

Smart Pointers

C++ 11 standard solved a major safety problem with the language: Managing memory

- Creating memory
- Deleting memory
- Managing who owns memory and how many owners are there?

Example

Problematic code

Memory becomes difficult to manage when you give the pointers or references away. Here is an example.

```
// Interface for an internet socket API
class socket_i {
public:
    virtual void send() = 0;
    virtual void recv() = 0;
    virtual ~socket_i() {}
};

// Child class that implements an interface
class socket : public socket_i {
public:
    void send() override {
    }

    void recv() override {
    }
};

// An HTTP library that uses a socket interface to communicate
class http {
public:
```

```

    http(socket_i *sock) : m_socket(sock) {
    }
private:
    socket_i *m_socket;
};
// Multiple problems with this code
http* create_http() {
    socket s;
    http http_obj(&s);
    return http_obj;
}
int main(void) {
    http* http_obj = create_http();}

```

One of the problems can be simplified to this:

```

int* get() {
    int x;
    return &x;
}
void problem() {
    int* pointer = get();
    *pointer = 123;}

```

Fixed Version 1

The partially corrected version creates a new memory reference, and it leaves it up to the consumer of the code to actually delete the object. This is just one of the issues: memory management problem. Because you are relying on the user of the code to delete your memory, it creates traps in your code.

```

// Let us partially fix our code
http* create_http() {
    socket s;
    // Let us create new memory which will not go out of scope
    // even when this function exits
    // BUT:
    // We have now created memory management problem:

```

```

// ie:
// - Who deletes this memory?
// - When do they do it?
// - Can you ensure that they delete it?
http * http_obj = new http(&s);
return http_obj;
}
int main(void) {
    http* http_obj = create_http();

    // We better delete our object or else it is memory leak
    delete http_obj;}

```

std::unique_ptr

Let us introduce the concept of using a unique pointer such that we do not have to worry about deleting it.

```

// Let us partially fix our code
std::unique_ptr<http> create_http() {
    socket s;
    std::unique_ptr<http> http_obj = std::make_unique<http>(&s);
    return http_obj;
}
int main(void) {
    std::unique_ptr<http> http_obj = create_http();
    // No need to worry about deleting our allocated resource
    //delete http_obj}

```

The unique_ptr is also special because it will force you to maintain "one owner".

std::shared_ptr

If we continue in our example, we will notice that there is still a fundamental flaw which is that the socket that was created would go out of scope, and the http class would then end up using a reference to memory that no longer is alive (no longer in scope).

```

// Interface for an internet socket API

```

```

class socket_i {
public:
    virtual void send() = 0;
    virtual void recv() = 0;
    virtual ~socket_i() {}
};
// Child class that implements an interface
class socket : public socket_i {
public:
    void send() override {
    }

    void recv() override {
    }
};
// An HTTP library that uses a socket interface to communicate
class http {
public:
    http(std::shared_ptr<socket_i> sock) : m_socket(sock) {
    }
private:
    std::shared_ptr<socket_i> m_socket;
};

```

Cleaned up code

```

#include <iostream>
#include <memory>
// Interface for an internet socket API
class socket_i {
public:
    virtual bool open(const std::string& hostname) = 0;
    virtual void send(const std::string& data_to_transit) = 0;
    virtual std::string recv() = 0;
    virtual ~socket_i() {}
};

```

```

// An HTTP library that uses a socket interface to communicate
class http {
public:
    http(std::shared_ptr<socket_i> sock) : m_socket(sock) {
    }
    ~http() {
        std::cout << "Destructor of http class has been called" << std::endl;
    }
    void send_request(const std::string& host, const std::string& resource) {
        std::string request = "GET " + resource + " HTTP/1.1\r\n";
        request += "Host: " + host + "\r\n";
        request += "Connection: close\r\n\r\n";
        m_socket->send(request);
        std::string response = m_socket->recv();
        std::cout << "Response:\n" << response << std::endl;
    }
private:
    std::shared_ptr<socket_i> m_socket;
};

class linux_socket : public socket_i {
public:
    bool open(const std::string& hostname) override {
        return false;
    }
    void send(const std::string& data_to_transit) override {
    }

    std::string recv() override {
        return std::string{};
    }
};

// Let us partially fix our code
std::unique_ptr<http> create_http() {
    auto socket = std::make_shared<linux_socket>();
    // References to the shared pointer:
    std::cout << "reference count before: " << socket.use_count() << std::endl;
}

```

```
// Let us create new memory which will not go out of scope
// even when this function exits
// BUT:
// We have now created memory management problem:
// ie:
// - Who deletes this memory?
// - When do they do it?
// - Can you ensure that they delete it?
std::unique_ptr<http> http_obj = std::make_unique<http>(socket);
std::cout << "reference count after: " << socket.use_count() << std::endl;
return http_obj;
}
int main(void) {
    auto http_obj = create_http();
    // This uses "socket" which was created at the first line inside create_http()
    http_obj->send_request("google.com", "index.html");
    // We better delete our object or else it is memory leak
    //delete http_obj;
    std::cout << "End of main()" << std::endl;}
```

Revision #9

Created 1 year ago by [Preet Kang](#)

Updated 1 year ago by [Preet Kang](#)