C++ Raw and Smart Pointers

v1.0 By An7ar35

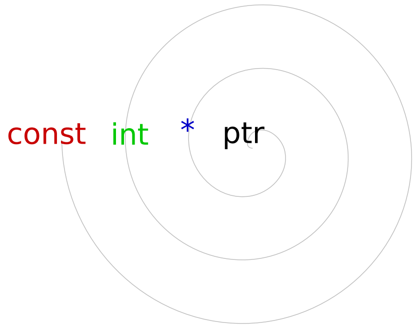
Contents

1 Raw pointers (a.k.a. The Dark Arts)

1.1 Reading declarations

1.1.1 Clockwise-Spiral Rule

The Clockwise-Spiral rule can be used to easily understand complex C-style declarations with pointer. To parse the definition just start with the main variable name (`ptr`) and move in a clockwise spiral fashion grabbing each elements as you encounter them.



On the right we start with `ptr` then move, in order, to: `*`, `int` and finally `const`. In plain english, this becomes:

`ptr` is a pointer (`*`) to an integer (`int`) that is constant (`const`).

In plain English...

Starting with the unknown element, move in a spiral/clockwise direction; when encountering the following elements replace them with the corresponding English statements:

`[n]` or `[]`	Array of `n` elements or array of undefined size
`(T,U)`	function passing type T and U
`*`	pointer to
`&`	reference to

Just remember to always resolve anything in parenthesis `(..)` first!

1.1.2 Example

```
int * (*compare)( double *, double * );
```

`(*compare)(...)`	`compare` is a pointer to a function,
`( double *, double * )`	taking 2 x pointers to `double`,
`int *`	returning a pointer to an `int`.

```
void (*execute)( int , void (*funct)( float , float ) );
```

`(*execute)(...)`	`execute` is a pointer to a function,
`( int, void (*funct)(...) )`	taking an `int` and a pointer to a function, `funct`,
`void (*funct)( float, float )`	which takes 2 x `float` and returns `void`,
`void (*execute)(...)`	returning `void`.

1.2 Raw pointers & references

A pointer simply holds a memory address where something is stored. It 'points' to the value in the memory.

```cpp
#include <iostream>

int main() {
    int val  = 666;  /* \m/ value */
    int *ptr = &val; /* pointer */
    int copy = val;  /* copy of value */

    std::cout << "val....: " << val << std::endl;   //value
    std::cout << "&val...: " << &val << std::endl;  //address of value
    std::cout << "ptr....: " << ptr << std::endl;   //pointer
    std::cout << "*ptr...: " << *ptr << std::endl;  //de-referenced pointer
    std::cout << "&ptr...: " << &ptr << std::endl;  //address of pointer
    std::cout << "copy...: " << copy << std::endl;  //copy of value
    std::cout << "&copy..: " << &copy << std::endl; //address of copy

    *ptr = 1000;
    std::cout << "val....: " << val << std::endl;   //modified value
    std::cout << "copy...: " << copy << std::endl;  //copy of original value

    return 0;
}
```

Output:

```
val....: 666
&val...: 0x7ffe6b67f3f8
ptr....: 0x7ffe6b67f3f8
*ptr...: 666
&ptr...: 0x7ffe6b67f400
copy...: 666
&copy..: 0x7ffe6b67f3fc
val....: 1000
copy...: 666
```

Explanations:

Line 4, 8	The value is declared, assigned (`666`) and stored in an address in memory (a 64bit address of `0x7ffe6b67f3f8` in this case). The address is runtime specific.
Line 5, 9	A pointer `ptr` is created to point to the address of the value. We pass `val`'s address using the address-of operator `&`.
Line 10	The address stored in the pointer `ptr` is the same as the address of the value `val`.
Line 11	To get to the value `val` from the pointer `ptr` we need to use the de-reference operator `*`
Line 12	Checking the address where pointer `ptr` resides in memory yield `0x7ffe6b67f400` which is not the same as `val`'s address.
Line 6, 13	A copy of the value is made.
Line 14	The address of the copy `0x7ffe6b67f3fc` is different than the original value's.
Line 16, 17	Assigning a new value (`1000`) to the de-referenced pointer `*ptr` modifies the original value from `666` to `1000`.
Line 18	The copy of the original value remains unchanged as it resides in a different memory location.

1.3 Null and invalid pointers

Code with un-initialised or invalid pointer, although discouraged, will compile. Undefined behaviour will happen during runtime if strong checks are not observed. For example the following will compile and run. Sometimes a segmentation fault will crash the program or, worse, undefined behaviour in the form of the pointer accessing random crap in the memory (i.e. invalid values).

```
int * ptr;                         /* un-initialised pointer */
std::cout << *ptr << std::endl; /* SEGFAULT or whatever is at the memory location */
```

Same goes for initialised pointers with invalid addresses like in the following example:

```
int array[] = { 111, 222, 333, 444 };
int * ptr = array + 10;         /* invalid pointer */
std::cout << *ptr << std::endl; /* whatever is at the memory location */
```

Safe coding practice when using raw pointer should make use of null initialisation at least when there are no other alternatives.

Null pointer initialisation

There are 3 ways to indicate a pointer is initialised:

```
int * ptr = 0;          /* C++ */
int * ptr = nullptr;    /* C++ - preferred */
int * ptr = NULL;       /* Defined constant NULL - old */
```

1.4 Using const with pointers

The `const` keyword is used to declare something as ‘constant’ (i.e. does not change).

