

# Advanced memory handling

- Storage classes in C++
- The new and delete operators
- Overloading new and delete
- New and delete expressions
- Objects with restricted storage classes
- The RAII idiom

# Advanced memory handling

- Storage classes in C++
- The new and delete operators
- Overloading new and delete
- New and delete expressions
- Objects with restricted storage classes
- The RAII idiom

```
namespace std
{
    using dangling_reference = string_view;
}
```

# Storage classes in C++

- String literals are read-only objects
- Automatic (local) variables
- Global (namespace) variables with static lifetime
- Local static variables
- Dynamic memory with new/delete or malloc/free
- Temporaries
- Arrays
- Subobjects (non-static class members)

# Temporaries

- Created when evaluating an expression
- Guaranteed to live until the **full expression** is evaluated

```
void f( string &s1, string &s2, string &s3)
{
    const char *cs = (s1+s2).c_str();
    cout << cs;      // Bad!!

    if ( strlen(cs = (s2+s3).c_str()) < 8 && cs[0] == 'a' ) // Ok
        cout << cs;      // Bad!!
}

void f( string &s1, string &s2, string &s3)
{
    cout << s1 + s2;      // lifetime extension:
    const string &s = s2 + s3; // binding to name keeps temporary
                                // alive until name goes out of scope
    if ( s.length() < 8 && s[0] == 'a' )
        cout << s;      // Ok
}
// s2+s3 destroyed here: when the "s" reference goes out of scope
```

# Lifetime extension

- When a (const) reference is set to a temporary, the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
    mystring(const std::string& s) : std::string(s) {};
    ~mystring() { std::cerr<<"lifetime end: "<< c_str() << '\n'; }
};

mystring operator+( const mystring& a, const mystring& b){
    std::string aa(a), bb(b);
    std::string res(aa+bb);
    return mystring(res);
}

int main() {
    {
        mystring first("first");
        mystring second("second");
        mystring third("third");
        mystring fourth("fourth");
        mystring fifth("fifth");

        const mystring& ref1 = first + second;
        static const mystring& ref2 = first + third;
        static const mystring& ref3 = std::max(fourth, fifth);
        std::cerr << "end of block" << '\n';
    }
    std::cerr << "end of main" << '\n';
}
```

# Lifetime extension

- When a (const) reference is set to a temporary, the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
    mystring(const std::string& s) : std::string(s) {};
    ~mystring() { std::cerr<<"lifetime end: "<< c_str() << '\n'; }
};

mystring operator+( const mystring& a, const mystring& b){
    std::string aa(a), bb(b);
    std::string res(aa+bb);
    return mystring(res);
}

int main() {
    {
        mystring first("first");
        mystring second("second");
        mystring third("third");
        mystring fourth("fourth");
        mystring fifth("fifth");

        const mystring& ref1 = first + second; // ok
        static const mystring& ref2 = first + third; // ok
        static const mystring& ref3 = std::max(fourth, fifth); // wrong!!!
        std::cerr << "end of block" << '\n';
    }
    std::cerr << "end of main" << '\n';
}
```

```
end of block
lifetime end: firstsecond
lifetime end: fifth
lifetime end: fourth
lifetime end: third
lifetime end: second
lifetime end: first
end of main
lifetime end: firstthird
```

# Lifetime extension

- When a (const) reference is set to a temporary, the temporary will live until the reference goes out of scope

```
template <class T>
constexpr const T& max(const T& a, const T& b);

int main() {
    mystring first("first");
    mystring second("second");
    mystring third("third");
    mystring fourth("fourth");
    mystring fifth("fifth");

    const mystring& ref1 = first + second;           // ok
    static const mystring& ref2 = first + third;     // ok
    static const mystring& ref3 = std::max(fourth, fifth); // wrong!!!
    std::cerr << "end of block" << '\n';
}
std::cerr << "end of main" << '\n';
}
```

end of block  
lifetime end: firstsecond  
lifetime end: fifth  
lifetime end: fourth  
lifetime end: third  
lifetime end: second  
lifetime end: first  
end of main  
lifetime end: firstthird

