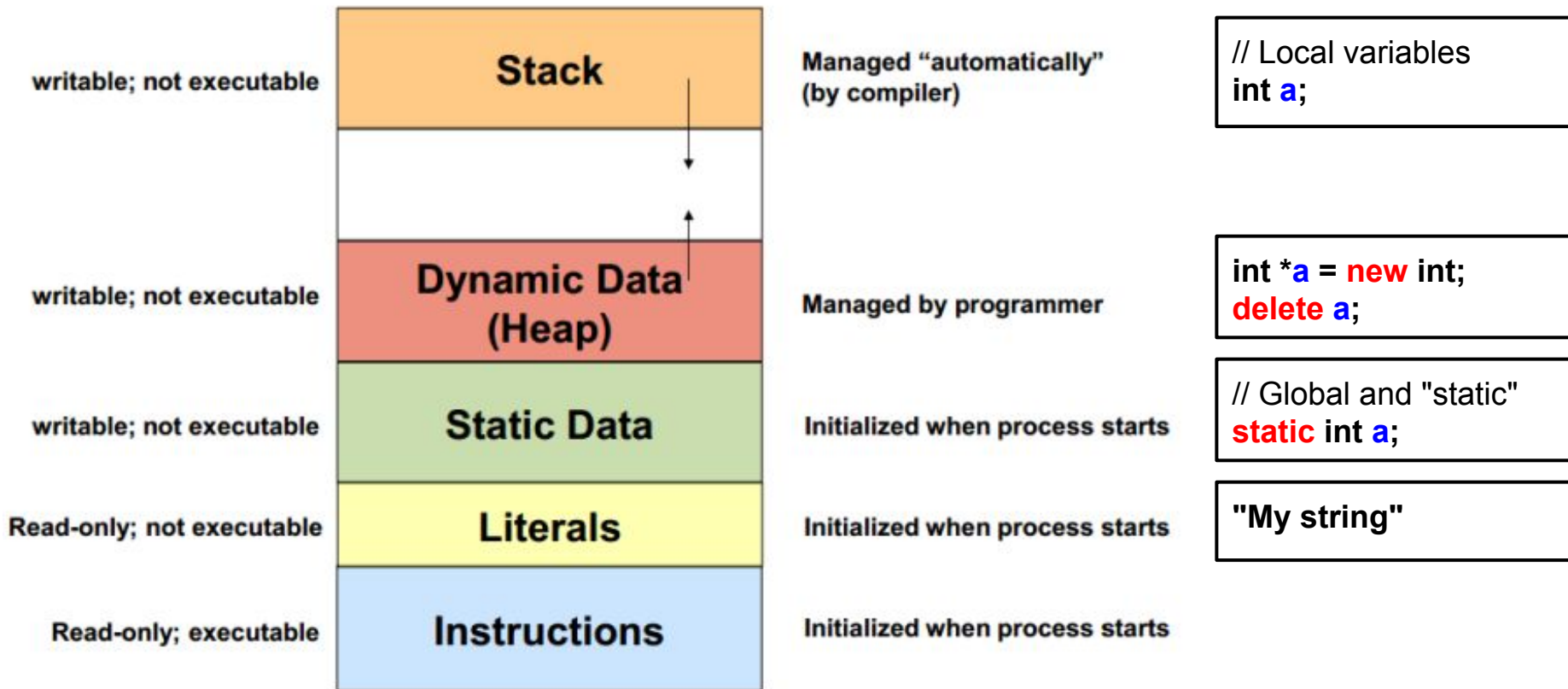


C++ Course 5: Smart Pointers. Miscellanea 1.

2020 by Oleksiy Grechnyev

C++ memory model



Object Life Cycle

Every C++ object is born and eventually dies

| Scope | Example | Birth | Death |
|------------------------|--------------------------------|-----------------|-------------------|
| Local variable | string s = "Jezebel"; | Definition | The closing } |
| Temporary object | string ("Jack"); | Code line start | Code line end |
| Global/static variable | static string ("Jill"); | Program start | Program end |
| Heap object | new string ("Maria"); | new | delete !!! |
| Class field | string s = "Lilith"; | Class Ctor | Class Dtor |
| Smart pointers | ? | ? | ? |

Working with heap memory: new and delete

Old-style C++ (before C++ 11):

```
void fun(){  
    string * pS = new string("Some text");  
    ..  
    delete pS; // Don't forget delete !!! Otherwise a memory leak !  
}
```

```
class A{  
public:  
    A(const char * s) : pS(new std::string(s)) {} // Ctor  
    ...  
    ~A(){ delete pS;} // Dtor  
private:  
    string * pS;  
};
```

Trouble with heap objects 1

```
string * pS = new string("Some text");
```

If we forget **delete**: Memory Leak!

If we put delete twice : double **delete**. Program crashes!

```
delete pS; // No way to check if pS is deleted already
```

```
delete pS; // Double delete. CRASH !
```

Things that make it **worse** : multiple returns, exceptions.

```
void fun(){
```

```
    string * pS = new string("Some text");
```

```
    ..
```

```
    if (...) return; // Forgot delete here ! Memory leak !!!
```

```
    ...
```

```
    if (..) throw runtime_error("HAHA !"); // Forgot delete here ! Memory leak !!!
```

```
    delete pS;
```

```
}
```

Trouble with heap objects 2

There is no way to tell if a pointer points to a valid heap object:

```
string s1("Nel Zelphe");  
string *pS1 = &s1;  
delete s1; // Wrong ! pS1 points to a local, and not a heap object!  
string *pS2 = new string("Sophia Esteed"); // Created a heap object  
delete pS2; // Deleted it. OK !  
delete pS2; // Error ! Double delete !  
int *pI = new int; // int heap object  
string *pS3 = (string *) pI; // pS3 points to an int heap object (Wrong type !)  
delete pS3; // Error ! Wrong type !
```

My program crashes, why ??????