To mark something as constant the keyword is used in the declaration. `const int` and `int const` are equivalent. A constant can only be set during declaration. Trying to define/re-define a `const` variable will yield a compile error (line 2).

```
1 const int i = 100; /* i is a constant int */
2 i = 1; /* Cannot assign to variable 'i' with const-qualitfied type 'const int' */
```

With pointers, the `const` keyword can be used in different ways.

We can declare the pointer `ptr` to be to a constant variable so that the variable cannot be changed via a pointer de-reference `*ptr` even though the variable itself `y` is not declared constant:

```
int x = 1;
int y = 2;
const int * ptr = &x; /* Pointer to a constant integer. */

 ptr = &y; /* OK. The pointer itself is not constant. */
*ptr = 3;  /* ERROR. Read-only variable is not assignable. */
   y = 3;  /* OK. The variable 'y' is not declared constant. */
```

If we declare the pointer to be constant instead the variable can be modified with de-referencing but the pointer itself cannot be re-assigned to another variable address:

```
int x = 1;
int y = 2;
int * const ptr = &x; /* Constant pointer to an integer. */

 ptr = &y; /* ERROR. Cannot assign to variable 'ptr' with const-qualified type
           'int *const'. */
*ptr = 3;  /* OK. The pointer type 'int' is not declared constant. */
```

When both the variable and pointer are declared constant in the pointer declaration then they cannot be modified thereafter:

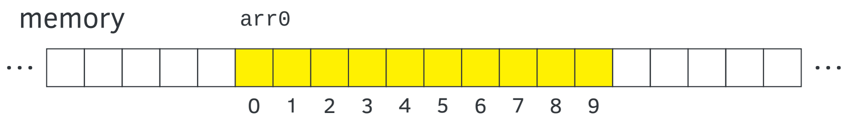
```c
int x = 1;
int y = 2;
const int * const ptr = &x;

 ptr = &y; /* ERROR. Cannot assign to variable 'ptr' with const-qualified type
              'const int *const'. */
*ptr = 3;  /* ERROR. Read-only variable is not assignable. */
   x = 3;  /* OK. The variable 'x' is not declared constant. */
```

1.5 Arrays

Arrays are a typed series of elements placed as a contiguous block in memory.

```c
int arr[10]; /* Declaring an integer array of size 10. */
```

Here we have an array `arr0` of 10 elements in memory that can be accessed by using an index `[0..9]`.



1.5.1 Declaration and Initialisation

Array declarations are done using the format:

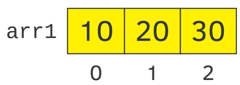
```
type name[n];
```

Where `type` is the element type stored in the array, `name` is the name of the array and `n` is the number element slots (space) that the array can accommodate.

Initialisation of an array can be done with a initialising list:

```c
int arr1[3] = { 10, 20, 30 };
```

Arrays cannot be initialised with more elements than the number declared in the brackets. Here, 3 elements are declared for `arr1[3]`



```
int arr2[3] = { 10, 20 };
```

When arrays are initialised with less than their size, the remaining elements are set to their default values. In the case of `int` it means `0`.

arr2	10	20	0
	0	1	2

```
int arr3[3] = { };
```

When initialised with no values then all elements are set to their default values.

arr3	0	0	0
	0	1	2

```
int arr4[] = { 10, 20, 30 };
int arr4[] { 10, 20, 30 }; /* equivalent using C++'s universal initialization style */
```

When an array is declared with no size then its size is deduced by the number of elements being it is initialised with.

arr4	10	20	30
	0	1	2

1.5.2 Array pointers

A pointer can be assigned an array:

```
int arr[10];
int *arr_ptr = arr; /* Pointer points to the array and can be reassigned later too. */
```

Assigning a pointer to an array variable is not valid:

```
arr = arr_ptr; /* NOT VALID. */
```

There are multiple approaches to access an element with or without a pointer:

```
char arr[] = "hello";
char *ptr_arr[5] = { &arr[0], &arr[1], &arr[2], &arr[3], &arr[0] };

std::cout << *ptr_arr << std::endl;          /* "hello" */
std::cout << arr[4] << std::endl;            /* 'o' */
std::cout << ( arr + 4 ) << std::endl;       /* 'o' */
std::cout << *ptr_arr + 4 << std::endl;      /* 'o' */
std::cout << ( *ptr_arr )[4]  << std::endl; /* 'o' */

(*ptr_arr)[0] = 'c';                  /* Modifying first element of 'arr' to 'c'. */
std::cout << *ptr_arr << std::endl; /* "cello" */
```

Assigning variable addresses to an array is also possible:

```
int a = 1;
int b = 2;
int c = 3;
int *arr[] = { &a, &b, &c }; /* 1, 2, 3 */

*arr[1] = 10; /* Modifies 'b' via the array pointer */

for( int i = 0; i < 3; i++ )
    printf( "%i ", *arr[i] );
}
```

Output:

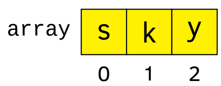
```
1 10 3
```

Array element access

E.g.: when `n` = 1, all will return `'k'`.

`array[n]`	Offset of `n` in array
`*(array + n)`	Pointed to by `(array + n)`

`(*pointer)[n]`	De-referenced pointer with array offset of `n`
`*pointer + n`	Pointed to by de-referenced pointer + `n`




```
char * pointer = array;
```

Size and length methods

`sizeof(T)`

Amount of memory (in bytes) that the variable or type `T` occupies.

`size_t strlen(const char *str)`

Returns the length of string `str` excluding the null-terminating character (`\0`).

