PKG 09: MODERN C++ SMART POINTERS, "AUTOMATIC" MEMORY MANAGEMENT

Bjarne Stroustrup, Scott Meyers, Stanley Lippman, Josee Lajoe, Barbara Moo, And Stephen Prata --- C++ Primer Plus, Articles, Presentations, IsoCpp, and CppCon

WHY SMART POINTERS? The basics.

Although necessary at time, dynamic memory is notoriously tricky to manage correctly. Programs tend to use dynamic memory for one of 3 purposes:

1. They do not know how many objects they will need.
2. They do not know the precise type of the objects they need.
3. They want to share data between several objects.

One common reason to use dynamic memory is to allow multiple objects to share the same state.

For same reasons as we usually initialize variables, it is also a good idea to initialize dynamically allocated objects.
Dynamic memory managed through built-in pointers (rather than smart pointers) exists until it is explicitly freed/deleted.
Managing Dynamic Memory is Error-Prone. There are three common problems with using new and delete to manage dynamic memory:

1. Forgetting to delete memory: Neglecting to delete dynamic memory is known as a "memory leak," because the memory is never returned to the Free-store. Testing for memory leaks is difficult because they usually cannot be detected until the application is run for a long enough time to actually exhaust memory.
2. Using an object after it has been deleted: This error can sometimes be detected by making the pointer null after the delete.
3. Deleting the same memory twice: This error can happen when two pointers address the same dynamically allocated object. If delete is applied to one of the pointers, then the object's memory is returned to the Free-store. If we subsequently delete the second pointer, then the Free-store may be corrupted.

You can avoid all of the above problems by using smart pointers exclusively. The smart pointers will take care of deleting the memory only when there are no remaining smart pointers pointing to that memory.
It is dangerous to use a built-in pointer to access an object owned by a smart pointer because we may not know when that object is destroyed.
Use get (function) only to pass access to the pointer to code that you know will not delete the pointer. In particular, never use get to initialize or assign (the object) to another smart pointer.

When to use Smart Pointers?

1. Code which involves tracking the ownership of a piece of memory, allocating or de-allocating memory. The smart pointers often save you the need to do things explicitly.
2. Objects that must be allocated with new, but you like to have the same lifetime as other objects/variables on the Run-time stack. Objects assigned to smart pointers will be deleted when program exits that function or block.
3. Data members of classes, so when an object is deleted all the owned data is deleted as well (without any special code in the destructor).

When not to use Smart Pointers (or better to use Raw Pointers)?

1. Memory ownership is obvious such as in a function which gets a pointer from a parameter and do not allocate, deallocate, or store a copy of a pointer which outlasts a function execution.
2. The pointer should not actually own data. For example, when you are just using the data, but you want it to survive the function.
3. The smart pointer is not itself going to be destroyed at some point. You do not want it to sit in the memory that never gets destroyed.
4. If 2 smart pointers point to the same data, then consider unique ptr.

SMART POINTERS: unique_ptr

```
// Name.h
#pragma once
...

class Name {
public:
	Name();
	Name(const string &, const string &);

	~Name();

	string getFirstName() const;
	string getLastName() const;
	string getName() const;
	void setName(const string &, const string &);

private:
	string firstName{ "N/A" };
	string lastName{ "N/A" };
};

// Name.cpp
#include "Name.h"

Name::Name() {}
Name::Name(const string & firstName, const string & lastName)
	: firstName(firstName), lastName(lastName) {
}

Name::~Name() {
	cout << "Name, Destructor: " << this << ", " << this->getName() << endl;
}
string Name::getFirstName() const {
	return this->firstName;
}
string Name::getLastName() const {
	return this->lastName;
}
string Name::getName() const {
	return this->firstName + " " + this->lastName;
}
void Name::setName(const string& firstName, const string& lastName) {
	this->firstName = firstName;
	this->lastName = lastName;
}
```

unique_ptr: Provides unique ownership of an object. During the time that an instance of this class, known as unique pointer, references an object, no other smart pointer can reference the same object. A unique pointer uses a simplified form of reference counting to maintain a reference count of either 0 or 1 for its managed object.

shared_ptr: Provides shared ownership of an object. Several instances of this class can reference the same object. These instances, which we will call shared pointers, use reference counting to detect when an object is no longer reachable by a smart pointer.