# Lifetime extension

- When a (const) reference is set to a temporary, **or member of a temporary** the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
    mystring(const std::string& s) : std::string(s) {};
    ~mystring() { std::cerr<<"lifetime end: "<< c_str() << '\n'; }
    int ifield;
};

mystring operator+( const mystring& a, const mystring& b){
    std::string aa(a), bb(b);
    std::string res(aa+bb);
    return mystring(res);
}

int main() {
    {
        mystring first("first");
        mystring fourth("fourth");
        mystring fifth("fifth");

        static const int &ref4 = (first + fourth).ifield;
        static const char *const &ref5 = (first + fifth).c_str();
        std::cerr << "end of block" << '\n';
    }
    std::cerr << "end of main" << '\n';
}
```



# Lifetime extension

- When a (const) reference is set to a temporary, **or member of a temporary** the temporary will live until the reference goes out of scope

```
struct mystring : std::string {
    mystring(const std::string& s) : std::string(s) {};
    ~mystring() { std::cerr<<"lifetime end: "<< c_str() << '\n'; }
    int ifield;
};

mystring operator+( const mystring& a, const mystring& b){
    std::string aa(a), bb(b);
    std::string res(aa+bb);
    return mystring(res);
}

int main() {
    {
        mystring first("first");
        mystring fourth("fourth");
        mystring fifth("fifth");

        static const int &ref4 = (first + fourth).ifield;           // ok
        static const char *const &ref5 = (first + fifth).c_str(); // wrong!!!
        std::cerr << "end of block" << '\n';
    }
    std::cerr << "end of main" << '\n';
}
```

end of block  
lifetime end: firstfifth  
lifetime end: fifth  
lifetime end: fourth  
lifetime end: first  
end of main  
lifetime end: firstfourth

# New and delete

- New and delete expressions
- New and delete operators

```
void f()  
{  
    try  
    {  
        int *ip1 = new int;           // new expression (uninitialized)  
        int *ip2 = new int(42);      // new expression (initialized)  
        int *ap1 = new int[10];      // array new expression (uninitialized)  
        int *ap2 = new int[]{1,2,3}; // array new expression  
        auto ip3 = new auto(1);      // creates int *ip3 and one int in heap  
  
        // new operator  
        int *ptr = reinterpret_cast<int *>(::operator new(sizeof(int)));  
  
        ::operator delete(ptr);      // delete operator  
        delete ip;                   // delete expression  
        delete [] ap;                // array delete expression  
    }  
    catch(std::bad_alloc e) { ... }  
}
```

# New expression

New expression do the following 3 steps when called as `new X{`

- Allocate memory for X ( usually calling `operator new(sizeof(X))` )
  - Calls the constructor of X passing parameters if exists
  - Converts pointer to the new object from `void*` to `X*` and returns
- But, what if
    - Operator new throws `bad_alloc`?
    - The constructor throws something?

```
X *ptr;
```

```
try
{
    ptr = new X(par1, par2);
}
catch( ... )
{
    // handle exceptions. Memory leak when constructor throws???
```

# New and delete operators

- New expr. guarantees no leak if the constructor throws exception,

```
X *ptr;  
try  
{  
  
    ptr = new      X(par1, par2); // allocate and run the constructor  
  
}  
catch( ... )  
{  
    // handle exceptions. No memory leak  
}
```

# New and delete operators

- New expr. guarantees no leak if the constructor throws exception

```
X *ptr;
try
{
    void *vp = operator new( sizeof(X) ); // allocate memory
    try {
        ptr = new(vp) X(par1, par2); // run the constructor @vp
    }
    catch( ... ) { // catch all exceptions
        operator delete(vp); // delete memory
        throw; // rethrow exception
    }
    ptr = reinterpret_cast<X*>(vp); // set pointer
}
catch( ... )
{
    // handle exceptions. No memory leak
}
```

# New and delete operators

- May throw `std::bad_alloc` exception
- Returns `void*`

```
namespace std
{
    class bad_alloc : public exception { /* ... */ };
}
```

```
void* operator new(size_t);    // operator new() may throw bad_alloc
void operator delete(void *) // operator delete() never throws
```

# New and delete operators

- May throw `std::bad_alloc` exception
- Returns `void*`

```
namespace std
{
    class bad_alloc : public exception { /* ... */ };
}
```

```
void* operator new(size_t); // operator new() may throw bad_alloc
void operator delete(void *) noexcept; // delete() since C++11
```

# Nothrow version

- In some environments we want to avoid exceptions
- Returns nullptr if allocation is unsuccessful

```
// indicator for allocation that doesn't throw exceptions
struct nothrow_t {};
extern const nothrow_t nothrow;