E.g.:

```
char c       = 'h';
char array[] = "Hello!";

std::cout << sizeof(c) << std::endl;      /* 1 */
std::cout << sizeof(char) << std::endl;  /* 1 */
std::cout << sizeof(array) << std::endl; /* 7 */
std::cout << strlen(array) << std::endl; /* 6 */
```

1.6 Pointer arithmetic

Pointers can be incremented or decremented (`++` and `--` operators) based on the size of the data type pointed to.

Types take up different amount of space in memory based on their sizes (see your `stdint.h` header file). For example the `char` type takes up 1 byte in memory.

If we create an array from a string literal "Hello!", each `char` in the array will take 8 bit (or 1 Byte) of memory:

0 – 7	8 – 15	16 – 23	24 – 31	32 – 39	40 – 47	48 – 55
'H'	'e'	'l'	'l'	'o'	'!'	'\0'

The final `char` is the null-terminator automatically assigned for string literals by the compiler. It signifies the end of the contiguous sequence of characters.

To iterate across the sequence of characters using the pointer we can use the increment operator (`++`):

```
#include <iostream>
#include <cstring>

int main() {
    char array[] = "Hello!";
    char * ptr = array;

    for( int i = 0; i < sizeof(array); i++ ) {
        printf( "%c", *ptr ); /* Prints a literal char (%c) */
        ptr++;                /* Increment pointer (same as ptr = ptr + 1)*/
    }

    return 0;
}
```

The code results in the following being printed:

```
Hello!
```

1.6.1 Operators

`ptr++`	Post-increment: returns pre-increment (old) pointer.
`++ptr`	Pre-increment: returns post-increment (new) pointer.
`ptr--`	Post-decrement: returns pre-decrement (old) pointer.
`--ptr`	Pre-decrement: returns post-decrement (new) pointer.

1.6.2 Ordering

Postfix operators have higher precedence than prefix operators.

Operator ordering rule of thumbs

Move right then move left of variable.

E.g.: `++*ptr--`:

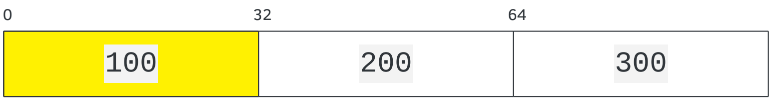
Step 1	`ptr--`	Post-decrement pointer, return pre-decrement pointer.
Step 2	`*ptr`	De-reference the pointer to access value.
Step 3	`++value`	Pre-increment the value, returning the post-incremented value.

```c
int  array[] = { 100, 200, 300 };
int * ptr = array + 1;

printf( "%i\n", *ptr ); /* Pointer at [1]: 200 */
int i = ++*ptr--;       /* i.e.: ++(*(ptr--))) */
printf( "%i\n", i );    /* Value at [1]: 201   */
printf( "%i\n", *ptr ); /* Pointer at [0]: 100 */
```

Here are the different possible combination of dereference and pre/post-fix operators:

`*ptr++`	or `*(ptr++)`: Increment `ptr`, dereference unincremented (old) address
`*++ptr`	or `*(++ptr)`: Increment `ptr`, dereference incremented (new) address
`++*pt`	or `++(*ptr)`: De-reference `ptr`, increment the value it points to
`(*ptr)++`	De-reference `ptr`, post-increment the value it points to

Example:




```c
int  array[] = { 100, 200, 300 };
int * ptr = array;          /* pointer at [0] */

printf( "%i\n", *ptr++ );   /* prints [0], pointer at [1] */
printf( "%i\n", *++ptr );   /* prints [2], pointer at [2] */
printf( "%i\n", ++*ptr );   /* increments [2]'s value to '301' then prints */
printf( "%i\n", (*ptr)++ ); /* prints [2] then increments [2]'s value to '302' */
```

Explanations:

Line 4	The value is accessed (`100`) then the pointer incremented after.
Line 5	The pointer is incremented first, then the value accessed (`300`).
Line 6	The value is incremented and then accessed (`301`).
Line 7	The value is accessed (`301`) then incremented.

1.7 Void pointers

Void pointers is something to be avoided if at all possible in best-practice C++.

A void pointer `void *` is a type-less pointer. The value's length and de-referencing properties it points to are unknown. In order to de-reference a void pointer it must first be cast as a typed pointer matching the data type of what it points to.

```
int a = 666;
void * ptr = &a;
std::cout << *( (int *) ptr ) << std::endl;
std::cout << *( static_cast<int*>( ptr ) ) << std::endl; /* Since we know it's an int */
```

1.7.1 Generic function using void pointers

Void pointers can be used to pass generic parameters to a function:

```
#include <iostream>

enum class Type {
    INT, DOUBLE, STR
};

void printer( void * value, Type type ) {
    switch( type ) {
        case Type::INT:
            std::cout << *( static_cast<int*>( value ) ) << std::endl;
            break;
        case Type::DOUBLE:
            std::cout << *( static_cast<double*>( value ) ) << std::endl;
            break;
        case Type::STR:
            std::cout << *( static_cast<std::string*>( value ) ) << std::endl;
            break;
    }
}

int main() {
    int i = 666;
```

```
    double d = 1.2345;
    std::string s { "Hello, modern world!" };

    printer( &i, Type::INT );         /* 666 */
    printer( &d, Type::DOUBLE );      /* 1.2345 */
    printer( &s, Type::STR );         /* Hello, modern world! */
    return 0;
}
```

1.7.2 Templated function (alternative)

A much better alternative would be using **templates** for compile-time type-inference.

```
template<class T> void printer( T * value ) {
    std::cout << *value << std::endl;
}

int main() {
    int i = 666;
    double d = 1.2345;
    std::string s { "Hello, modern world!" };

    printer<int>( &i );
    printer<double>( &d );
    printer<std::string>( &s );
    return 0;
}
```

1.8 Functions

1.8.1 Function pointers

It is possible to have pointer to functions. This makes passing functions as arguments to other functions possible. To declare a pointer to a function:

`return_type (*pointer_name)( /* function_arguments */ ) = function_name;`.

