32-smart-pointers

June 2, 2024

1 Standard library: smart pointers

1.1 Pointers to nothing

Using `0` or `NULLPTR` does not differentiate a null integer from a null pointer

[1]:
```
#include <iostream>
```

[2]:
```
void process( char * a_value ) { std::cout<<"process(char *): "<<a_value<<std::
↪endl ; }
```

[4]:
```
void process( int a_value )    { std::cout<<"process(int): "<<a_value<<std::
↪endl ; }
```

[6]:
```
void process( long a_value ) { std::cout<<"process(long): "<<a_value<<std::endl␣
↪; }
```

[5]:
```
const int NULLPTR = 0 ;
```

[6]:
```
process(0) ;
```

```
process(int): 0
```

[7]:
```
process(NULLPTR) ;
```

```
process(int): 0
```

[8]:
```
process(NULL) ;    // implementation dependent
```

```
process(long): 0
```

This can lead to unwanted behaviors in case of overloading.

C++ 11 introduces `nullptr`, convertible to any type of pointer

[7]:
```
process(nullptr) ;
```

From now on, any pointer whose value is not yet known should be initiated with `nullptr`.

1.2 `shared_ptr`

- The easiest way to handle objects made with `new`… and the slowest.
- A kind of **guard** which is **counting the references**.

- Provides dereferencing operators * and ->.

A copyable pointer

[5]:
```
%%file tmp.smart.cpp

#include <memory>
#include <vector>
#include <string>
#include <iostream>

void print( std::shared_ptr<std::string> a_text )
 { std::cout<<(*a_text)<<std::endl ; }

int main() {
  std::shared_ptr<std::string> text {new std::string("hello")} ;
  print(text) ;
}
```

```
Overwriting tmp.smart.cpp
```

[6]:
```
!rm -f tmp.smart.exe && g++ -std=c++11 tmp.smart.cpp -o tmp.smart.exe
```

[7]:
```
!./tmp.smart.exe
```

```
hello
```

Shared pointers are practical, but expensive

- They are doubled in size, compared to an ordinary pointer, because they also point to a *control block* which notably contains the current number of references to the pointed object.
- The creation of the first pointer to a given object implies the dynamic creation of the *control block* associated with the pointed object.
- **Increasing or decreasing the number of references should be done in a thread-safe manner**, so it is a bit slowed down.

1.3 `std::unique_ptr`

- The most efficient way to handle objects made with `new`.
- A kind of guard which is **moving the ownership**.
- Provides dereferencing operators * and ->.

Difficulty: it is "move-only"

[8]:
```
%%file tmp.smart.cpp

#include <memory>
#include <vector>
#include <string>
#include <iostream>
```

```
void print_val( std::unique_ptr<std::string> a_text )
 { std::cout<<(*a_text)<<std::endl ; }

int main() {
  std::unique_ptr<std::string> text {new std::string("hello")} ;
  print_val(text) ;
}
```

```
Overwriting tmp.smart.cpp
```

[9]:
```
!rm -f tmp.smart.exe && g++ -std=c++11 tmp.smart.cpp -o tmp.smart.exe
```

```
tmp.smart.cpp: In function ‘int main()’:
tmp.smart.cpp:12:12: error: use of deleted function ‘std::unique_ptr<_Tp,
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&) [with _Tp =
std::__cxx11::basic_string<char>; _Dp =
std::default_delete<std::__cxx11::basic_string<char> >]’
   12 |   print_val(text) ;
      |   ~~~~~~~~~^~~~~~
In file included from /usr/local/include/c++/13.2.0/memory:78,
                 from tmp.smart.cpp:2:
/usr/local/include/c++/13.2.0/bits/unique_ptr.h:522:7: note: declared here
  522 |       unique_ptr(const unique_ptr&) = delete;
      |       ^~~~~~~~~~
tmp.smart.cpp:7:46: note:   initializing argument 1 of ‘void
print_val(std::unique_ptr<std::__cxx11::basic_string<char> >)’
    7 | void print_val( std::unique_ptr<std::string> a_text )
      |                 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~~
```

[10]:
```
%%file tmp.smart.cpp

#include <memory>
#include <vector>
#include <string>
#include <iostream>

void print_ref( std::unique_ptr<std::string> const & a_text )
 { std::cout<<(*a_text)<<std::endl ; }

int main() {
  std::unique_ptr<std::string> text {new std::string("hello")} ;
  print_ref(text) ;
}
```

```
Overwriting tmp.smart.cpp
```

