Classes

Ivan Sorokin
Сорокин Иван Владимирович

Introduction (1)

- Perhaps the simplest application of classes is grouping of the related data

```
struct point
{
    double x, y, z;
};
```

Member functions (1)

```
struct complex
{
    double re;
    double im;
};

void conjugate(complex* c)
{
    c->im = -c->im;
}
```

Member functions (2)

```
struct complex
{
    double re;
    double im;
};

void conjugate(complex* c)
{
    c->im = -c->im;
}
```

```
struct complex
{
    void conjugate()
    {
        im = -im;
    }

    double re;
    double im;
};
```

Member functions (3)

```
struct complex
{
    double re;
    double im;
};

void conjugate(complex* c)
{
    c->im = -c->im;
}
```

```
struct complex
{
    void conjugate(complex* this)
    {
        this->im = -this->im;
    }

    double re;
    double im;
};
```

Member functions (4)

```
struct complex
{
    double re;
    double im;
};

void conjugate(complex* c)
{
    c->im = -c->im;
}
```

```
struct complex
{
    void conjugate()
    {
        im = -im;
    }

    double re;
    double im;
};
```

Member functions (4)

```
struct complex
{
private:
    double re;
    double im;
};

void conjugate(complex* c)
{
    c->im = -c->im; // ERROR
}
```

```
struct complex
{
    void conjugate()
    {
        im = -im; // OK
    }

private:
    double re;
    double im;
};
```

Class invariant (1)

```
struct node
{
    node* parent;
    node* left;
    node* right;
    int32_t value;
};

struct binary_search_tree
{
    node* root;
};
```

Invariant:

Class invariant (2)

```
struct node
{
    node* parent;
    node* left;
    node* right;
    int32_t value;
};

struct binary_search_tree
{
    node* root;
};
```

```
Invariant:

if (left) assert(left->parent == this);
if (right) assert(right->parent == this);
```

Class invariant (3)

```
struct node
{
    node* parent;
    node* left;
    node* right;
    int32_t value;
};

struct binary_search_tree
{
    node* root;
};
```

```
Invariant:

if (left) assert(left->parent == this);
if (right) assert(right->parent == this);

for each node n in the left subtree:
    assert(n.value < value);
for each node n in the right subtree:
    assert(n.value > value);
```

Class invariant (4)

```
struct node
{
    node* prev;
    node* next;
    int32_t value;
};

struct doubly_linked_list
{
    node* first;
};
```

Invariant:

Class invariant (5)

```
struct node
{
    node* prev;
    node* next;
    int32_t value;
};

struct doubly_linked_list
{
    node* first;
};
```

```
Invariant:

assert(prev->next == this);
assert(next->prev == this);
```

Class invariant (6)

```
struct node
{
    node* next;
    int32_t value;
};

struct singly_linked_list
{
    node* first;
};
```

Invariant:

Class invariant (7)

```
struct node
{
    node* next;
    int32_t value;
};

struct singly_linked_list
{
    node* first;
};
```

```
Invariant:

No loops
```

Class invariant (8)

```
struct rational
{
    int32_t num;
    int32_t denom;
};
```

Invariant:

Class invariant (9)

```
struct rational
{
    int32_t num;
    int32_t denom;
};
```

```
Invariant:

denom != 0
```

Class invariant (10)

```
struct rational
{
    int32_t num;
    int32_t denom;
};
```

```
Invariant:

1. denom != 0

2. denom > 0
```

Class invariant (11)

```
struct rational
{
    int32_t num;
    int32_t denom;
};
```

```
Invariant:

1. denom != 0

2. denom > 0

3. denom > 0 && abs(gcd(num, denom)) == 1
```

Class invariant (12)

```
struct string
{
    char* data;
    size_t size;
    size_t capacity;
};
```

Invariant:

Class invariant (13)

```
struct string
{
    char* data;
    size_t size;
    size_t capacity;
};
```

```
Invariant:

size <= capacity
```