weak_ptr: Provides a "weak," or non-owing, reference to an object that is already managed by a shared pointer. Since a weak pointer does not have ownership of the object, it cannot affect the object's lifetime. That is, you cannot use this type of pointer to delete object. You can use it when you want two objects to reference each other. Weak pointers do not use reference counting.

```
SMART POINTERS: unique_ptr
```

```
mickey, uPtr, @:           008FF8B4
mickey, uPtr, value :      00BFA328
mickey, uPtr, owned obj @: 00BFA328

Syntax: Mickey Mouse
Syntax: Mickey Super Mouse
Syntax: Mouse

mickey, uPtr, @:     008FF8B4
mickey, owned obj @: 00000000
goofy,  uPtr @:      008FF8A8
goofy,  owned obj @: 00BFA328
```

```
goofy, owned obj @: 00BFA328
Name, Destructor: 00BFA328, Mickey Mouse
goofy, owned obj @: 00000000

minnie, owned obj @:   00BF9EB0
minnie, owned obj @:   00000000

tempRawPtr, pointee @: 00BF9EB0
daisy, owned obj @:    00BF9EB0

donald, owned obj:    00BFA188 | Donald Duck
sylvester, owned obj: 00BFA1F0 | Sylvester Cat
donald, owned obj:    00BFA1F0 | Sylvester Cat
sylvester, owned obj: 00BFA188 | Donald Duck
```

```
int main() {
	unique_ptr<Name> mickey{ make_unique<Name>("Mickey", "Mouse") };
	cout << "mickey, uPtr, @:           " << &mickey      << endl;
	cout << "mickey, uPtr, value :      " << mickey       << endl;
	cout << "mickey, uPtr, owned obj @: " << mickey.get() << endl;

	// Syntax
	cout << "Syntax: " << mickey->getName()     << endl;
	mickey->setName("Mickey", "Super Mouse");
	cout << "Syntax: " << mickey->getName()     << endl;
	mickey->setName("Mickey", "Mouse");
	cout << "Syntax: " << mickey->getLastName() << endl;

	// move()
	// unique_ptr<Name> goofy = mickey; // Compile Error
	unique_ptr<Name> goofy{ move(mickey) };
	cout << "mickey, uPtr, @:     " << &mickey      << endl;
	cout << "mickey, owned obj @: " << mickey.get() << endl;
	cout << "goofy,  uPtr @:      " << &goofy       << endl;
	cout << "goofy,  owned obj @: " << goofy.get()  << endl;

	// reset()
	cout << "goofy, owned obj @: " << goofy.get() << endl;
	goofy.reset();
	cout << "goofy, owned obj @: " << goofy.get() << endl;

	// reset(), release()
	unique_ptr<Name> minnie{ make_unique<Name>("Minnie", "Mouse") };
	cout << "minnie, owned obj @:   " << minnie.get()  << endl;

	unique_ptr<Name> daisy{ nullptr };
	Name *tmpRawPointer{ minnie.release() };
	daisy.reset(tmpRawPointer);
	cout << "minnie, owned obj @:   " << minnie.get()  << endl;
	cout << "tempRawPtr, pointee @: " << tmpRawPointer << endl;
	cout << "daisy, owned obj @:    " << daisy.get()   << endl;
	tmpRawPointer = nullptr; // * remove backdoor access

	// swap()
	unique_ptr<Name> donald{ make_unique<Name>("Donald", "Duck") };
	unique_ptr<Name> sylvester{ make_unique<Name>("Sylvester", "Cat") };

	cout << "donald, owned obj:    " << donald << " | "    << donald->getName();
	cout << "sylvester, owned obj: " << sylvester << " | " << sylvester->getName();
	swap(donald, sylvester); // or donald.swap(sylvester);
	cout << "donald, owned obj:    " << donald << " | "    << donald->getName();
	cout << "sylvester, owned obj: " << sylvester << " | " << sylvester->getName();

	cout << "\nEND OF PROGRAM" << endl;
	return 0;
}
```

```
END OF PROGRAM
Name, Destructor: 00BFA188, Donald Duck
Name, Destructor: 00BFA1F0, Sylvester Cat
Name, Destructor: 00BF9EB0, Minnie Mouse
```

