Smart Pointers, operator overloading "->", new and delete

Pal Adrian

Smart Pointers

- In modern C++ programming, the Standard Library includes *smart pointers*, which are used to help ensure that programs are free of memory and resource leaks and are exception-safe.
- Smart pointers are defined in the `std` namespace in the <memory> header file. They are crucial to the RAII or *Resource Acquisition Is Initialization* programming idiom.
- In most cases, when you initialize a raw pointer or resource handle to point to an actual resource, pass the pointer to a smart pointer immediately. In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.

The following example compares a raw pointer declaration to a smart pointer declaration.

```
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}

void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

"->" overloading operator

- The operator -> is used to overload member access.
- The class member access operator (->) can be overloaded but it is bit trickier. It is defined to give a class type a "pointer-like" behavior. The operator -> must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply.
- If the return value is another object of class type, not a pointer, then the subsequent member lookup is also handled by an `operator->` function. This is called the "drill-down behavior." The language chains together the `operator->` calls until the last one returns a pointer.

Example 1

```
#include <iostream>
struct A
{
    void foo() {std::cout << "Hi" << std::endl;}
};

struct B
{
    A a;
    A* operator->() {
        return &a;
    }
};

int main() {
    B b;
    b->foo();
}
```

This outputs:

```
-Hi
```

Example 2

```
struct client
    { int a; };

struct proxy {
    client *target;
    client *operator->() const
        { return target; }
};

struct proxy2 {
    proxy *target;
    proxy &operator->() const
        { return * target; }
};

void f() {
    client x = { 3 };
    proxy y = { & x };
    proxy2 z = { & y };

    std::cout << x.a << y->a << z->a; // print "333"
}
```

New operator

- The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
- **Syntax to use new operator**: To allocate memory of any data type, the syntax is:

```
pointer-variable = new data-type;
```

Example

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;

            OR

// Combine declaration of pointer
// and their assignment
int *p = new int;
```

Delete operator

- Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.
  Syntax:

```
// Release memory pointed by pointer-variable
     delete pointer-variable;
```

Example

- `delete p;`
- `delete q;`

To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;
```

Example:

```
   // It will free the entire array
   // pointed by p.
   delete[] p;
```