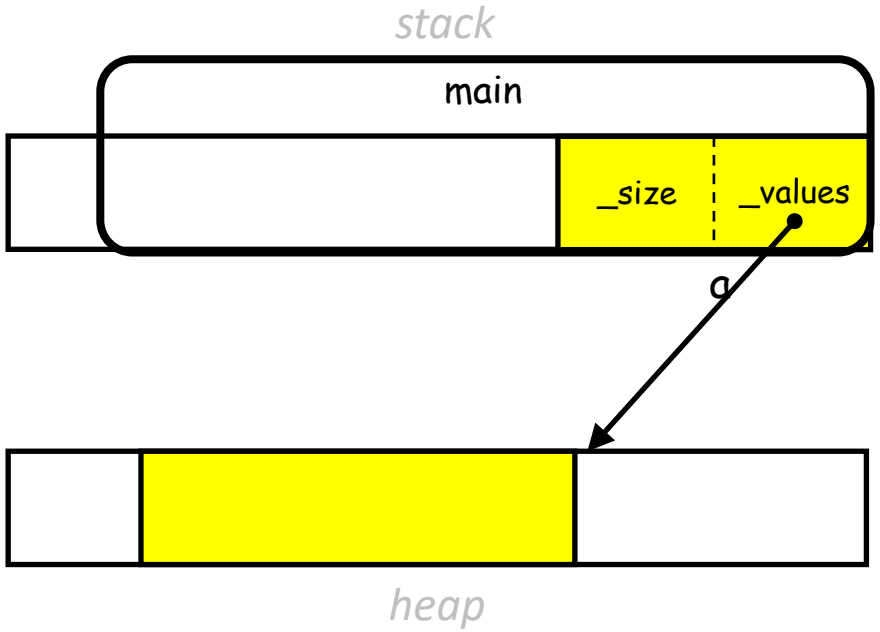
main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor



my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   MyArray( const MyArray &a ) : MyArray( a._size )
   {
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
   }
   ~MyArray( void ) { delete[] _values; }
   size_t size( void ) const { return _size; }
   float &operator[] ( int i ){ return _values[i]; }
};
#endif // MY_ARRAY_I
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

Rule of 3

What is wrong with this array class for dynamic memory allocation?

- Need a destructor
- Need a **deep** copy constructor
- Need a **deep** assignment operator

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray &operator = ( const MyArray &a )
   {
      delete[] _values;
      _values = new float[ a._size ];
      _size = a._size;
      for( size_t i=0 ; i<_size ; i++ ) _values[i] = a._values[i];
      return *this;
   }
   MyArray( size_t s ) : _
```

main.cpp

```
#include <iostream>
#include "my_array.h"

int main(void)
{
      MyArray a( 10 ) , b( 20 );
      b = a;
      return 0;
}
```

```
>> valgrind --leak-check=full ./main1
...
==1415557== HEAP SUMMARY:
==1415557==     in use at exit: 0 bytes in 0 blocks
==1415557==   total heap usage: 3 allocs, 3 frees, 72,784 bytes allocated
==1415557==
==1415557== All heap blocks were freed -- no leaks are possible
==1415557==
==1415557== For lists of detected and suppressed errors, rerun with: -s
==1415557== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Rule of 3

If you have a non-trivial destructor you likely need a non-trivial copy constructor and a non-trivial assignment operator.

Rule of 3

Recall:

The copy constructor is called when passing variables to and (possibly) from a function.

Rule of 3

Recall:

The copy constructor is called when passing variables to and (possibly) from a function.

⇒ Without an appropriate copy constructor this code will have memory troubles because:

1. a._values and b._values point to the same memory on the heap.
2. a._values is deallocated after sum returns.
3. b._values is deallocated after main returns.

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t s
   float &o
};
#endif //
```

main.cpp

```
#include <iostream>
#include "my_array.h"
float sum( MyArray a )
{
      float sum=0;
      for( size_t i=0 ; i<a.size() ; i++ ) sum += a[i];
      return sum;
}
int main(void)
{
      MyArray b( 10 );
      for( size_t i=0 ; i<b.size() ; i++ ) b[i] = 1;
      std::cout << "Sum: " << sum(b) << std::endl;
      return 0;
}
```

Rule of 3

Recall:

The copy constructor is called when passing variables to and (possibly) from a function.

⇒ Without an appropriate copy constructor this code will have memory troubles because:

1. c is initialized with the (default) copy constructor using b so b._values and c._values point to the same location
2. b._values is deallocated after get returns.
3. c._values is deallocated after main returns.

my_array.h

```
#ifndef MY_ARRAY_INCLUDED
#define MY_ARRAY_INCLUDED

class MyArray
{
   size_t _size;
   float *_values;
public:
   MyArray( size_t s ) : _size(s) , _values(new float[s]) {}
   ~MyArray( void ) { delete[] _values; }
   size_t s
   float &o
};
#endif //
```

main.cpp

```
#include <iostream>
#include "my_array.h"
MyArray get( size_t sz )
{
      MyArray a(sz) , b(sz);
      if( sz%2 ) return a;
      else       return b;
}

int main(void)
{
      MyArray c( get( 10 ) );
      return 0;
}
```

Outline

- Exercise 29
- Initialization and assignment
- Rule of 3
- Review questions

Review questions

1. What is difference between initialization and assignment?

Initialization happens when the variable is declared.

Review questions

2. Does the line `f2 = f1;` use initialization or assignment (assume `Foo` is a class and `f1` and `f2` are both of type `Foo`)?

Assignment

Review questions

3. Does the line `Foo f2 = f1;` use initialization or assignment (assume `Foo` is a class and `f1` is of type `Foo`)?

Initialization

Review questions

4. What is a shallow copy and what is a deep copy?

Shallow copy copies pointers. Deep copy allocates memory and copies over values being pointed to.

Review questions

5. What is the rule of 3?

If you need a non-default destructor to release resources, then you will mostly likely need a non-default copy constructor and a non-default assignment operator.

Exercise 30

- Website -> Course Materials -> Exercise 30