No way to check for these situations !!!

Solution: unique_ptr

`unique_ptr` takes a heap object under *exclusive ownership*

```
unique_ptr<string> u = make_unique<string>("Abracadabra");    // C++ 14
```

or

```
unique_ptr<string> u(new string("Abracadabra"));    // C++ 11
```

It is a thin and efficient wrapper around a raw pointer (No overheads !). Roughly speaking:

```
template <typename T>
```

```
class unique_ptr{
```

```
public:
```

```
    unique_ptr(const T & data): data(data) {} // Ctor
```

```
    ~unique_ptr() {delete data;} // Managed object is deleted when unique_ptr dies !
```

```
    ... // Also move Ctor and move assignment, but not copy !
```

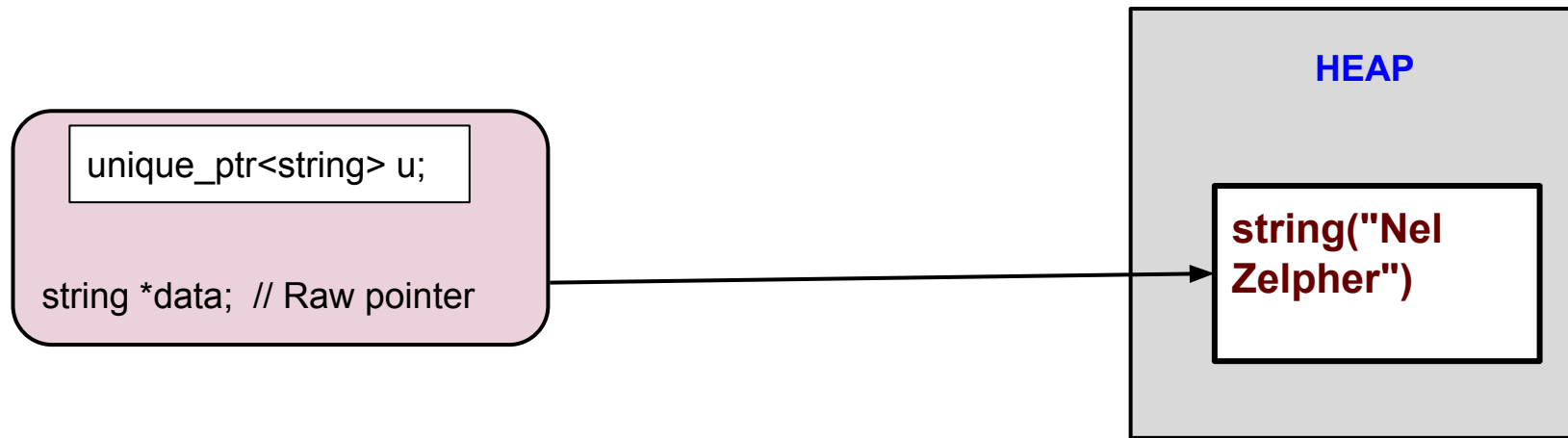
```
    // Also operator->, operator*, get(), release() ...
```

```
private:
```

```
    T * data;    // Pointer to the managed object
```

```
};
```

unique_ptr manages a heap object



unique_ptr manages a heap object (can also hold **nullptr** == empty **unique_ptr**)

While **unique_ptr** lives, the managed object lives

When **unique_ptr** dies, its destructor deletes the managed object by **delete**

unique_ptr has the size of a *raw pointer* (typically 8 bytes)

unique_ptr is as efficient as raw pointer, no size or performance overheads !

unique_ptr can be *moved* but not *copied*

Using heap with unique_ptr

```
void fun(){
    unique_ptr<string> u = make_unique<string>("Some text");
    ..
    // The string object is deleted here by the unique_ptr destructor!
} // The closing brace: u runs out of scope here

class A{
public:
    A(const char * s) : uS(new std::string(s)) {} // Ctor
    ...
    // No Destructor. The managed object is deleted automatically !
private:
    unique_ptr<string> uS; // When object of class A dies, Dtor of uS is called
};
// No more delete operators anywhere ! Managed objects die together with unique_ptr !
```

What not to do !

```
string *pS = new string("Maria Traydor"); // Created a heap object with new !  
unique_ptr<string> uS1(pS); // Created a unique_ptr out of it
```

Now **uS** has exclusive ownership of the heap object !

DON'T DO THIS:

```
delete pS; // Wrong ! unique_ptr takes care of it ! Double delete !  
unique_ptr<string> uS2(pS); // Wrong ! Creating two unique_ptr for the same object !!!
```

Cannot create 2 **unique_ptr** objects from a single heap object !!! Double delete !

make_unique makes it safer, no explicit **new**, no raw pointers:

```
auto uS1 = make_unique<string>("Maria Traydor");
```

Using unique_ptr 1

```
auto u = make_unique<string>("Maria Traydor");
```

What can we do with it ? Use it as a normal pointer!