// what to do, when error occurs on allocation
typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p) throw();

// nothrow version
void* operator new(size_t, const nothrow_t&) noexcept;
void operator delete(void*, const nothrow_t&) noexcept;

void* operator new[](size_t, const nothrow_t&) noexcept;
void operator delete[](void*, const nothrow_t&) noexcept;
```



# Placement new

- Never allocate / deallocate memory

```
// placement new and delete
void* operator new(size_t, void* p) { return p; }
void operator delete(void* p, void*) noexcept { }

void* operator new[](size_t, void* p) { return p; }
void operator delete[](void* p, void*) noexcept { }

#include <new>
void f()
{
    char *cp = new char[sizeof(C)];
    for( long long i = 0; i < 100000000; ++i)
    {
        C *dp = new(cp) C(i);
        // ...
        dp->~C(); // explicit call of destructor
    }
    delete [] cp;
    return 0;
}
```

# Overloading new and delete

- New and delete operators can be overloaded at two level
  - Class level (automatically static member functions)
  - Namespace level

```
struct node
{
    node( int v) { val = v; left = righth = 0; }
    void print() const { cout << val << " "; }

    /* static */ void *operator new( size_t sz) throw (bad_alloc);
    /* static */ void operator delete(void *p) noexcept;

    int val;
    node *left;
    node *right;
};
```

# Overloading new and delete

```
// member new and delete as static member
void *node::operator new( size_t sz) throw (bad_alloc)
{
    return ::operator new(sz);
}
void node::operator delete(void *p) noexcept
{
    ::operator delete(p);
}
// global new and delete
void *operator new( size_t sz) throw (bad_alloc)
{
    return std::malloc(sz);
}
void operator delete( void *p) noexcept
{
    std::free(p);
}
```

# Extra parameters for new

- One can define new and delete with extra parameters
  - Only new expression can pass extra parameters

```
struct node
{
    node( int v);
    void print() const;

    static void *operator new( size_t sz, int i);
    static void operator delete(void *p, int i) noexcept;

    int val;
    node *left;
    node *right;
};
void f()
{
    int arena = 3;
    node *r = new(arena) node(i);
}
```

# Alignment

```
#include <cassert> // example from https://www.bfilipek.com/2019/08/newnew-align.html
#include <cstdint>
#include <iostream>
#include <malloc.h>
#include <new>

class alignas(32) Vec3d {
    double x, y, z;
};

int main() {
    std::cout << "sizeof(Vec3d) is " << sizeof(Vec3d) << '\n';
    std::cout << "alignof(Vec3d) is " << alignof(Vec3d) << '\n';

    auto Vec = Vec3d{};
    auto pVec = new Vec3d[10];

    if(reinterpret_cast<uintptr_t>(&Vec) % alignof(Vec3d) == 0)
        std::cout << "Vec is aligned to alignof(Vec3d)!\n";
    else
        std::cout << "Vec is not aligned to alignof(Vec3d)!\n";

    if(reinterpret_cast<uintptr_t>(pVec) % alignof(Vec3d) == 0)
        std::cout << "pVec is aligned to alignof(Vec3d)!\n";
    else
        std::cout << "pVec is not aligned to alignof(Vec3d)!\n";

    delete[] pVec;
}
```

# Alignment

```
// Before C++17
```

```
sizeof(Vec3d) is 32
```

```
alignof(Vec3d) is 32
```

```
Vec is aligned to alignof(Vec3d)!
```

```
pVec is not aligned to alignof(Vec3d)!
```

# Alignment

```
// Before C++17
```

```
sizeof(Vec3d) is 32  
alignof(Vec3d) is 32  
Vec is aligned to alignof(Vec3d)!  
pVec is not aligned to alignof(Vec3d)!
```

```
// Since C++17
```

```
sizeof(Vec3d) is 32  
alignof(Vec3d) is 32  
Vec is aligned to alignof(Vec3d)!  
pVec is aligned to alignof(Vec3d)!
```

# Alignment

```
// Before C++17
```

```
sizeof(Vec3d) is 32  
alignof(Vec3d) is 32  
Vec is aligned to alignof(Vec3d)!  
pVec is not aligned to alignof(Vec3d)!
```

```
// Since C++17
```

```
sizeof(Vec3d) is 32  
alignof(Vec3d) is 32  
Vec is aligned to alignof(Vec3d)!  
pVec is aligned to alignof(Vec3d)!
```