SMART POINTERS: shared_ptr

```
mickey, sPtr, @:           004FF7B4
mickey, sPtr, value:       00616044
mickey, sPtr, owned obj @: 00616044

.use_count(): 5
.use_count(): 5
.unique():    0
.unique():    0

?before: 0 | 0
?before: 0 | 1

Before swap(): 00616044 | 006104DC
After swap():  006104DC | 00616044
```

```
mickey, owned obj @:  00616044
mickey, owned obj @:  00000000
goofy,  owned obj @:  00616044
goofy,  owned obj @:  00000000
minnie,  owned obj @: 00616044
minnie,  owned obj @: 00000000
donald,  owned obj @: 006104DC
Name, Destructor: 006104DC, Pluto Dog
donald,  owned obj @: 00000000
sylvester,  owned obj @: 00616044
sylvester,  owned obj @: 00000000

END OF PROGRAM
Name, Destructor: 00616044, Mickey Mouse
Name, Destructor: 00615D44, Daisy Duck
```

```
#include "Name.h"

int main() {
	shared_ptr<Name> mickey{ make_shared<Name>("Mickey", "Mouse") };
	cout << "mickey, sPtr, @:           " << &mickey        << endl;
	cout << "mickey, sPtr, value:       " << mickey         << endl;
	cout << "mickey, sPtr, owned obj @: " << mickey.get()   << endl;

	// use_count(), unique()
	shared_ptr<Name> minnie{ mickey }, goofy{ mickey }, donald{ mickey }, sylvester{mickey};
	cout << ".use_count(): " << mickey.use_count()    << endl;
	cout << ".use_count(): " << sylvester.use_count() << endl;
	cout << ".unique():    " << mickey.unique()       << endl;
	cout << ".unique():    " << goofy.unique()        << endl;

	// owner_before()
	cout << "?before: " << mickey.owner_before(donald) << "|" << donald.owner_before(minnie);
	shared_ptr<Name> daisy{ make_shared<Name>("Daisy", "Duck") };
	cout << "?before: " << mickey.owner_before(daisy) << "|" << daisy.owner_before(mickey);

	// swap()
	shared_ptr<Name> pluto{ make_shared<Name>("Pluto", "Dog") };
	cout << "Before swap(): " << donald.get() << " | " << pluto.get() << endl;
	pluto.swap(donald);
	cout << "After swap():  " << donald.get() << " | " << pluto.get() << endl;

	// reset()
	cout << "mickey, owned obj @:  " << mickey.get() << endl;
	mickey.reset();
	cout << "mickey, owned obj @:  " << mickey.get() << endl;
	cout << "goofy,  owned obj @:  " << goofy.get()  << endl;
	goofy.reset();
	cout << "goofy,  owned obj @:  " << goofy.get()  << endl;
	cout << "minnie,  owned obj @: " << minnie.get() << endl;
	minnie.reset();
	cout << "minnie,  owned obj @: " << minnie.get() << endl;
	cout << "donald,  owned obj @: " << donald.get() << endl;
	donald.reset();
	cout << "donald,  owned obj @: "    << donald.get()    << endl;
	cout << "sylvester,  owned obj @: " << sylvester.get() << endl;
	sylvester.reset();
	cout << "sylvester,  owned obj @: " << sylvester.get() << endl;

	cout << "\nEND OF PROGRAM" << endl;
	return 0;
}
```