Operators `*u`, `u->` are defined. `*u` is the underlying **string** object.

```
cout << "*u = " << *u << endl; // Dereferencing
```

```
*u = "Nel Zelpher"; // Change the string (NOT pointer !)
```

```
cout << "*u = " << *u << endl; // Print the string again !
```

```
cout << " u->size() = " << u->size() << endl; // Call method, operator-> is overloaded
```

```
cout << " (*u).size() = " << (*u).size() << endl; // The same !, operator* is overloaded
```

Check that **unique_ptr** object has a managed object (i.e. is not **nullptr**):

```
if (u)
```

```
...
```

Using unique_ptr 2

```
auto u = make_unique<string>("Maria Traydor");
```

```
u.reset();           // Force delete the managed object (if any), u becomes nullptr
u = nullptr;         // The same
string *p1 = u.get(); // Get the raw pointer (Don't delete it !)
delete p1; // Error !!!
string *p2 = u.release(); // Release ownership of the managed object, u becomes nullptr
delete p2;           // Now you must delete it !
```

`unique_ptr` cannot be copied but can be moved !

```
auto u2 = u;          // Error !!!
auto u2 = move(u);     // OK ! Ownership transferred to u2, u becomes nullptr !
```

`move` transfers ownership of the managed object to another `unique_ptr` object.

The old object `u` is reset (set to `nullptr`).

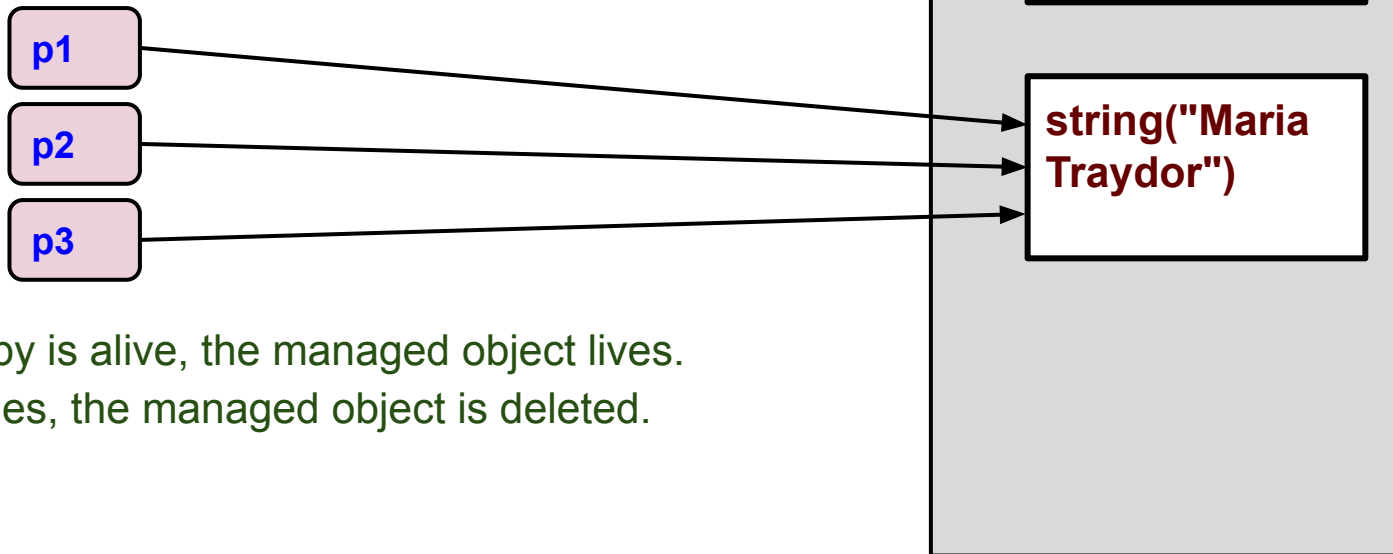
Extra slides: creating sources (factories) and sinks with `unique_ptr`

Smart pointer which can be copied ?