Class invariant (14)

```
struct string
{
    char* data;
    size_t size;
    size_t capacity;
};
```

```
Invariant:

size <= capacity


1. size != 0 <=> data != nullptr
2. if (size != 0) data != nullptr
```

Class invariant (15)

```
std::vector<int32_t> v;

for (...)
{
    // fill v
    // work

    v.clear();
}
```

```
v.shrink_to_fit();
```

Class invariant (17)

```
struct string
{
    char* data;
    size_t size;
    size_t capacity;
};
```

```
Invariant:

size <= capacity

if (size != 0) data != nullptr
```

Class invariant (16)

```
struct string
{
    char* data;
    size_t size;
    size_t capacity;
};
```

```
Invariant:

size <= capacity

if (size != 0) data != nullptr
capacity != 0 <=> data != nullptr
```

Class invariant (18)

```
struct string
{
    char* data;
    size_t size;
    size_t capacity;
};
```

```
Invariant:

size <= capacity

if (size != 0) data != nullptr
capacity != 0 <=> data != nullptr

data is a pointer to a buffer of size capacity
data[0] ... data[size - 1] are initialized
```

Implementation hiding (1)

```
struct complex
{
    double re;
    double im;
};
```

Implementation hiding (2)

```
struct complex
{
    double re;
    double im;
};

struct complex
{
    double arg;
    double norm;
};
```

Implementation hiding (3)

```
struct complex
{
    double re;
    double im;
};

struct polar_complex
{
    double arg;
    double norm;
};
```

Implementation hiding (4)

```
struct complex
{
    double re;
    double im;
};

struct polar_complex
{
    double arg;
    double norm;
};
```

```
struct complex
{
    void conjugate()
    {
        __imag__ repr = -__imag__ repr;
    }

    _Complex double repr;
};
```

Accessibility

- Accessibility can be used
  - To protect class invariant
  - To hide implementation details

Constructors (1)

- Constructors are the first functions that are called when object is created.
  - They are called automatically by the compiler when the object is created.
  - They are needed to establish the invariant of the class.

```
struct string
{
    string(char const* text)
    {
        size = strlen(text);
        capacity = size;
        data = static_cast<char*>(malloc(capacity + 1));
        memcpy(data, text, size + 1);
    }

private:
    char* data;
    size_t size;
    size_t capacity;
};
```

```
int main()
{
    string s("Hello, world!");
}
```

Constructors (2)

- Constructors with no arguments are called default constructors.
  - An empty default constructor is generated automatically unless the class has other constructors.

```
struct string
{
    string()
    {
        size = 0;
        capacity = 0
        data = strdup("");
    }

private:
    char* data;
    size_t size;
    size_t capacity;
};
```

```
int main()
{
    string s;
}
```

Constructors (3)

- Please we aware of a syntax ambiguity between constructor and function declaration.
  - Rule: if the text can be interpreted as a function declaration then it is a function otherwise a variable initializer.

```
string a;          // variable
string b("Hello"); // variable

string c();        // function declaration
```

Constructors (4)

- Constructors with one parameter are called converting constructors.
  - They can be used to convert a object of one type to another.

```
struct string
{
    string(char const* text);
};

void foo(string);

int main()
{
    foo("Hello");
}
```

```
struct complex
{
    complex(double re,
            double im = 0.);
};

void bar(complex);

int main()
{
    bar(42.);
}
```

Constructors (5)

- Sometimes this implicit conversion is undesirable.
  - The declaration specifier explicit can be used to suppress implicit conversions.

```
struct vector
{
    vector(size_t size);
};

void foo(vector);

int main()
{
    foo(42);
}
```

```
struct vector
{
    explicit vector(size_t size);
};

void foo(vector);

int main()
{
    foo(42); // ERROR
}
```

Constructors (6)

```
void foo(mytype);

mytype x(42);
mytype y = 42;                  // implicit

foo(static_cast<mytype>(42));
foo(42);                        // implicit
```