```
#include <cstdlib>
```

```
void *malloc( std::size_t size); // align to std::max_align_t  
// Since C++17 over-aligning:  
void *aligned_alloc( std::size_t alignment, std::size_t size);
```



# Operator new in C++17

- 14 global new() operator functions
- 8 class specific new() operators
- Corresponding delete() operators
- Overloading on std::align\_val\_t

```
enum class align_val_t : std::size_t {};
```

```
void *operator new( std::size_t sz, std::align_val_t al);  
void operator delete( void *ptr, std::align_val_t al) noexcept;
```

```
void f();  
{  
    auto ip    = new int{};    // uses usual default alignment == 16  
    auto pVec  = new Vec3d{}; // uses class specified alignment == 32  
    auto p64i = new(std::align_val_t{64}) int{}; // alignment == 64  
  
    delete ip;  
    delete pVec;  
    delete p64i;  
}
```

# Objects only in heap

- Sometimes restriction for storage location is useful

```
class X // a class should be allocated only on heap
{
public:
    X() {}
    void destroy() const { delete this; }
protected:
    ~X() {}
};
class Y : public X { };
// class Z { X xx; }; // use pointer instead!
void f()
{
    X* xp = new X;
    Y* yp = new Y;

    delete xp; // compile error
    xp->destroy(); // ok
}
```

# Objects never in heap

- Sometimes restriction for storage location is useful

```
class X // Class should be allocated only outside of heap
{
private:
    static void *operator new( size_t);
    static void operator delete(void *) noexcept;
    // static void operator delete(void *) noexcept = delete; in C++11
};
```

```
class Y : public X { };
class Z { X xx; }; // ok!
void f()
{
    X* xp = new X; // compile error
    X x; // ok
}
```

# RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this correct?  
void f()  
{  
    char *cp = new char[1024];  
  
    g(cp);  
    h(cp);  
  
    delete [] cp;  
}
```

# RAII

- Resource Acquisition Is Initialization
- The idea: keep a resource is expressed by object lifetime

```
// is this maintainable?  
void f()  
{  
    char *cp = new char[1024];  
  
    try  
    {  
        g(cp);  
        h(cp);  
        delete [] cp;  
    }  
    catch (...)  
    {  
        delete [] cp;  
        throw;  
    }  
}
```

# RAII

- Constructor allocates resource
- Destructor deallocates

```
// RAII
struct Res
{
    Res(int n) { cp = new char[n]; }
    ~Res() { delete [] cp; }
    char *getcp() const { return cp; }
};

void f()
{
    Res res(1024);

    g(res.getcp());
    h(res.getcp());
}
// resources will be free here
```

# RAII

- Constructor allocates resource
- Destructor deallocates

```
// RAII
struct Res
{
    Res(int n) { cp = new char[n]; }
    ~Res() { delete [] cp; }
    char *getcp() const { return cp; }
};
```

```
void f()
{
    Res res(1024);

    g(res.getcp());
    h(res.getcp());
}
// resources will be free here
```

What about copying Res?

# RAII

- Should be careful when implementing RAII
- Destructor calls only when **living object** goes out of scope
- Object lives only when constructor has successfully finished

// But be careful:

```
struct BadRes
{
    Res(int n) { cp = new char[n]; ... init(); ... }
    ~Res()     { delete [] cp; }
    char *cp;
    void init()
    {
        ... if (error) throw XXX;
    }
};
```



# RAII

- Should be careful when implementing RAII
- Destructor calls only when **living object** goes out of scope
- Object lives only when constructor has successfully finished

// But be careful:

```
struct BadRes
```

```
{
```

```
  Res(int n) { cp = new char[n]; ... init(); ... }
```

```
  ~Res()     { delete [] cp; }
```

```
  char *cp;
```

```
  void init()
```

```
  {
```

```
    ... if (error) throw XXX;
```

```
  }
```

```
};
```

**Possible memory leak if throws!**

# Typical RAII solutions

- Smart pointers for memory handling
- Guards for locking
- ifstream, ofstream objects for file-i/o
- std::containers

```
class X
{
public:
    void *non_thread_safe();
private:
    Mutex lock_;
};

void *X::non_thread_safe();
{
    Guard<Mutex> guard(lock_);
    /* critical section is here*/
}
```