ONE **unique_ptr** manages *ONE* heap object :

```
unique_ptr<string> u;
```

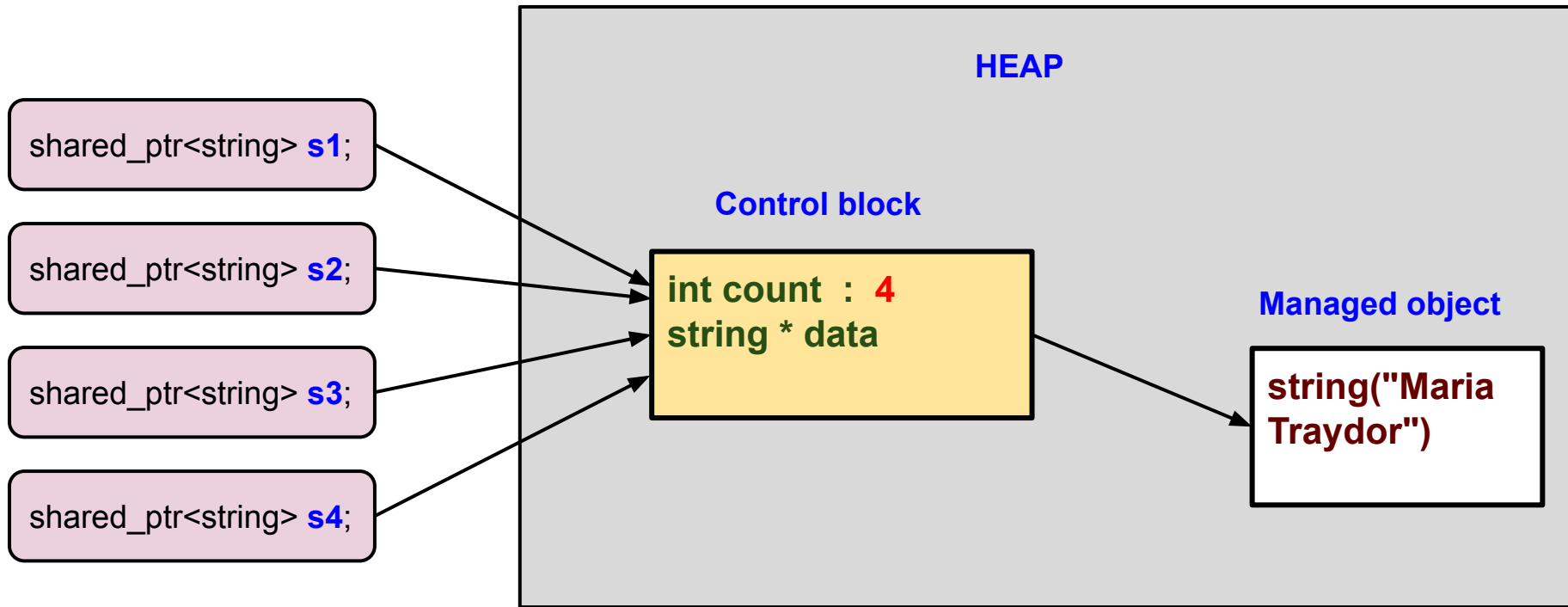
We want: *MANY* copies together manage *ONE* heap object :



While *at least one* copy is alive, the managed object lives.

When *the last* copy dies, the managed object is deleted.

shared_ptr concept : Reference Counting in C++



While *at least one* copy is alive, the **managed object** AND the **control block** live.
When *the last* copy dies, they are BOTH deleted (*reference counting pattern*).

Creating shared_ptr

Creating shared_ptr :

```
shared_ptr<Tjej> s1 = make_shared<Tjej>("Maria Traydor");  
auto s2 = make_shared<Tjej>("Nel Zelphe"); // Preferred, beautiful !  
shared_ptr<Tjej> s3(new Tjej("Sophia Esteed")); // Ugly, new without delete !
```

You can also correctly convert unique_ptr to shared_ptr (don't forget move !)

```
auto u = make_unique<Tjej>("Mirage Koas");  
shared_ptr<Tjej> s4(move(u)); // u is nullified, s4 takes over the object
```

shared_ptr can be both copied and moved:

```
auto s5 = s1;  
auto s7 = move(s3); // Now s3 becomes nullptr  
s3 = s2; // Copy pointer, not value  
  
*s1 = *s3; // Copy value, not pointer!
```

Using shared_ptr

Use it as a normal pointer!

```
auto s = make_shared<string>("Kajsa");  
cout << "s = " << *s << endl;  
*s = "Eva"; // Change the value, not ptr !  
cout << "s = " << *s << endl;  
cout << "s->size() = " << s->size() << endl;
```

Other things you can do:

```
s.reset(); // Resets s, does not delete the managed object unless it's the last copy  
s = nullptr; // The same  
cout << s.use_count() << endl; // Number of copies in existence  
cout << s.unique() << endl; // Is this the only copy ?  
string *str = s.get(); // Get a raw pointer  
if (s) // Check that s is valid (not nullptr)
```

...

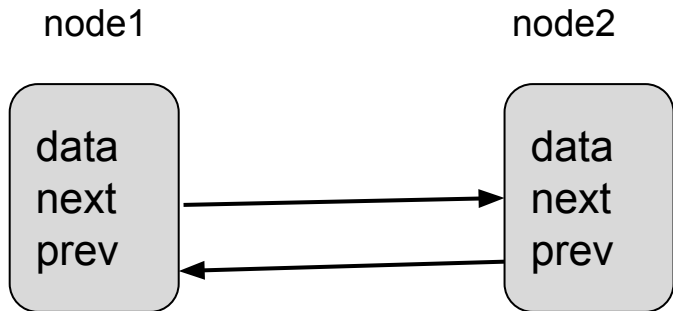
shared_ptr philosophy

- **shared_ptr** Introduces some overheads (control block), don't create 10^9 of them!
- **shared_ptr** feels very similar to Java/Python/.... objects
- Copying **shared_ptr** copies the reference, not the data
- However it uses *reference count*, not full Java-style garbage collection!
- Memory leak is still possible from *cyclic reference*
- The same principle (*reference count*) is used by **cv::Mat**
- *Objects in heap*: no matter where **shared_ptr** or **unique_ptr** is (stack, class fields), the *managed object* is always in the heap! This is good for large managed objects!
- BTW: Most C++ containers (e.g. **vector**) manage their own memory and keep their data in the heap. Exception : **array**.
- **shared_ptr<vector>** is a good combination: container with shared access and no copying.

Can shared_ptr cause a memory leak ?