Object lifetime (1)

```
#include <cstdlib>
#include <cstring>

struct string
{
    string(char const* text)
    {
        size = strlen(text);
        capacity = size;
        data = static_cast<char*>(malloc(capacity + 1));
        memcpy(data, text, size + 1);
    }

private:
    char* data;
    size_t size;
    size_t capacity;
};

int main()
{
    string str("Hello");
}
```

Object lifetime (2)

```
$ g++ -g -fsanitize=address main.cpp
$ ./a.out

=================================================================
==437==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 6 byte(s) in 1 object(s) allocated from:
    #0 0x7f8ec7c3f867 in __interceptor_malloc
../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x5604e801c46b in string::string(char const*) /home/ivan/main.cpp:10
    #2 0x5604e801c318 in main /home/ivan/main.cpp:22
    #3 0x7f8ec765bd8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: 6 byte(s) leaked in 1 allocation(s).
```

Object lifetime (3)

Destructors are the opposite of constructors. They are executed automatically by the compiler when the object is no longer needed.

```
struct string
{
    string(char const* text)
    {
        size = strlen(text);
        capacity = size;
        data = static_cast<char*>(malloc(capacity + 1));
        memcpy(data, text, size + 1);
    }

    ~string()
    {
        free(data);
    }

private:
    char* data;
    size_t size;
    size_t capacity;
};
```

Object lifetime (4)

- Global variables are initialized before main() is entered. They are destroyed after main() is exited.
- Local variables are initialized when their line is executed and are destroyed their compound statement is exited.
  - Normally it is when } is executed, but return, break, continue, throw, goto can exit from a compount statement.

Object lifetime (5)

```
int main()
{
    mytype a;
    mytype b;

    if (...)
    {
        mytype c;
        mytype d;
        // ...
    }

    // ...
}
```

Object lifetime (6)

```
void test()
{
    // ...

    {
        std::lock_guard m(some_mutex);
        // ...
    }

    // ...
}
```

Object lifetime (7)

```
struct complex
{
    complex(double re, double im);
};

void bar(complex);

int main()
{
    bar(complex(0.866, 0.5));
}
```

Object lifetime (7)

- Global variables are initialized before main() is entered. They are destroyed after main() is exited.
- Local variables are initialized when their line is executed and are destroyed their compound statement is exited.
  - Normally it is when } is executed, but return, break, continue, throw, goto can exit from a compount statement.
- Temporary objects are initialized when the expression is executed and are destroyed at the end of the statement (;).

Object lifetime (8)

```
string foo();

int main()
{
    puts(foo().c_str());
}
```

Const member functions (1)

```
#include <cmath>

struct complex
{
    double re;
    double im;
};

double norm(complex const* c)
{
    return hypot(c->re, c->im);
}
```

```
#include <cmath>

struct complex
{
    double norm()
    {
        return hypot(re, im);
    }

    double re;
    double im;
};
```

Const member functions (2)

```
#include <cmath>

struct complex
{
    double re;
    double im;
};

double norm(complex const* c)
{
    return hypot(c->re, c->im);
}
```

```
#include <cmath>

struct complex
{
    double norm() const
    {
        return hypot(re, im);
    }

    double re;
    double im;
};
```

Const member functions (3)

```
#include <cmath>

struct complex
{
    double norm() /*const*/
    {
        return hypot(re, im);
    }

    double re;
    double im;
};
```

```
complex const I = {0, 1};

int main()
{
    I.norm();
}
```

```
$ g++ 1.cpp
1.cpp: In function ‘int main()’:
1.cpp:18:11: error: passing ‘const complex’ as ‘this’ argument discards qualifiers
[-fpermissive]
   18 |     I.norm();
      |     ~~~~~~^~
1.cpp:5:12: note:   in call to ‘double complex::norm()’
    5 |     double norm() /*const*/
      |            ^~~~
```

Operator overloading (1)

```
sum(product(difference(1, t), v0), product(t, v1))
```

Operator overloading (2)

```
sum(product(difference(1, t), v0), product(t, v1))
```

```
(1 - t) * v0 + t * v1
```