In practice:

```
int compare( int a, int b ) { /* ... */ }

int main() {
    int (*compareMethod_ptr)( int, int ) = compare;
    printf( "%i", compareMethod_ptr( 10, 100 ) );
    return 0;
}
```

In C++, `std::function` should be preferred instead. (see Appendix A for example)

1.8.2 Example

Here we are passing a comparison function `compare(..)` to a printer function `printer(..)` in order to print the result of comparing indexed element in 2 arrays (`array1` and `array2`).

```cpp
#include <iostream>

int compare( const int &a, const int &b ) {
    if( a > b )
        return 1;
    if( a < b )
        return -1;
    return 0;
}

void printer( int a, int b, int (*function)( const int &a, const int &b ) ) {
    printf( ”%i\n”, function( a, b ) );
}

int main() {
    const size_t arr_size = 3;
    int array1[arr_size] = { 1, 2, 3 };
    int array2[arr_size] = { 1, 3, 2 };

    for( int i = 0; i < arr_size; i++ ) {
        printf( ”Comparison between %i and %i: ”, array1[i], array2[i] );
        printer( array1[i], array2[i], compare );
    };

    return 0;
}
```

Output:

```
Comparison between 1 and 1: 0
Comparison between 2 and 3: -1
Comparison between 3 and 2: 1
```

2 C++ Smart Pointers

Smart pointer were introduced to C++ in order to provide a safe and managed way to deal with pointer and help with garbage collection using RAII (Resource Acquisition Is Initialization).

2.1 Unique pointers (`unique_ptr<T>`)

Unique pointers (`std::unique_ptr<T>`) are used to enforce explicit ownership of objects and provides an automatic way of triggering a disposal event (default or custom) of the object via the `Deleter` when it goes out of scope.

```
/* For a single object */
template <class T, class Deleter = std::default_delete<T>> class unique_ptr;
/* For a dynamically-allocated array of objects */
template <class T, class Deleter> class unique_ptr<T[], Deleter>;
```

The `Deleter` is triggered:

1. when the `unique_ptr` is destroyed or
2. when the `unique_ptr` is assigned another pointer.

2.1.1 Construction and initialisation

```
/* Heap allocation */
auto p = std::make_unique<T>( /* T constructor args */ );          /* Preferred */
auto p = std::unique_ptr<T>( new T( /* T constructor args */ )
```

Using class `A` (see Appendix B):

```
auto p0 = std::unique_ptr<A>();                 /* Empty unique_ptr<A> */
auto p1 = std::make_unique<A>();                /* A's default constructor */
auto p2 = std::make_unique<A>( "Hello!" );  /* A's constructor with argument */
```

2.1.2 Observers/Modifiers

Note: Great care needs to be taken when making a copy of a smart pointer's raw pointer. The raw pointer copy can end up becoming a dangling pointer (i.e.: points to nothing) when the `Deleter` is called. This is why it should be avoided in practice.

```
/* Accessing object */
A copy = *p1;                /* Copying p1's object */
A * temp = p0.get();         /* AVOID — Getting access to the raw pointer */

/* Moving object ownership */
p0.reset( p1.release() );    /* Passing ownership of p1's object to p0 */
p0 = std::move( p1 );        /* Better alternative passing ownership */
p1.swap( p2 );               /* Swapping objects owners */

/* Checking if there is an associated managed object */
if( p1 )
    std::cout << *p1 << std::endl;
```

2.1.3 Destruction

If no deleter is supplied, the default one will destroy the object and deallocate the memory by using the `delete` operator.

```
/* Out-of-Scope destruction */
void funct() {
    auto p = make_unique<T>();  /* Creates 'T' object managed by 'p' */
    /* ... using 'p' ... */
}                               /* Scope ends, deleter is called, object is destroyed */

/* Called destruction */
p1.reset();                  /* Calls the deleter on the object managed by p1 */
p1.reset( nullptr );         /* Same as above */

/* Ownership release */
A * ptr = p2.release()       /* AVOID — Removes ownership responsibility from p2 */
```

2.1.4 Example

In this example below, we are using a simple class `A` (see Appendix B). Using the overloaded `++()` operator, we can increase a 'print' counter within it.

A `printer(...)` method (lines 5-9) prints `A`'s string variable and increments the print counter.

The program (lines 11-17) creates a pointer to an class `A` object `a` and sends it to the printer method using move-semantics (`std::move(..)`).

Every time the pointer is passed to the `printer(..)` method the ownership of the pointer is transferred to the method (line 16). The method then passes the ownership back via `return` to the caller (lines 8 and 16); This back and forth happens on each iteration of the `for` loop.

```
1  #include <iostream>
2  #include <memory>
3  #include "A.h"
4
5  std::unique_ptr<A> printer( std::unique_ptr<A> a ) {
6      std::cout << a->str() << std::endl;
7      ++(*a); /* increment print count */
8      return std::move( a );
9  }
10
11 int main() {
12     auto a = std::make_unique<A>( "Hello!" );
13     for( int i = 0; i < 5; i++ )
14         a = printer( std::move( a ) );
15
16     std::cout << *a << std::endl;
17 }
```