Suppose we want a double-linked list of nodes:

```
struct Node{  
    Node(const string & data) : data(data) {}  
    string data;  
    shared_ptr<Node> next;    // Next node  
    shared_ptr<Node> prev;    // Previous node  
};
```



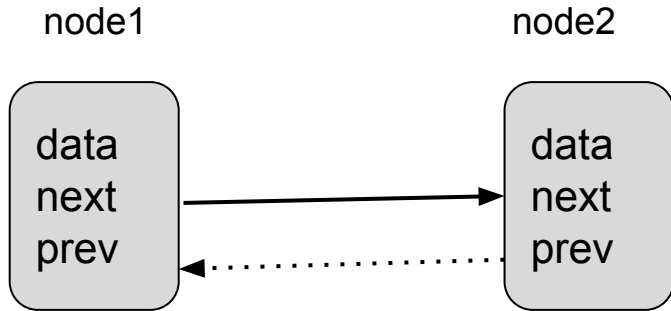
shared_ptr cyclic reference ! **Memory leak !!!**

Nodes are never deleted as they keep each other alive!

Solution: weak_ptr

Let us rewrite our node like this:

```
struct Node{  
    Node(const string & data) : data(data) {}  
    string data;  
    shared_ptr<Node> next;    // Next node  
    weak_ptr<Node> prev;     // Previous node : weak_ptr !!!!  
};
```



No more **shared_ptr** cyclic reference ! No more memory leak !!!

Extra slides : Using weak_ptr

Exceptions

Exception interrupts the flow of the program

```
if (a >= 0)
    throw std::runtime_error("The user is an idiot !"); // No "new" here !!! Important !
```

Exception can be caught (`std::runtime_error` is a subclass of `std::exception`)

```
try {
    ...
} catch (const std::exception & e) {
    cerr << e.what() << endl;
}
```

In the exception is NOT caught, the program terminates

Re-throw and noexcept

Re-throw the exception just caught:

```
catch (...) { // "..." means "Catch everything"
    cout << "Caught something !!! " << endl;
    throw ; // Re-throw the same exception again
}
```

noexcept : Specifies that a function/method does not throw exceptions.

It includes the exceptions *passing through* (f1() -> f2() -> f(3)).

```
int add(int a, int b) noexcept {
    return a+b;
}
```

noexcept allows for better optimization. If the function throws anyway, it cannot be caught and the program terminates. Always used for move Ctors/assignment.

Extra Slides on exceptions: Your own exception classes etc.

std::pair : container with 2 elements of different types

pair<A, B> is a simple container with two fields of different types T, U:

```
template <typename T, typename U>
```

```
struct pair{
```

```
    ...
```

```
    T first;
```

```
    U second;
```

```
};
```

```
pair<string, int> p1("Maria Traydor", 19);
```

```
auto p2 = make_pair(string("Nel Zelpher"), 23); // Easy to get types wrong !
```

```
auto p3 = make_pair<string, int>("Nel Zelpher", 23); // Better !
```

```
auto p4 = pair<string, int>("Nel Zelpher", 23); // Basically same result
```

```
pair<string, int> p5;
```

```
p5 = {"Sophia Esteed", 19}; // Assign with a list
```

```
cout << p1.first << " : " << p1.second << endl;
```

Read it yourself : std::tuple

std::optional (C++ 17) : either an object or an empty container

Creating **optional**:

```
optional<string> o1;           // Empty optional  
optional<string> o2{"Arboga 10.2%"}; // Optional with a string
```

Using **optional**:

| | |
|--------------------------------------|---|
| o2.has_value() | Does o2 have value? false if o2 is empty |
| o2.value() | Value of o2 , throws bad_optional_access if empty |
| if (o2) ... | Check that o2 is not empty |
| o2.value_or("Default string") | Value, or default value if empty |

More options:

```
o2.reset();           // Clear o2 (make it empty)  
o2 = nullopt;         // The same  
o2 = make_optional<string>("Spendrups"); // Create optional, don't copy/move string
```

Note : **unique_ptr** can be used instead of **optional** (**nullptr** for empty value)

std::any (C++ 17) : object of any type (or empty)

Container of dynamic type in C++ ! Can be assigned object of (almost) any type !

```
any a;           // Create an empty any object
a = 17;          // a becomes int
a = string("Reimi Saionji"); // a becomes string
a.reset();       // Make a empty
```

Using any:

| | |
|-------------------------|--|
| a.type() | Type of a (std::type_info object), or void for empty |
| a.type().name() | Name of the type (not really verbose !) |
| a.type() == typeid(int) | Compare (at runtime) to a known type |
| a.has_value() | Does a contain anything? false for empty |
| any_cast<int>(a) | Retrieve the int value, throws bad_any_cast if wrong type |

Note: The types must be *strictly* identical ! No **int/double** or **int/long** conversion !

enum class and enum

enum class : C++ enumerated type:

```
enum class Color {red, green, blue, cyan, magenta, yellow, orange};  
Color a = Color::magenta; // Create a variable  
int b = static_cast<int>(a); // Need explicit cast to convert to int  
switch (a) { // Use in switch  
    case Color::green:  
        ...
```

Note: constants are scoped by **Color::**, no name conflicts !

enum : C enum type

```
enum ColorE {red, green, blue, cyan, magenta, yellow, orange};  
ColorE a = magenta; // Create a variable, names leak, bad !  
int b = a; // Implicit conversion to int, dangerous !
```

Do not use this in C++, except maybe *inside* class definition

Extra slides : enum

Library of the day : Eigen (Example 5_6)

Eigen is a header-only C++ linear algebra library (can optionally use lapack)

Plays a role similar to *numpy*

```
Eigen::MatrixXd m(2, 3); // Double matrix : 2 rows, 3 cols
m << 1, 2, 3, 4, 5, 6; // Provide values (overloaded operators << and ,)

Eigen::VectorXd v(3); // Column double 3-vector
v << 1, 0, -1;

cout << "m = " << m << endl;
cout << "v = " << v << endl;
cout << "m*v = " << m*v << endl; // Multiply matrix by vector
```

- Matrices are *copied* (unlike in OpenCV)
- Don't use fixed-size matrices like **Matrix4d**, alignment issues!
- **ArrayXXd** is similar to **MatrixXd**, but with element-wise operator *****
- Contains diagonalization, SVD, ...