SMART POINTERS: weak_ptr

```
mickey, sPtr, @:           004FFA40
mickey, sPtr, value:       00878FF4
mickey, sPtr, owned obj @: 00878FF4

weakMickey, wPtr, @:     004FFA30
weakMickey, wPtr, @:     004FFA30
use_count(), # of references: 1
expired(), deleted?           0
Name, Destructor: 00878FF4, mickey mouse
use_count(), # of references: 0
expired(), deleted?           1
```

```
minnie, not empty. Initialized.
# of references/owners: 2
mickey, owned obj @:    008704F4
minnie, owned obj @:    008704F4
.lock() returns:        008704F4

.lock(), observed obj type: class Name
.lock(), observed obj type: class Name

weakMickey, expired(): 0
weakMickey, expired(): 1
mickey, owned obj @:    008704F4
minnie, owned obj @:    008704F4
```

```
int main() {
	shared_ptr<Name> mickey{ make_shared<Name>("mickey", "mouse") };
	cout << "mickey, sPtr, @:           " << &mickey      << endl;
	cout << "mickey, sPtr, value:       " << mickey       << endl;
	cout << "mickey, sPtr, owned obj @: " << mickey.get() << endl;

	weak_ptr<Name> weakMickey{ mickey };
	cout << "weakMickey, wPtr, @:     "    << &weakMickey << endl;
	cout << "weakMickey, wPtr, @:     "    << addressof(weakMickey) << endl;
	// cout << "weakMickey, wPtr, value: " << weakMickey;         // ERROR
	// cout << "weakMickey, wPtr, value: " << weakMickey.get();   // ERROR
	// cout << "weakMickey, wPtr, value: " << *weakMickey;        // ERROR

	// use_count(), expired()
	cout << "use_count(), # of references: " << weakMickey.use_count() << endl;
	cout << "expired(), deleted?           " << weakMickey.expired()   << endl;
	mickey.reset();
	cout << "use_count(), # of references: " << weakMickey.use_count() << endl;
	cout << "expired(), deleted?           " << weakMickey.expired()   << endl;

	// lock()
	mickey = make_shared<Name>("iMickey", "iMouse");
	weakMickey = mickey;

	shared_ptr<Name> minnie{ weakMickey.lock() };
	if (minnie) {
		cout << "minnie, not empty. Initialized." << endl;
	}
	cout << "# of references/owners: " << weakMickey.use_count() << endl;
	cout << "mickey, owned obj @:    " << mickey.get() << endl;
	cout << "minnie, owned obj @:    " << minnie.get() << endl;
	cout << ".lock() returns:       " << weakMickey.lock() << endl << endl;

	// element_type()
	cout << ".lock(), observed obj type: " << typeid(*weakMickey.lock()).name();
	weak_ptr<Name>::element_type goofy{ *weakMickey.lock() };
	cout << ".lock(), observed obj type: " << typeid(goofy).name();

	// reset()
	cout << "weakMickey, expired(): " << weakMickey.expired() << endl;
	weakMickey.reset();
	cout << "weakMickey, expired(): "  << weakMickey.expired() << endl;
	cout << "mickey, owned obj @:    " << mickey.get() << endl;
	cout << "minnie, owned obj @:    " << minnie.get() << endl;

	cout << "\nEND OF PROGRAM" << endl;
	return 0;
}
```

```
END OF PROGRAM
Name, Destructor: 004FF9E0, iMickey iMouse
Name, Destructor: 008704F4, iMickey iMouse
```

```
SMART POINTERS: weak_ptr
```

```
Name, Destructor: 011053C0, Mickey Mouse
mickeyPtr points to: class Name

goofy_wptr is dangling/stale: no
Name, Destructor: 011053C0, Goofy Dog
goofy_wptr is dangling/stale: YES
```

```
// Weak vs. Raw
...

int main(){
      // dangling/stale pointer - cannot check
      Name *mickey = new Name{ "Mickey", "Mouse" };
      Name *mickeyPtr{ mickey };
      delete mickey;
      mickey = nullptr;
      cout << "mickeyPtr points to: " << typeid(*mickeyPtr).name(); // ambiguous
      // cout << "mickeyPtr points to: " << mickeyPtr->getname();   // crash

      // dangling/stale pointer - can check
      shared_ptr<Name> goofy(new Name{ "Goofy", "Dog" });
      weak_ptr<Name> goofy_wptr{ goofy };
      cout << endl;
      cout << "goofy_wptr is dangling/stale: ";
      cout << (goofy_wptr.expired() ? "YES" : "no") << endl;
      goofy.reset();
      cout << "goofy_wptr is dangling/stale: ";
      cout << (goofy_wptr.expired() ? "YES" : "no") << endl;
```

```
Mickey Mouse
Minnie Mouse
Donald Duck
Pluto Dog
Sylvester Cat

42 68 35 1 70 25 79

2 related shared_ptr: equal
```

```
      // more syntax: unique_ptr
      vector<unique_ptr<Name>> names;
      names.push_back(make_unique <Name>("Mickey", "Mouse"));
      names.push_back(make_unique <Name>("Minnie", "Mouse"));
      names.push_back(make_unique <Name>("Donald", "Duck"));
      names.push_back(make_unique <Name>("Pluto", "Dog"));
      names.push_back(make_unique <Name>("Sylvester", "Cat"));

      for (const auto& name : names) {
            cout << name->getName() << endl;
      }
```

```
END OF PROGRAM
Name, Destructor: 01105164, Tango Princess
Name, Destructor: 0110520C, Mulan Princess
Name, Destructor: 01104F90, Cinderella Princess
Name, Destructor: 011053C0, Mickey Mouse
Name, Destructor: 0110A548, Minnie Mouse
Name, Destructor: 01105D90, Donald Duck
Name, Destructor: 011004E8, Pluto Dog
Name, Destructor: 01105E18, Sylvester Cat
```

```
      // more syntax: unique_ptr
      cout << endl;
      auto arr_uptr{ make_unique<int[]>(7) };
      for (int i = 0; i < 7; i++) {
            arr_uptr[i] = rand() % 100 + 1;
            cout << arr_uptr[i] << " ";
      }

      // more syntax: shared_ptr, exception-safe, low construction overhead
      auto mulan{ make_shared<Name>("Mulan", "Princess") };               // preferred
      shared_ptr<Name> cinderella{ new Name{ "Cinderella", "Princess" }}; // 2 steps
      shared_ptr<Name> tangled {nullptr};
      tangled = make_shared<Name>("Tangled", "Princess");
      auto aNewPrincess1 { cinderella };
      auto aNewPrincess2 { cinderella };
      auto anotherNewPrincess{ tangled };
      mulan.swap(cinderella);
      cout << "2 related shared_ptr: " <<
                        ((aNewPrincess1 == aNewPrincess2) ? "equal" : "not equal");

      cout << "\n\nEND OF PROGRAM" << endl;
      return 0;
}
```