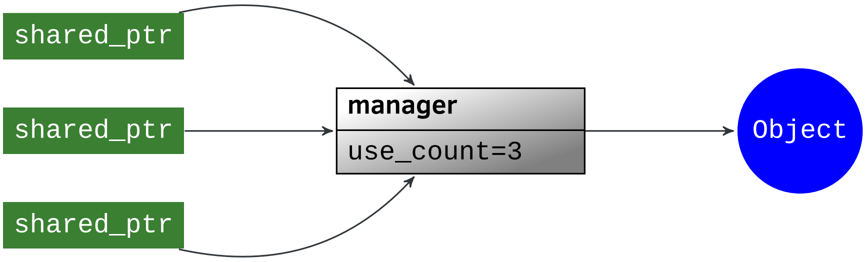
Output:

```
A::A( std::string ) constructor.
Hello!
Hello!
Hello!
Hello!
Hello!
"Hello!" printed 5 times.
A::~A() destructor.
```

2.2 Shared pointers (`shared_ptr<T>`)

Shared pointers are used to managed shared resources and provides an automatic way of triggering a disposal event (default or custom) of the object via the `Deleter` when it goes out of scope.

```
template<class T> class shared_ptr;
```



The `Deleter` is triggered at `use_count = 0`:

1. when the last `std::shared_ptr` pointing to the shared object is destroyed or
2. when the last `std::shared_ptr` pointing to the shared object is assigned another pointer.

2.2.1 Construction and initialisation

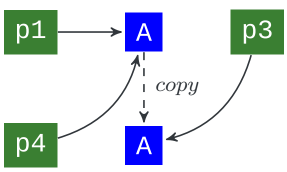
```
auto p = std::make_shared<T>( /* constructor args */ );          /* Preferred */
auto p = std::shared_ptr<T>( new T( /* constructor args */ ) );
```

Using class `A` (see Appendix B):

```
1 auto p0 = std::unique_ptr<A>();              /* Empty shared_ptr<A> */
2 auto p1 = std::make_shared<A>();             /* A's default constructor, use_count=1 */
3 auto p2 = std::make_shared<A>( "Hello!" );   /* A's constructor with argument */
4
5 auto p3 = std::make_shared<A>( *p1 );        /* Copies p1's managed object */
6 auto p4 = std::shared_ptr<A>( p1 );          /* Copy of shared pointer, use_count=2 */
```

Notice that when passing a managed object to a new shared pointer, a copy of that object will be created for the new shared pointer to manage (line 5).

To create pointers with a shared `use_count`, the previous instance must be passed to the constructor (line 6).



2.2.2 Observers/Modifiers

```
/* Accessing object */
A copy = *p1;                /* Copying p1's object */
A * temp = p0.get();         /* AVOID — Getting access to the raw pointer */

/* Sharing object ownership */
p0 = p1;                     /* Sharing ownership of p1's object with p0 */
p1.swap( p2 );               /* Swapping managed object */

/* Checking if there is an associated managed object */
if( p1 )
    std::cout << *p1 << std::endl;
```

2.2.3 Destruction

Object destruction is triggered when the `use_count` hits `0` on a managed object.

```
auto p0 = std::make_shared<A>( "Instance of A" );   /*  use_count() = 1 */
auto p1 = p0;                                        /*  use_count() = 2 */
auto p2 = p0;                                        /*  use_count() = 3 */

p0.reset();          /* use_count() = 2 */
p1.reset();          /* use_count() = 1 */
p2.reset();          /* use_count() = 0, managed object 'A' destroyed. */
```

2.2.4 Example

```
auto p0 = std::make_shared<A>( "Hello from P0!" );
auto p1 = std::make_shared<A>( "Hello from P1!" );
```

If we assign `p1` to `p0`, `p0` becomes now points to the same managed object as `p1`. `p0`'s previously managed object is destroyed as no other shared pointers points to it (`use_count=0`).

```
p0 = p1;
std::cout << *p0 << std::endl;
std::cout << *p1 << std::endl;
```

Output:

```
A::A( std::string ) constructor.
"Hello from P0!" printed 0 times.
"Hello from P0!" printed 0 times.
A::~A() destructor.
```

2.3 Custom deleters

There are 3 ways to construct a deleter:

- **function**

```
void deleter( T *ptr ) { /* ... */ }

int main() {
    auto p = std::unique_ptr<T, std::function<void( T * )>>( new T(), deleter );
    /* ... */
}
```

- **lambda**

```
auto deleter = []( T *ptr ) { /* ... */ }
auto p = std::unique_ptr<T, decltype( deleter )>( new T(), deleter );

/* OR */

auto q = std::unique_ptr<T, std::function<void( T * )>>(
    new T(),
    []( T * ptr ) { /* ... */ }
);
```

- **struct**

```
struct Deleter {
    void operator()( T *ptr ) { /* ... */ }
}

int main() {
    auto p = std::unique_ptr<T, Deleter>( new T(), Deleter );
    /* ... */
}
```

2.3.1 Example

Here we are using a lambda function as the deleter (lines 10-14). `deleter` increments a garbage counter variable (line 9-10) every time it is used (i.e.: when a `std::unique_ptr` managing a class `A` instance is reset or destroyed).

The `typedef` at line 15 declares an alias for the `std::unique_ptr` with the custom deleter definition. The vector is then declared using this alias (line 16).

Line 18-22 creates and adds 5 $\times$ `std::unique_ptr` in the vector using the custom deleter function.