Operator overloading (3)

```
struct complex
{
    complex(double re, double im);
    double re, im;
};

complex operator+(complex a, complex b)
{
    return complex(a.re + b.re, a.im + b.im);
}

complex operator-(complex a, complex b)
{
    return complex(a.re + b.re, a.im + b.im);
}

complex operator*(complex a, complex b)
{
    return complex(a.re * b.re - a.im * b.im,
                   a.re * b.im + a.im * b.re);
}

complex operator/(complex a, complex b)
{
    double re = a.re * b.re + a.im * b.im;
    double im = a.im * b.re - a.re * b.im;
    double scale = 1. / (b.re * b.re + b.im * b.im);
    return complex(re * scale, im * scale);
}
```

Operator overloading (4)

```
void test()
{
    complex x, y;
    complex c1 = x + y;
    complex c2 = operator+(x, y);
}
```

Operator overloading (5)

```
complex operator+=(complex a, complex b);

void test()
{
    complex x, y;
    x += y;  // problem: x is unchanged
}
```

Operator overloading (6)

```
complex operator+=(complex* a, complex b);

void test()
{
    complex x, y;
    &x += y;
}
```

Operator overloading (7)

```
int main()
{
    int x, y;
    int* p = &x;

    int z = *p;
    *p = 42;

    p = &y;
}
```

Operator overloading (8)

```
int main()
{
    int x, y;
    int* p = &x;

    int z = *p;
    *p = 42;

    p = &y;
}
```

```
int main()
{
    int x, y;
    int& r = x;

    int z = r;
    r = 42;

    // impossible
}
```

Operator overloading (9)

```
complex operator+=(complex& a, complex b)
{
    a.re += b.re;
    a.im += b.im;
    return a;
}

int main()
{
    complex x, y;
    x += y;
}
```

Operator overloading (10)

```
complex operator+=(complex& a, complex b)
{
    a.re += b.re;
    a.im += b.im;
    return a;
}

int main()
{
    complex x, y, z;
    (x += y) += z; // ERROR
}
```

Operator overloading (11)

```
complex& operator+=(complex& a, complex b)
{
    a.re += b.re;
    a.im += b.im;
    return a;
}

int main()
{
    complex x, y, z;
    (x += y) += z;
}
```

Operator overloading (12)

```
T operator+(T const& a, T const& b);
T operator-(T const& a, T const& b);
T operator*(T const& a, T const& b);
T operator/(T const& a, T const& b);
T operator%(T const& a, T const& b);


T& operator+=(T& a, T const& b);
T& operator-=(T& a, T const& b);
T& operator*=(T& a, T const& b);
T& operator/=(T& a, T const& b);
T& operator%=(T& a, T const& b);
```

Operator overloading (13)

```
T operator+(T const& a, T const& b);
T operator-(T const& a, T const& b);
T operator*(T const& a, T const& b);
T operator/(T const& a, T const& b);
T operator%(T const& a, T const& b);

T& operator+=(T& a, T const& b);
T& operator-=(T& a, T const& b);
T& operator*=(T& a, T const& b);
T& operator/=(T& a, T const& b);
T& operator%=(T& a, T const& b);

T operator+(T const& a);
T operator-(T const& a);
```

Operator overloading (14)

```
T& operator++(T& a); // prefix
T operator++(T& a, int); // postfix
T& operator--(T& a); // prefix
T operator--(T& a, int); // postfix
```

Operator overloading (15)

```
T& operator++(T& a); // prefix
T operator++(T& a, int); // postfix
T& operator--(T& a); // prefix
T operator--(T& a, int); // postfix


bool operator<(T const& a, T const& b);
bool operator<=(T const& a, T const& b);
bool operator==(T const& a, T const& b);
bool operator!=(T const& a, T const& b);
bool operator>(T const& a, T const& b);
bool operator>=(T const& a, T const& b);
```