SMART PONTERS: HOW TO

SCOTT MEYERS:

- ✓ unique ptr is a small, fast, move-only, smart pointer for managing resources with exclusive-ownership semantics.
- By default, resource destruction takes place via delete, but custom deleters can be specified. Stateful deleters and function pointers as deleters increase the size of unique ptr objects.
- ✓ Converting unique ptr to shared ptr is easy.
- ✓ shared ptr(s) offer convenience approaching that of garbage collection for the shared lifetime management of arbitrary resources.
- ✓ Compared to unique ptr, shared ptr objects are typically twice as big, incur overhead for control blocks, and require atomic reference count manipulations.
- Default resource destruction is via delete, but custom deleters are supported. The type of the delete has no effect on the type of the shared ptr.
- ✓ Avoid creating shared ptr(s) from variables of raw pointer type.
- ✓ Use weak ptr for shared ptr like pointers that can dangle.
- ✓ Potential use case for weak ptr include caching, observer lists, and the prevention of shared ptr cycles.
- ✓ Compared to direct use of new, make functions eliminate source code duplication, improve exception safety, and, for make shared and allocate shared, generate code that's smaller and faster.
- Situations where use of make functions is inappropriate include the need to specify custom deleters and a desire to pass braced initializers.
- ✓ For shared ptr(s), additional situations where make functions may be ill-advised include (1) classes with custom memory management and (2) systems with memory concerns, very large objects, and weak ptr(s) that outlive the corresponding shared ptr(s).

SMART POINTERS Functions from Microsoft Developer Network (MSDN)

<memory> functions: addressof | make_unique | make_shared | …
More: https://docs.microsoft.com/en-us/cpp/standard-library/memory-functions

unique ptr Class, Member functions:

1. get Returns stored_ptr.
2. get_deleter Returns a reference to stored_deleter.
3. release stores pointer() in stored_ptr and returns its previous contents.
4. reset Releases the currently owned resource and accepts a new resource.
5. swap Exchanges resource and deleter with the provided unique_ptr.

shared ptr Class, Member functions:

1. get Gets address of owned resource.
2. owner_before True if this shared_ptr is ordered before (or less than)
3. reset Replace owned resource. | the provided pointer
4. swap Swaps two shared_ptr objects.
5. unique Tests if owned resource is unique.
6. use_count Counts numbers of resource owners.

weak ptr Class, Member functions:

1. element_type The type of the element.
2. expired Tests if ownership has expired.
3. lock Obtains exclusive ownership of a resource.
4. owner_before Returns true if this weak_ptr is ordered before (or less than)
5. reset Releases owned resource. | the provided pointer
6. swap Swaps two weak_ptr objects.
7. use_count Counts number of designated shared_ptr objects.