When the `main()` method goes out of scope (i.e. when the program ends), the destructors are called. As these are called, the custom deleter is used on each `std::unique_ptr` in the vector.

```
#include <iostream>
#include <memory>
#include <functional>
#include <mutex>
#include "A.h"

int main() {
    std::function deleter = [&]( A *ptr ) {
        static unsigned garbage_counter; /* static variable, only exists once */
        garbage_counter++;
        std::default_delete<A>{}( ptr ); /* calls the default deleter on the object pointer
        std::cout << "garbage counter now at: " << garbage_counter << std::endl;
    };

    typedef std::unique_ptr<A, decltype( deleter )> PtrA_t;
    std::vector<PtrA_t> v;

    for( int i = 0; i < 5; i++ )
        v.emplace_back(
            new A(),
            deleter
        );

    std::cout << "-------------[END]-------------" << std::endl;
    return 0;
}
```

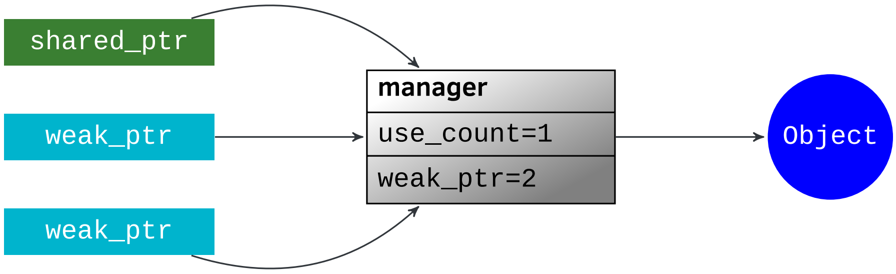
Output:

```
A::A() constructor.
A::A() constructor.
A::A() constructor.
A::A() constructor.
A::A() constructor.
-------------[END]-------------
A::~A() destructor.
garbage counter now at: 1
A::~A() destructor.
garbage counter now at: 2
A::~A() destructor.
garbage counter now at: 3
A::~A() destructor.
garbage counter now at: 4
A::~A() destructor.
garbage counter now at: 5
```

2.4 Weak pointers (`weak_ptr<T>`)

Weak pointers are used in tandem with `std::shared_ptr` as a non-owning (and non-managing) pointer. It enables safe access where temporary access to the managed object is needed.

```
template<class T> class weak_ptr;
```



Weak pointers are used in the case of circular references (see section 3.2).

2.4.1 Construction and initialisation

```
auto p = std::weak_ptr<T>();
auto p = std::weak_ptr( const weak_ptr<T> &r );
auto p = std::weak_ptr( const std::shared_ptr<T> &r );
auto p = std::weak_ptr( weak_ptr<T> &&r );
```

Using class `A` (see Appendix B):

```
auto s = std::make_shared<A>( "Hello!" );   /* shared pointer */

auto w = std::weak_ptr( s );                /* weak pointer to 's'' managed object */
auto copy = std::weak_ptr( w );             /* copy of weak pointer 'pw' */
```

The lifetime of the managed object is still being taken care of by the `shared_ptr`. If that goes out of scope or is reset then conversion of the `std::weak_ptr` to a `std::shared_ptr` will fail.

2.4.2 Observers/Modifiers

```
auto s = std::make_shared<A>( "Hello!" );   /* shared pointer */
auto w = std::weak_ptr( s );                /* weak pointer */

/* Accessing managed object */
auto shared_ptr = w.lock(); /* Creating a shared pointer from weak pointer 'p' */
bool expired = w.expired(); /* Checks existance of managed object */

/* Checking if there is an associated managed object */
if( shared_ptr )
    std::cout << *shared_ptr << std::endl;
/* OR */
if( auto temp = w.lock )
    std::cout << *temp << std::endl;
```

2.4.3 Destruction

```
auto s = std::make_shared<A>( "Hello!" );   /* shared pointer */
auto w = std::weak_ptr( s );                /* weak pointer to 's'' managed object */

w.reset();  /* w.expired() = 1, 's' is still managing the object */
```

2.4.4 Example

Lines 5-8 is a method that takes in a `std::weak_ptr` by value and prints `.str()` if the managed object still exists.

Lines 10-14 is a method that creates local shared pointer to created class `A` object and return a `std::weak_ptr` to it. As the shared pointer `ptr` goes out of scope, it will be destroyed along the managed object at the end of the method.

In `main()`:

- PART I: Lines 19-20 create a shared pointer to a local object and then send a weak pointer made from it to the `printer` method.
- PART II: Line 24 calls the `createWeakPtrOnStack(..)` method to get back a weak pointer to the object created on the stack of the method.
  Lines 26-27 tests the validity of the shared pointer's managed object that `weak` points to and prints `.str()` if the test passes (it won't).

```cpp
#include <iostream>
#include <memory>
#include "A.h"

void printer( std::weak_ptr<A> p ) {
    if( auto temp_shared = p.lock() )
        std::cout << temp_shared->str() << std::endl;
}

std::weak_ptr<A> createWeakPtrOnStack( const std::string &value ) {
    auto ptr = std::make_shared<A>( value );
    std::cout << "Object A on function stack: " << ptr->str() << std::endl;
    return std::weak_ptr( ptr );
}

int main() {
    std::cout << "-----------[PART I]-----------" << std::endl;
    /* Local shared pointer sent to printer function as a weak pointer in scope */
    auto ptr = std::make_shared<A>( "Managed object" );
    printer( std::weak_ptr( ptr ) );

    std::cout << "-----------[PART II]-----------" << std::endl;
    /* Weak pointer to shared pointer destroyed when function is out-of-scope */
    auto weak = createWeakPtrOnStack( "Function object" );

    if( auto temp = weak.lock() ) /* failed as no more managed object */
        std::cout << temp->str() << std::endl;

    std::cout << "-------------[END]-------------" << std::endl;
    return 0;
}
```

Output:

```
-----------[PART I]-----------
A::A( std::string ) constructor.
Managed object
-----------[PART II]-----------
A::A( std::string ) constructor.
Object A on function stack: Function object
A::~A() destructor.
-------------[END]-------------
A::~A() destructor.
```

3 Problems to avoid

3.1 Stack allocation

Concerns:

- raw pointers (`*`),
- `std::unique_ptr`,
- `std::shared_ptr`.