Operator overloading (16)

```
struct mytype
{
    mytype& operator+=(mytype const& b);

    mytype& operator++();
    mytype& operator++(int);
};


mytype foo();

void test(mytype x)
{
    foo() += x; // OK, but should be ERROR

    ++foo();    // OK, but should be ERROR
    foo()++;    // OK, but should be ERROR
}
```

Operator overloading (17)

```
struct mytype
{
    mytype& operator+=(mytype const& b);
};



mytype foo();

void test(mytype x)
{
    foo() += x;
    foo().operator+=(x);
}
```

```
struct mytype
{};

mytype& operator+=(mytype& a, mytype const& b);


mytype foo();

void test(mytype x)
{
    foo() += x;
    operator+=(foo(), x);
}
```

Operator overloading (16)

```
struct mytype
{
    mytype& operator+=(mytype const& b) &;

    mytype& operator++() &;
    mytype& operator++(int) &;
};


mytype foo();

void test(mytype x)
{
    foo() += x; // ERROR

    ++foo();    // ERROR
    foo()++;    // ERROR
}
```

Copy constructor (1)

```
#include <cstdlib>
#include <cstring>

struct string
{
    string(char const* text)
    {
        size = strlen(text);
        capacity = size;
        data = static_cast<char*>(malloc(capacity + 1));
        memcpy(data, text, size + 1);
    }

    ~string()
    {
        free(data);
    }

private:
    char* data;
    size_t size;
    size_t capacity;
};
```

```
int main()
{
    string s("Hello");
    string t = s;
}
```

Copy constructor (2)

```
$ g++ -g -fsanitize=address main.cpp
$ ./a.out
=================================================================
==463==ERROR: AddressSanitizer: attempting double-free on 0x602000000010 in thread T0:
    #0 0x7f3b05ee2517 in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x5576100a56c1 in string::~string() /home/ivan/main.cpp:16
    #2 0x5576100a54bc in main /home/ivan/main.cpp:29
    #3 0x7f3b058fed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7f3b058fee3f in __libc_start_main_impl ../csu/libc-start.c:392
    #5 0x5576100a5224 in _start (/home/ivan/a.out+0x1224)


0x602000000010 is located 0 bytes inside of 6-byte region [0x602000000010,0x602000000016)
freed by thread T0 here:
    #0 0x7f3b05ee2517 in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x5576100a56c1 in string::~string() /home/ivan/main.cpp:16
    #2 0x5576100a54ad in main /home/ivan/main.cpp:29
    #3 0x7f3b058fed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58


previously allocated by thread T0 here:
    #0 0x7f3b05ee2867 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x5576100a5631 in string::string(char const*) /home/ivan/main.cpp:10
    #2 0x5576100a53b5 in main /home/ivan/main.cpp:27
    #3 0x7f3b058fed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58


SUMMARY: AddressSanitizer: double-free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127 in
__interceptor_free
==463==ABORTING
```

Copy constructor (3)

- Copy constructor is a constructor that is called when an object need to copied.
- When it is not defined by the user, it is generated by the compiler doing member-wise copy.

```
void foo(mytype);
mytype a;

mytype b(a);    // copy
mytype b = a;   // copy
foo(a);         // copy
```

Copy constructor (4)

```
string(string const& other)
{
    size = other.size;
    capacity = size;
    data = static_cast<char*>(malloc(capacity + 1));
    memcpy(data, other.data, size + 1);
}
```

Assignment operator (1)

```
int main()
{
    string s("Hello");
    string t;
    t = s;
}
```