SMART PONTERS: HOW TO

Smart Pointer Pitfalls:

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they are used properly. To use smart pointers correctly, we must adhere to a set of conventions:

- Do not use the same built-in pointer value to initialize (or reset) more than one smart pointer.
- Do not delete the pointer returned from get().
- Do not use get() to initialize or reset another smart pointer.
- If you use a pointer returned by get(), remember that the pointer will become invalid when the last corresponding smart pointer goes away.
- If you use a smart pointer to manage resource other than memory allocated by new, remember to pass a deleter.

Good practice:

- One manager per each managed object.
- Do not use smart pointers to manage objects on Runtime stack.
- Do not use raw pointers to refer to the same object.

- Use unique ptr by default.
- Use unique ptr if we do not plan to share a resource.
- Use unique ptr first and convert unique ptr to shared ptr when needed.
- Use unique ptr and do not use auto ptr unless we must use old code.

- Use make_unique() and make_share()
- Initialize smart pointers during their declaration.
- Use get() with caution or avoid get().

- Custom deleter for array object managed by a shared_ptr.
- Avoid cyclic references.
- Use release() with caution.
- Use lock() with checking.

Converting unique ptr to shared ptr:

```
addressof(), goofy_unique: 013FF510
Converting...
addressof(), goofy_unique: 00000000
addressof(), goofy_shared: 013FF510
use_count(), goofy_shared: 1

use_count(), Pluto Dog:    1
use_count(), Pluto Dog:    2

END OF PROGRAM
Name, Destructor: 013FF578, Pluto Dog
Name, Destructor: 013FF510, Goofy Dog
```

```
        // Please also see
        // the previous conversions.



        cout << endl;
        unique_ptr<Name> goofy_unique{ make_unique<Name>("Goofy", "Dog") };
        cout << "addressof(), goofy_unique: " << addressof(*goofy_unique) << endl;
        cout << "Converting..." << endl;
        shared_ptr<Name> goofy_shared{ move(goofy_unique) };
        cout << "addressof(), goofy_unique: " << addressof(*goofy_unique) << endl;
        cout << "addressof(), goofy_shared: " << addressof(*goofy_shared) << endl;
        cout << "use_count(), goofy_shared: " << goofy_shared.use_count() << endl;

        cout << endl;
        shared_ptr<Name> pluto_shared{ make_unique<Name>("Pluto", "Dog") };
        cout << "use_count(), Pluto Dog:    " << pluto_shared.use_count() << endl;
        shared_ptr<Name> pluto_shared_2{ pluto_shared };
        cout << "use_count(), Pluto Dog:    " << pluto_shared.use_count() << endl;
```

```
#include <iostream>
#include <string>
using namespace std;

class StuName {
public:
	StuName() {}

	StuName(string name) {
		this->name = make_unique<string>(name);
	}

	~StuName() {
		cout << " " << *this->name << ": Destructor called." << endl;
	}

	const string& getName() const {
		return *this->name;
	}
private:
	unique_ptr<string> name{ nullptr };
};

void passByValue(const unique_ptr<StuName> name) {}

void passByReference(const unique_ptr<StuName>& uPtr_Ref) {

	cout << "BEGIN of passByReference()-"           << endl;
	cout << "&uPtr_Ref:  " << &uPtr_Ref             << endl;
	cout << " uPtr_Ref:  " <<  uPtr_Ref             << endl;
	cout << "&name:      " << &uPtr_Ref->getName()  << endl;
	cout << " name:      " <<  uPtr_Ref->getName()  << endl;
	cout << "END of passByReference()---"   << endl << endl;
}

void passByMove(const unique_ptr<StuName> uPtr_Mov) {

	cout << "BEGIN of passByMove() -----"          << endl;
	cout << "&uPtr_Mov:  " << &uPtr_Mov            << endl;
	cout << " uPtr_Mov:  " <<  uPtr_Mov            << endl;
	cout << "&name:      " << &uPtr_Mov->getName() << endl;
	cout << " name:      " <<  uPtr_Mov->getName() << endl;
	cout << "END of passByMove() -------"  << endl << endl;
}

void passByShare(const shared_ptr<StuName> sPtr_Sha) {