An issue arises when we create a pointer to an object on the stack and pass it outside its scope:

```cpp
/* Stack allocation */
std::unique_ptr<int> stackAlloc() {
    int i = 666;                                  /* int on function stack */
    auto p = std::unique_ptr<int>( &i );          /* creating unique_ptr to 'i' */
    return std::move( p );                        /* return unique_ptr to object on stack */
}

int main() {
    auto p = stackAlloc();
    std::cout << *p << std::endl;                 /* function stack and 'i' are gone! */
    return 0;
}                                                 /* ERROR: free(): invalid pointer */
```

The equivalent function using a raw pointer:

```cpp
int * stackAlloc() {
    int i = 666;
    return &i;                                    /* value escapes local scope */
}
```

When a pointer to an stack object is passed outside of the function's scope we end up with a dangling pointer (i.e.: pointing to invalid memory). When the function ends, the function stack unwinds and all objects created therein are disposed of. Thus, the object pointed to by the pointer is gone and we end up with a crash at best or undefined behaviour at worse.

Any object created in a function that will be shared outside its scope should be allocated on the **heap** instead (e.g.: using `std::make_unique<T>(...)` or `std::unique<T>(new T(...))`).

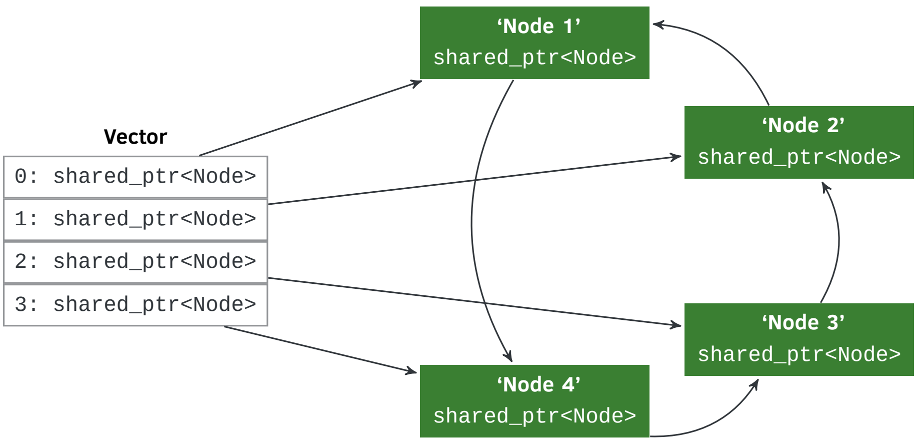
3.2 Circular referencing

Concerns:

- `std::shared_ptr`,
- raw pointers (`*`) - *potentially when using clearing up without being careful.*

Circular referencing can be a problem with reference-counting pointers. If there is a loop between objects that have referencing pointers to each other it create a cyclic dependency. In short, it creates an issue when calling destructors as each pointer in the loop keeps each other alive.

E.g.: A container (`std::vector` in this case) holds shared pointers to nodes in a graph. Each of the nodes has a shared pointer that references the previous connected node. If we close the loop by linking up the last node to the first one we have a cyclic dependency.



Even when the container is destroyed the node objects won't be as they hold references to each other. The result is that we end up with a memory leak.

Using class `Node` (see Appendix C):

```
#include <iostream>
#include <memory>
#include <vector>
#include "Node.h"

int main() {
    std::vector<std::shared_ptr<Node>> v {};
    auto first  = v.emplace_back( std::make_shared<Node>( "Node 1" ) );
    auto second = v.emplace_back( std::make_shared<Node>( "Node 2", first ) );
    auto third  = v.emplace_back( std::make_shared<Node>( "Node 3", second ) );
    auto fourth = v.emplace_back( std::make_shared<Node>( "Node 4", third ) );
    first->attach( fourth ); /* Loop complete */

    /* Iterating through the vector */
    std::cout << "----------[VECTOR CONTENT]----------" << std::endl;
```

```
    for( const std::shared_ptr<Node> &n : v )
        std::cout << n->str() << std::endl;

    /* Iterating 2x through nodes using 'previous()' */
    std::cout << "---------[NODE ITERATION]---------" << std::endl;
    auto current = first;
    for( auto i = 1; i < ( v.size() * 2 ); ++i ) {
        std::cout << i << ": " << current->str() << std::endl;
        current = current->previous();
    }

    std::cout << "-------------[END]-------------" << std::endl;
    return 0;
}
```

Output:

```
---------[VECTOR CONTENT]---------
Node 1
Node 2
Node 3
Node 4
---------[NODE ITERATION]---------
1: Node 1
2: Node 4
3: Node 3
4: Node 2
5: Node 1
6: Node 4
7: Node 3
-------------[END]-------------
```

Notice that the `Node::~Node()` destructor is never called when the program ends (and the vector `v` goes out of scope).