Type conversions (type casts)

Implicit conversions:

Primitive types, pointers to **void ***, pointer upcast, constructors, cast operators

C++-style casts. Conversion from type B to type A:

B **b**;

const_cast<A>(**b**) // Remove const from a pointer or reference

static_cast<A>(**b**) // Various type conversions

dynamic_cast<A>(**b**) // Safe polymorphic cast (pointer or reference)

reinterpret_cast<A>(**b**) // Reinterpret memory bytes as different type

C-style casts:

(A)**b** // Roughly speaking **const_cast**, **static_cast**, **reinterpret_cast**

A(**b**) // In that order

Implicit conversions : Dangerous !

Implicit conversions:

```
A a;  
B b;  
b = a;           // A is converted to B on assignment  
myfunc(a);       // Expects argument of type B, type A provided instead
```

Primitive types:

```
float a = 777;    // Possible loss of accuracy  
int b = 3.5;      // Loss of accuracy  
char c = 1987;    // Loss of higher bytes (Warning !)
```

Other types:

```
void * pV = &a;    // Cast any pointer to void *  
string s = "Phoenix"; // Constructor (NOT explicit !) : cast const char[8] to string  
bool b(cout);      // Cast operator: cast ostream to bool
```

const_cast: Remove (cast away) const from ptrs, refs

Converts **const type *** to **type *** or **const type &** to **type &** .

Bad style, don't do this without a very good reason!

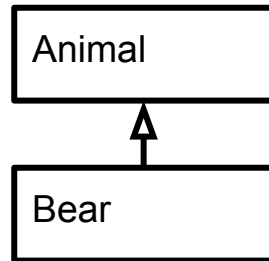
```
int a = 17;
const int & crA = a;
int & rA = const_cast<int &>(crA);    // Remove const
rA = 20;
cout << "a = " << a << endl;      // Prints 20

double b = 1.1;
const double * cpB = &b;
double * pB = const_cast<double *>(cpB);    // Remove const
*pB = 2.2;
cout << "b = " << b << endl;      // Prints 2.2
```

C++ Polymorphism: Upcasts and Downcasts

```
class Animal {...};  
class Bear: public Animal {...};
```

Bear inherits **Animal**



Upcasts: Every **Bear** is an **Animal** ! Safe and implicit.

Bear & to **Animal &**, **Bear *** to **Animal ***

Downcasts: Not every **Animal** is a **Bear** !

Animal & to **Bear &**, **Animal *** to **Bear ***

Requires **static_cast** or **dynamic_cast** !

Only works if our **Animal &** or **Animal *** points to a **Bear** object !

static_cast: All "normal" casts

1. Implicit conversion made explicit (specify precise type and remove warnings)

```
string s = static_cast<string>("Idiot !");
```

2. enum class <-> int

```
int i = static_cast<int>(Num::Four);
```

3. void * to any pointer (No checks !)

4. Reference/pointer downcast: **Base *** to **Derived ***, **Base &** to **Derived &**. No checks!

Does not check that the object is of **Derived** class, unsafe !

```
Derived d;  
Base & rB = d;  
Base * pB = &d;  
Derived & rD = static_cast<Derived &>(rB);    // Downcast  
Derived * pD = static_cast<Derived *>(pB);    // No checks !  
rD.print();                                  // Prints "Derived" twice  
pD->print();
```

dynamic_cast: Pointer/reference downcast with checks

Like previous example, but with checks at runtime (slower):

```
Derived d;  
Base & rB = d;  
Base * pB = &d;  
Derived & rD = dynamic_cast<Derived &> (rB);    // Downcast  
Derived * pD = dynamic_cast<Derived *> (pB);    // With checks !  
rD.print();                                     // Prints "Derived" twice  
pD->print();
```

throws **bad_cast** (for references) or returns **nullptr** (for pointers) if the type is wrong:

```
Base b2;  
Base & rB2 = b2;  
Base * pB2 = &b2;  
Derived & rD2 = dynamic_cast<Derived &> (rB2);    // throws bad_cast  
Derived * pD2 = dynamic_cast<Derived *> (pB2);    // returns nullptr
```


reinterpret_cast, C-style casts

reinterpret_cast interprets memory bytes as a different type

Converts pointers of different types (e.g. **int *** to **double ***) with no checks.

```
int i = 17;  
int *pi = &i;    // Pointer  
long long j = reinterpret_cast<long long>(pi);
```

C-style casts (Bad style, IMHO OK for primitive types):

Roughly speaking: **const_cast**, **static_cast** or **reinterpret_cast** in this order.

```
char c = (char) 2017;  
int i = (int) 13.456789;  
Derived & rD = (Derived &) rB;  
Derived * pD = (Derived *) pB;
```

Extra slides: **unique_ptr**, **shared_ptr** and **polymorphism**

Thank you for your attention !