	cout << "BEGIN of passByShare() ----" << endl;
	cout << "use_count(): " << sPtr_Sha.use_count() << endl;
	cout << "&sPtr_Sha: " << &sPtr_Sha            << endl;
	cout << " sPtr_Sha: " <<  sPtr_Sha            << endl;
	cout << "&name:     " << &sPtr_Sha->getName() << endl;
	cout << " name:     " <<  sPtr_Sha->getName() << endl;
	cout << "END of passByShare() ------" << endl << endl;
}
```

```
550  SMART PONTERS: PASSING SMART POINTERS
551
552  int main() {
553        unique_ptr<StuName> name_uPtr{
554                              make_unique<StuName>("Mickey") };
555
556        // Any problem?
557        // passByValue(name_uPtr);
558        // passByValue(make_unique<StuName>("MickeyParameter"));
559
560        //
561        cout << "\nBefore PASS-BY-REFERENCE:"           << endl;
562        cout << "&name_uPtr: " << &name_uPtr            << endl;
563        cout << " name_uPtr: " <<  name_uPtr            << endl;
564        cout << "&name:      " << &name_uPtr->getName() << endl;
565        cout << " name:      " <<  name_uPtr->getName() << endl;
566
567        passByReference(name_uPtr);
568        cout << " name_uPtr: " << name_uPtr << endl;
569
570        //
571        cout << "\nBefore PASS-BY-MOVE:"                << endl;
572        cout << "&name_uPtr: " << &name_uPtr            << endl;
573        cout << " name_uPtr: " <<  name_uPtr            << endl;
574        cout << "&name:      " << &name_uPtr->getName() << endl;
575        cout << " name:      " <<  name_uPtr->getName() << endl;
576
577        passByMove(move(name_uPtr));
578        cout << " name_uPtr: " << name_uPtr << endl;
579
580        //
581        cout << "\nBefore PASS-BY-SHARE:" << endl;
582        name_uPtr = make_unique<StuName>("Minnie");
583        cout << "&name_uPtr: " << &name_uPtr            << endl;
584        cout << " name_uPtr: " << name_uPtr             << endl;
585        cout << "&name:      " << &name_uPtr->getName() << endl;
586        cout << " name:      " <<  name_uPtr->getName() << endl;
587        cout << " Converting..."
588        shared_ptr<StuName> name_sPtr{ name_uPtr.release() };
589
590        cout << "&name_sPtr: " << &name_sPtr            << endl;
591        cout << " name_sPtr: " <<  name_sPtr            << endl;
592        cout << "&name:      " << &name_sPtr->getName() << endl;
593        cout << " name:      " <<  name_sPtr->getName() << endl;
594
595        cout << "\nuse_count(): " << name_sPtr.use_count();
596        passByShare(name_sPtr);
597        cout << "use_count(): "   << name_sPtr.use_count();
598        passByShare(name_sPtr);
599        cout << "use_count(): "   << name_sPtr.use_count();
600
601        cout << "\nEND of Program" << endl;
602        return 0;
603  }
604
605
606
607
608
609
610
611
```

```
Before PASS-BY-REFERENCE:
&name_uPtr: 005CF7BC
 name_uPtr: 00A3DA80
&name:     00A43F78
 name:     Mickey
BEGIN of passByReference()-
&uPtr_Ref:  005CF7BC
 uPtr_Ref:  00A3DA80
&name:     00A43F78
 name:     Mickey
END of passByReference()---

 name_uPtr: 00A3DA80

Before PASS-BY-MOVE:
&name_uPtr: 005CF7BC
 name_uPtr: 00A3DA80
&name:     00A43F78
 name:     Mickey
BEGIN of passByMove() -----
&uPtr_Mov:  005CF694
 uPtr_Mov:  00A3DA80
&name:     00A43F78
 name:     Mickey
END of passByMove() -------

 Mickey: Destructor called.
 name_uPtr: 00000000

Before PASS-BY-SHARE:
&name_uPtr: 005CF7BC
 name_uPtr: 00A3D990
&name:     00A44488
 name:     Minnie
 Converting...
&name_sPtr: 005CF7AC
 name_sPtr: 00A3D990
&name:     00A44488
 name:     Minnie

use_count(): 1
BEGIN of passByShare() ----
use_count(): 2
&sPtr_Sha: 005CF690
 sPtr_Sha: 00A3D990
&name:     00A44488
 name:     Minnie
END of passByShare() ------

use_count(): 1
BEGIN of passByShare() ----
use_count(): 2
&sPtr_Sha: 005CF690
 sPtr_Sha: 00A3D990
&name:     00A44488
 name:     Minnie
END of passByShare() ------

use_count(): 1

END of Program
 Minnie: Destructor called.
```