[11]:
```
!rm -f tmp.smart.exe && g++ -std=c++11 tmp.smart.cpp -o tmp.smart.exe
```

[12]:
```
!./tmp.smart.exe
```

```
hello
```

Yet, usable in a collection

[13]:
```
%%file tmp.smart.cpp

#include <memory>
#include <vector>
#include <string>
#include <iostream>

int main(){

  std::vector<std::unique_ptr<std::string>> words ;

  words.push_back(std::unique_ptr<std::string>(new std::string("hello"))) ;
  words.push_back(std::unique_ptr<std::string>(new std::string("world"))) ;
  words.push_back(std::unique_ptr<std::string>(new std::string("!"))) ;

  for ( auto const & word : words )
   { std::cout<<(*word)<<" " ; }
}
```

```
Overwriting tmp.smart.cpp
```

[14]:
```
!rm -f tmp.smart.exe && g++ -std=c++11 tmp.smart.cpp -o tmp.smart.exe
```

[15]:
```
!./tmp.smart.exe
```

```
hello world !
```

The uniques pointers above are made on the fly, i.e. temporary, i.e. right values. Therefore, they can be **moved** into the vector.

In the range-based loop, do not forget the **&**, or the compiler will try to copy the unique pointers when reading them, and fail.

1.4 `make_unique` and `make_shared`

Usual trap: giving a raw pointer to several smart pointers

[16]:
```
%%file tmp.smart.cpp

#include <iostream>
#include <string>
#include <memory>

int main() {
  //...
  int * ip = new int {1} ;
  //...
```

```
  std::shared_ptr<int> sp1 {ip} ;
  //...
  std::shared_ptr<int> sp2 {ip} ;
  //...
}
```

```
Overwriting tmp.smart.cpp
```

[17]:
```
!rm -f tmp.smart.exe && g++ -std=c++11 tmp.smart.cpp -o tmp.smart.exe
```

[18]:
```
!./tmp.smart.exe
```

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

Instead, give the result of `new` directly to the smart pointer

[19]:
```
%%file tmp.smart.cpp

#include <iostream>
#include <string>
#include <memory>

int main() {
  //...
  std::shared_ptr<int> sp1 {new int {1}} ;
  //...
  std::shared_ptr<int> sp2 {sp1} ;
  //...
}
```

```
Overwriting tmp.smart.cpp
```

[20]:
```
!rm -f tmp.smart.exe && g++ -std=c++11 tmp.smart.cpp -o tmp.smart.exe
```

[21]:
```
!./tmp.smart.exe
```

Even better: use `make_shared` and `make_unique`

[26]:
```
%%file tmp.smart.cpp

#include <iostream>
#include <string>
#include <memory>

int main() {
  //...
  auto up1 { std::make_unique<int>(1) } ;
  //...
  auto & up2 { up1 } ;
```

```
  //...
  auto sp1 { std::make_shared<int>(1) } ;
  //...
  auto sp2 { sp1 } ;
  //...
}
```

```
Overwriting tmp.smart.cpp
```

[27]:
```
!rm -f tmp.smart.exe && g++ -std=c++14 tmp.smart.cpp -o tmp.smart.exe
```

[28]:
```
!./tmp.smart.exe
```

1.5 Questions ?

2 Exercise

Eliminate the raw pointers from the example, and use smart pointers instead, so that the explicit call to `delete` in `main()` can be removed.

[1]:
```
%%file tmp.pointers.cpp

#include <iostream>

class MyData
 {
  public :
    MyData( int a_data ) : m_data(a_data)
     { std::cout<<"MyData::MyData("<<m_data<<")"<<std::endl ; }
    int data() const { return m_data ; }
    ~MyData() { std::cout<<"MyData::~MyData("<<m_data<<")"<<std::endl ; }
  private :
    int m_data ;
 } ;

void print( MyData const * a_data_ptr )
 { std::cout<<a_data_ptr->data()<<std::endl ; }

int main()
 {
  MyData * data_ptr = new MyData(42) ;
  print(data_ptr) ;
  delete data_ptr ;
  return 0 ;
 }
```

```
Writing tmp.pointers.cpp
```

[ ]:
```
!rm -f tmp.pointers.exe && g++ -std=c++17 tmp.pointers.cpp -o tmp.pointers.exe
```

[ ]:
```
!./tmp.pointers.exe
```