unique_ptr and source (factory) functions

Source (factory) creates a heap object and returns **unique_ptr**

```
unique_ptr<Tjej> factory1(const string & name){  
    return make_unique<Tjej>(name); // No need for explicit move here  
}
```

```
unique_ptr<Tjej> factory2(const string & name){  
    unique_ptr<Tjej> upT = make_unique<Tjej>(name);  
    /// Do something with upT  
    return move(upT); // Will work without move also  
}
```

Usage:

```
auto upT1 = factory1("Nel Zelpher");  
auto upT2 = factory2("Claire Lasbard");  
...  
} // upT1, upT2 and managed objects are destroyed HERE !
```

unique_ptr and sinks (consumers)

Sink (consumer) eats **unique_ptr** and destroys it (and the managed object)

Argument is passed *by value* (NOT reference !) using **move** :

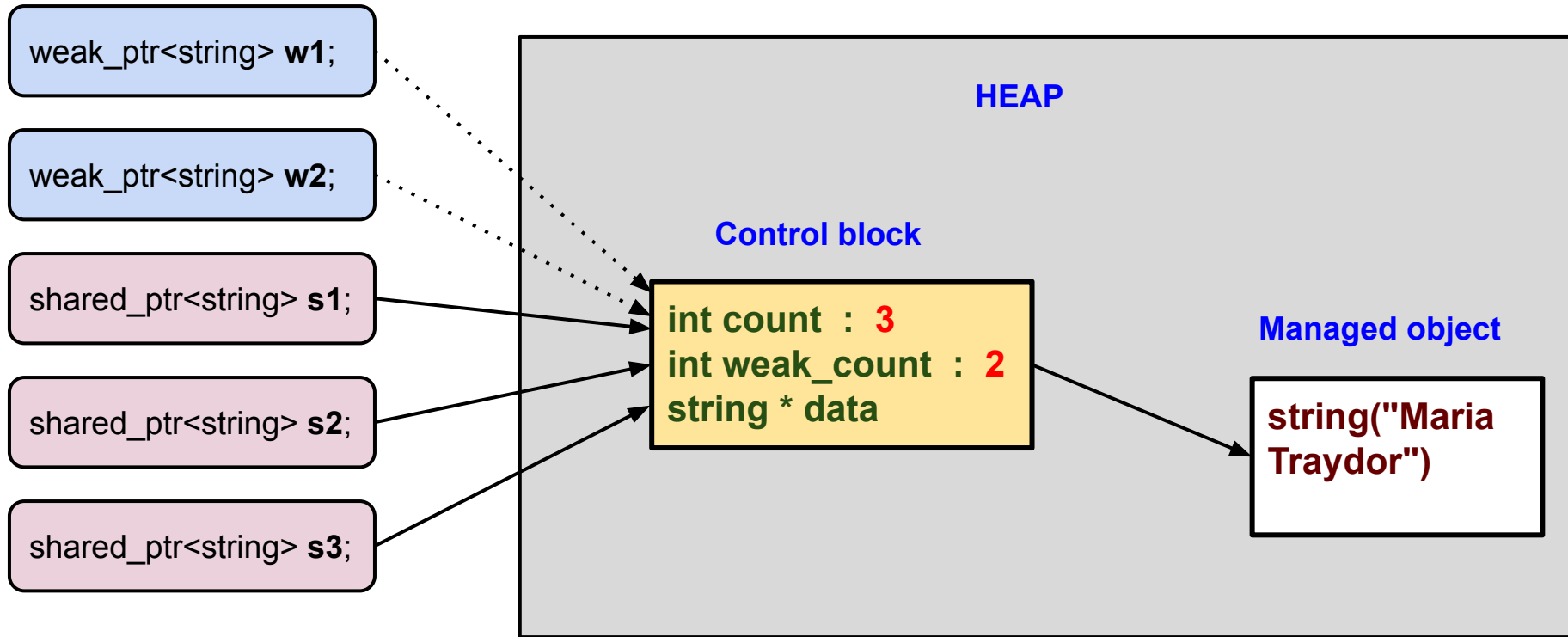
```
void sink(unique_ptr<Tjej> upT){    // By value !!! NO ref ! MOVED here !
    // upT is local in this function
    cout << "Sink: name = " << upT->getName() << endl;
    // Now upT and the managed object are destroyed as they go out of scope !
}
```

Usage:

```
auto upT = make_unique<Tjej>("Sophia Esteed");
sink(move(upT));                // Move to sink

// The managed object is destroyed by now and upT == nullptr
...
} // Nothing happens here !
```

weak_ptr concept: a weak reference to a shared_ptr



When the last **shared_ptr** dies, the **managed object** is deleted, the **control block** stays !

When the last **weak_ptr** dies, the **control block** is deleted !