3.2.1 Solution

We can remedy this problem by changing just a couple of things:

1. Modify the `Node` class (Appendix C) to use `std::weak_ptr<Node>` to hold reference to the previous node in the graph:

```cpp
class Node {
  public:
    /* ... */
    std::weak_ptr<Node> & previous();
    /* ... */
  private:
    /* ... */
    std::weak_ptr<Node> _previous;
};
```

2. Change the `for` loop in the program (Lines 22-25) to:

```cpp
for( auto i = 1; i < ( v.size() * 2 ); ++i ) {
    std::cout << i << ": " << current->str() << std::endl;
    current = current->previous().lock();
}
```

<u>Output:</u>

```
---------[VECTOR CONTENT]---------
Node 1
Node 2
Node 3
Node 4
---------[NODE ITERATION]---------
1: Node 1
2: Node 4
3: Node 3
4: Node 2
5: Node 1
6: Node 4
7: Node 3
-------------[END]-------------
Node::~Node() destructor for "Node 1"
Node::~Node() destructor for "Node 2"
Node::~Node() destructor for "Node 3"
Node::~Node() destructor for "Node 4"
```

4 Change Log

Release	Section	Sub-section	Change description
1.0	1. Raw Pointers	-	Initial commit
	2. Smart Pointers	-	Initial commit
	3. Problems to avoid	-	Initial commit
	Appendix	-	Initial commit

Appendices

A Using `std::function` instead of function pointers

```cpp
struct Printer {
    void operator ()( const std::string &s ) {
        std::cout << "from Printer struct using operator(): " << s << std::endl;
    }
};

void apply( const std::string &s, std::function<void(const std::string &)> f ) {
    f( s );                         /* Applying the function on string 's' */
}

int main() {
    /* Lambda function */
    auto f = []( const std::string &s ) {
        std::cout << "from lambda function: " << s << std::endl;
    };

    f( "local hello!" );            /* call lambda function */
    apply( "Hello!", f );           /* passing lambda function to 'apply' method */
    apply( "Hello!", Printer());    /* passing Printer struct to 'apply' method */

    return 0;
}
```

Output:

```
from lambda function: local hello!
from lambda function: Hello!
from Printer struct using operator(): Hello!
```

B Code for class 'A'

Class `A` includes implementations for copy/move constructors and assignment operators.

B.1 Header file (`A.h`)

```
#ifndef A_H
#define A_H

#include <iostream>
#include <memory>

class A {
  public:
    A();
    explicit A( std::string value );
    ~A();
    A( A &&a ) noexcept;
    A( const A &a );

    friend std::ostream& operator <<( std::ostream &out, const A &a );

    A & operator =( A const &rhs);
    A & operator =( A &&rhs ) noexcept;
    A & operator ++();
    A   operator ++( int );

    std::string str() const;
    size_t printCount() const;

  private:
    std::string _s;
    unsigned    _print_count;
};

#endif //A_H
```

B.2 Implementation file (A.cpp)

```cpp
#include "A.h"

A::A() :
    _s( "" ),
    _print_count( 0 )
{
    std::cout << "A::A() default-constructor." << std::endl;
};

A::A( std::string value ) :
    _s( std::move( value ) ),
    _print_count( 0 )
{
    std::cout << "A::A( std::string ) constructor." << std::endl;
}

A::~A() {
    std::cout << "A::~A() destructor." << std::endl;
}

A::A( A &&a ) noexcept :
    _s( std::move( a._s ) ),
    _print_count( a._print_count )
{
    std::cout << "A::A( A && ) move-constructor." << std::endl;
}

A::A( const A &a ) :
    _s( a._s ),
    _print_count( a._print_count )
{
    std::cout << "A::A( const A & ) copy-constructor." << std::endl;
}

std::ostream& operator <<( std::ostream &out, const A &a ) {
    out << "\"" << a._s << "\" printed " << a._print_count << " times.";
    return out;
}

A &A::operator =( A const &rhs ) {
    std::cout << "A::operator=( A const & ) copy-assignment." << std::endl;
    if( &rhs != this ) {
        _s = rhs._s;
        _print_count = rhs._print_count;
    }
    return *this;
```

```
}

A &A::operator =( A &&rhs ) noexcept {
    std::cout << "A::operator=( A && ) move-assignment." << std::endl;
    _s = std::move( rhs._s );
    _print_count = rhs._print_count;
    return *this;
}

A &A::operator ++() {
    ++_print_count;
    return *this;
}

A A::operator ++( int ) {
    A temp = *this;
    ++_print_count;
    return temp;
}

std::string A::str() const {
    return _s;
}

size_t A::printCount() const {
    return _print_count;
}
```

C Code for class ‘Node’

C.1 Header file (Node.h)

```
#ifndef NODE_H
#define NODE_H

#include <string>
#include <memory>

class Node {
  public:
    explicit Node( std::string name );
    Node( std::string name, std::shared_ptr<Node> & node );
    ~Node();

    void attach( std::shared_ptr<Node> &node );
    std::shared_ptr<Node> & previous();
    std::string str() const;

  private:
    std::string _name;
    std::shared_ptr<Node> _previous;
};

#endif //NODE_H
```

C.2 Implementation file (`Node.cpp`)

```
#include "Node.h"

Node::Node( std::string name ) :
    _name( std::move( name ) )
{}

Node::Node( std::string name, std::shared_ptr<Node> &node ) :
    _name( std::move( name ) ),
    _previous( node )
{}

Node::~Node() {
    std::cout << "Node::~Node() destructor for \"" << _name << "\"" << std::endl;
}

void Node::attach( std::shared_ptr<Node> &node ) {
    _previous = node;
}

std::shared_ptr<Node> & Node::previous() {
    return _previous;
}

std::string Node::str() const {
    return _name;
}
```