Assignment operator (2)

```
$ g++ -g -fsanitize=address main.cpp
$ ./a.out
=================================================================
==494==ERROR: AddressSanitizer: attempting double-free on 0x602000000010 in thread T0:
    #0 0x7f19629f2517 in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x55b1db72979d in string::~string() /home/ivan/main.cpp:31
    #2 0x55b1db7294e8 in main /home/ivan/main.cpp:45
    #3 0x7f196240ed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58
    #4 0x7f196240ee3f in __libc_start_main_impl ../csu/libc-start.c:392
    #5 0x55b1db729244 in _start (/home/ivan/a.out+0x1244)


0x602000000010 is located 0 bytes inside of 6-byte region [0x602000000010,0x602000000016)
freed by thread T0 here:
    #0 0x7f19629f2517 in __interceptor_free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127
    #1 0x55b1db72979d in string::~string() /home/ivan/main.cpp:31
    #2 0x55b1db7294d9 in main /home/ivan/main.cpp:45
    #3 0x7f196240ed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58


previously allocated by thread T0 here:
    #0 0x7f19629f2867 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:145
    #1 0x55b1db72970d in string::string(char const*) /home/ivan/main.cpp:17
    #2 0x55b1db7293d5 in main /home/ivan/main.cpp:42
    #3 0x7f196240ed8f in __libc_start_call_main ../sysdeps/nptl/libc_start_call_main.h:58


SUMMARY: AddressSanitizer: double-free ../../../../src/libsanitizer/asan/asan_malloc_linux.cpp:127 in
__interceptor_free
==494==ABORTING
```

Assignment operator (3)

```
string& operator=(string const& rhs)
{
    size = rhs.size;
    capacity = size;
    data = static_cast<char*>(malloc(capacity + 1));
    memcpy(data, rhs.data, size + 1);
    return *this;
}
```

Assignment operator (4)

```
$ g++ -g -fsanitize=address main.cpp
$ ./a.out

=================================================================
==524==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 1 byte(s) in 1 object(s) allocated from:
    #0 0x7fd1352489a7 in __interceptor_strdup
../../../../src/libsanitizer/asan/asan_interceptors.cpp:454
    #1 0x55b911a44532 in string::string() /home/ivan/main.cpp:10
    #2 0x55b911a443a1 in main /home/ivan/main.cpp:52
    #3 0x7fd134cbdd8f in __libc_start_call_main
../sysdeps/nptl/libc_start_call_main.h:58

SUMMARY: AddressSanitizer: 1 byte(s) leaked in 1 allocation(s).
```

Assignment operator (5)

```
string& operator=(string const& rhs)
{
    free(data);

    size = rhs.size;
    capacity = size;
    data = static_cast<char*>(malloc(capacity + 1));
    memcpy(data, rhs.data, size + 1);
    return *this;
}
```

Assignment operator (6)

```
string& operator=(string const& rhs)
{
    free(data);

    size = rhs.size;
    capacity = size;
    data = static_cast<char*>(malloc(capacity + 1));
    memcpy(data, rhs.data, size + 1);
    return *this;
}



int main()
{
    string s("Hello");
    s = s;
}
```

Assignment operator (7)

```
string& operator=(string const& rhs)
{
    size = rhs.size;
    capacity = size;
    char* new_data = static_cast<char*>(malloc(capacity + 1));
    memcpy(new_data, rhs.data, size + 1);

    free(data);
    data = new_data;

    return *this;
}
```

Assignment operator (8)

```
string& operator=(string rhs)
{
    swap(rhs);
    return *this;
}

void swap(string& other)
{
    using std::swap;
    swap(data, other.data);
    swap(size, other.size);
    swap(capacity, other.capacity);
}
```

Member initializer list (1)

How many memory allocations does this code make?

```
struct person
{
    person()
    {
        name = "Eric";
        surname = "Adams";
    }

private:
    string name;
    string surname;
};

int main()
{
    person p;
}
```

Member initializer list (2)

How many memory allocations does this code make?

```
struct person
{
    person()
    {
        name = "Eric";
        surname = "Adams";
    }

private:
    string name;
    string surname;
};

int main()
{
    person p;
}
```

- 2 default constructors
- 2 constructors (char const*)
- 2 assignment operators

Member initializer list (3)

How many memory allocations does this code make?

```
struct person
{
    person()
        : name("Eric")
        , surname("Adams")
    {}

private:
    string name;
    string surname;
};

int main()
{
    person p;
}
```

- 2 constructors (char const*)

Member initializer list (4)

```
struct range
{
    range(double lo, double hi);
    range(double p)
        : range(p, p)
    {}
};
```