Using weak_ptr

Create a `weak_ptr` out of a `shared_ptr` :

```
shared_ptr<string> s = make_shared<string>("Maria Traydor");  
weak_ptr<string> w = s;
```

Restore `shared_ptr` from a `weak_ptr` (creates one more `shared_ptr` copy):

```
if (! w.expired() ) {           // Check if w is expired  
    shared_ptr<string> s = w.lock();    // Returns nullptr if expired  
    ...  
}
```

`weak_ptr` is *expired* when the last `shared_ptr` dies and the managed object is deleted

`weak_ptr` can be reset, copied and moved, but cannot be dereferenced:

```
*w           // Error !!!  
w->size()     // Error !!!
```

`weak_ptr` can be also used for the *weak reference* pattern:

 caches, observers (like **WeakReference** in Java)

Objects of any type can be thrown and caught

```
try {  
    throw 17;                // int  
    throw (unsigned)17;      // unsigned, not int !  
} catch (int i) {  
    cout << "int exception " << i << endl;  
} catch (double d) {  
    cout << "double exception " << d << endl;  
} catch (const string & s) {  
    cout << "string exception " << s << endl;  
} catch (const char * cS) {    // Will catch string literals  
    cout << "char * exception " << cS << endl;  
} catch (...) {               // Default  
    cout << "Unknown exception" << endl;  
}
```

NEVER EVER use **new** in **throw** !!!! Memory leak !!!

But the standard class is `std::exception` and subclasses

```
throw runtime_error("Earthquake"); // std::string parameter !  
throw logic_error("Earthquake");  
// invalid_argument, domain_error, length_error, out_of_range  
// range_error, overflow_error, underflow_error
```

Define your own exception: Inherit **runtime_error** (simplest), or implement **std::exception**
what() message is printed if not caught (**std::exception** subclasses only !)

```
struct DiamondException : public std::runtime_error{  
    explicit DiamondException(const std::string & s) :  
        runtime_error("Diamond: " + s) {}  
};  
  
struct SapphireException : public std::exception{  
    const char * what() const noexcept override {  
        return "Al2O3";  
    }  
};
```


Exceptions and function calls

```
void f3() {  
    int a;  
    throw runtime_error("HAHA");  
}  
void f2() {  
    string s("Local String"); f3();  
}  
void f1(){ f2(); }  
void main() {  
    try {  
        f1(); // Calls f1() -> f2() -> f(3)  
    } catch (...) { /* Some code*/ }  
}
```

Exception thrown in **f3** goes right through **f2** and **f1** and is caught in **main**.
Any local variables of **f3**, **f2**, **f1** are properly deleted (*stack unwinding*)

enum (class) : specify numerical values

```
enum Color {red = 17, green, blue, cyan, magenta, yellow, orange};  
cout << red << " " << green << " " << blue << " " << cyan << " " << magenta <<  
      " " << yellow << " " << orange << endl;
```

```
17 18 19 20 21 22 23
```

Danger !

```
enum class Color {red, green, blue, cyan = 1, magenta, yellow, orange};
```

```
0 1 2 1 2 3 4
```

Values are repeated !!!

Using **enum** for **int** constants, anonymous unscoped **enum**

(Note: considered to be bad style, use **constexpr** instead)

```
enum {OK = 0, FILE_NOT_FOUND = 1234, BAD_DATA = 1235, IO_ERROR = 1236};
```

```
int i = readData();
```

```
switch (i) {
```

```
    ...
```

Enum underlying type

You can choose the underlying type of **enum**

```
enum Color1 : int {red, green, blue, cyan , magenta, yellow, orange};  
enum class Color2 : unsigned char {red, green, blue, cyan , magenta, yellow, orange};  
enum class Color3 : long long {red, green, blue, cyan , magenta, yellow, orange};  
sizeof(Color1) == 4  
sizeof(Color2) == 1  
sizeof(Color3) == 8
```

The default type is **int** (4 bytes)

You can use **unsigned char** (1 byte) to save memory !

unique_ptr and polymorphism (upcasts, downcasts) ?

Bear is a subclass of **Animal**. Can we do something like this ???

```
auto upB = make_unique<Bear>("Teddy", 7);  
auto upA = dynamic_cast<unique_ptr<Animal> >(upB);
```

unique_ptr and polymorphism (upcasts, downcasts) ?

Bear is a subclass of **Animal**. Can we do something like this ???

```
auto upB = make_unique<Bear>("Teddy");  
auto upA = dynamic_cast<unique_ptr<Animal> > (upB); // Wrong !!!
```

NO !!! This would be like copying **unique_ptr** !
You DO NOT upcast/downcast **unique_ptr** !

The correct way: use ***upB** as a reference or **upB.get()** as a raw pointer:

```
Animal & rA = *upB;           // Reference upcast. Good !  
Animal * pA = upB.get();      // Pointer upcast. Ugly ! Don't delete pA !
```

Or downcast with **dynamic_cast** :

```
unique_ptr<Animal>(new Bear("Teddy")); // Animal unique_ptr which holds a Bear  
Bear & rB = dynamic_cast<Bear &>(*upA); // Reference downcast. Good !  
Bear * pB = dynamic_cast<Bear *>(upA.get()); // Pointer downcast. Ugly ! Don't delete pB !
```

shared_ptr and polymorphism: upcasts and downcasts

```
auto sB = make_shared<Bear>("Teddy");    // Create a shared_ptr<Bear> object

cout << "Ref upcast :" << endl;
Animal & rA = *sB;
rA.talk();

cout << "Raw ptr upcast :" << endl;
Animal * pA = sB.get();                  // Get the raw Bear * ptr
pA->talk();

cout << "shared_ptr upcast :" << endl;
shared_ptr<Animal> sA = sB;               // Implicit upcast
sA->talk();

cout << "shared_ptr downcast :" << endl;
shared_ptr<Bear> sB1 = static_pointer_cast<Bear> (sA);    // No checks !
shared_ptr<Bear> sB2 = dynamic_pointer_cast<Bear> (sA);    // Checks. Safe.
sB1->talk();
sB2->talk();
```

title

text