Member initializers

```
#include <cstdint>

struct node
{
    node(int32_t value)
        : value(value)
    {}

private:
    node* parent = nullptr;
    node* left   = nullptr;
    node* right  = nullptr;
    int32_t value;
};
```

Special member functions (1)

```
struct mytype
{
    // generated unless any other constructor is defined by user
    mytype();

    // generated unless defined by user
    mytype(mytype const&);
    mytype& operator=(mytype const&);
    ~mytype();
};
```

Special member functions (2)

- Evaluate if compiler generated special members are appropriate for your class.
- If no consider
  - Either defining your own.
  - Or disabling them with = delete
- If yes consider
  - Marking them with = default

Special member functions (3)

```
struct opened_file
{
    opened_file(opened_file const&) = delete;
    opened_file& operator=(opened_file const&) = delete;
};
```

```
struct range
{
    range() = default;
    range(double lo, double hi);
};
```

Special member functions (4)

The rule of three. If a class defines any of the following then it should probably explicitly define all three:

- destructor
- copy constructor
- copy assignment operator

Special member functions (5)

The rule of zero. If you can avoid defining special member functions, do.

```
struct person
{
    string name;
    string surname;
};
```

User-defined conversion functions (1)

```
struct foobar
{};

struct convertible_from_foobar
{
    convertible_from_foobar(foobar const&);
};

struct convertible_to_foobar
{
    operator foobar() const;
};
```

```
void test_from()
{
    foobar a;

    convertible_from_foobar b1 = a;
    convertible_from_foobar b2(a);
}

void test_to()
{
    convertible_to_foobar a;

    foobar b1 = a;
    foobar b2(a);
}
```

User-defined conversion functions (2)

```
struct foobar
{};

struct convertible_from_foobar
{
    explicit convertible_from_foobar(foobar const&);
};

struct convertible_to_foobar
{
    explicit operator foobar() const;
};
```

```
void test_from()
{
    foobar a;

    convertible_from_foobar b1 = a; // error
    convertible_from_foobar b2(a);  // OK
}

void test_to()
{
    convertible_to_foobar a;

    foobar b1 = a;    // error
    foobar b2(a);     // OK
}
```

User-defined conversion functions (3)

```
struct my_int
{
    my_int(int32_t val);
    operator int32_t() const;
};

bool operator==(my_int, my_int);
bool operator!=(my_int, my_int);

void test(my_int x, int32_t b)
{
    a == b;
}
```

Mutable (1)

```
struct matrix
{
    matrix const& inverted() const
    {
        return inverted_
            ? *inverted_
            : (inverted_ = calc_inv());
    }

    // ...

    mutable matrix* inverted_;
};
```

Mutable (2)

Common applications of mutable include:

- Caches. When the operation is logically const, but needs to update the cache.
- Mutexes.

Custom object lifetime

```
#include <cstdint>

struct node
{
    node(int32_t value)
        : value(value)
    {}

private:
    node* parent = nullptr;
    node* left   = nullptr;
    node* right  = nullptr;
    int32_t value;
};

int main()
{
    node* p = new node(42);
    delete p;
}
```

Custom object lifetime (2)

```
void test()
{
    void* p1 = malloc(sizeof(int32_t) * 10);
    free(p1);

    void* p2 = operator new(sizeof(int32_t) * 10);
    operator delete(p2);

    int32_t* p3 = new int32_t(42);
    delete p3;

    int32_t* p4 = new int32_t[10];
    delete[] p4;
}
```

Custom object lifetime (3)

```
#include <new>

struct mytype
{
    mytype(double x, double y);
    double x, y;
};

void test()
{
    mytype* buffer = static_cast<mytype*>(operator new(sizeof(mytype) * 10));
    for (size_t i = 0; i != 5; ++i)
        new (buffer + i) mytype(1., 2.);

    for (size_t i = 5; i != 0; --i)
        buffer[i - 1].~mytype();
    operator delete(buffer);
}
```