Important C++ Concepts for Interview

CONTRIBUTOR

SANTOSH KUMAR MISHRA

SDE @ Microsoft

@imsantoshmishra

Preprocessor

#pragma Directive: This directive is a special purpose directive and is used to turn on or off some features. This type of directives is compiler-specific i.e., they vary from compiler to compiler. Some of the #pragma directives are discussed below:

#pragma startup and #pragma exit: These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()) and just before program exit (just before the control returns from main()).

Note: Below program will not work with GCC compilers.
Look at the below program:

```
#include<stdio.h>

void func1();
void func2();

#pragma startup func1
#pragma exit func2

void func1()
{
    printf("Inside func1()\n");
}

void func2()
{
    printf("Inside func2()\n");
}

int main()
{
    printf("Inside main()\n");

    return 0;
}
```

Output:

```
Inside func1()
Inside main()
Inside func2()
```

The above code will produce the output as given below when run on GCC compilers:

```
Inside main()
```

This happens because GCC does not supports #pragma startup or exit. However, you can use the below code for a similar output on GCC compilers.

```
#include<stdio.h>

void func1();
void func2();
```

```
void __attribute__((constructor)) func1();
void __attribute__((destructor)) func2();

void func1()
{
    printf("Inside func1()\n");
}

void func2()
{
    printf("Inside func2()\n");
}

int main()
{
    printf("Inside main()\n");

    return 0;
}
```

#pragma warn Directive: This directive is used to hide the warning message which are displayed during compilation.
We can hide the warnings as shown below:

#pragma warn -rvl: This directive hides those warning which are raised when a function which is supposed to return a value does not returns a value.

#pragma warn -par: This directive hides those warning which are raised when a function does not use the parameters passed to it.

#pragma warn -rch: This directive hides those warning which are raised when a code is unreachable. For example: any code written after the return statement in a function is unreachable.

References vs Pointers

Both references and pointers can be used to change local variables of one function inside another function. Both can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain.
Despite above similarities, there are following differences between references and pointers.

A pointer can be declared as void but a reference can never be void
For example

```
int a = 10;
void* aa = &a;. //it is valid
void &ar = a; // it is not valid
Thanks to Shweta Bansal for adding this point.
```

References are less powerful than pointers

1) Once a reference is created, it cannot be later made to reference another object; it cannot be reseated. This is often done with pointers.
2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.

3) A reference must be initialized when declared. There is no such restriction with pointers

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have above restrictions, and can be used to implement all data structures. References being more powerful in Java, is the main reason Java doesn't need pointers.

References are safer and easier to use:
1) Safer: Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)

2) Easier to use: References don't need dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator (->) is needed to access members.

Together with the above reasons, there are few places like copy constructor argument where pointer cannot be used. Reference must be used pass the argument in copy constructor. Similarly, references must be used for overloading some operators like ++.

Name Mangling
1. Since C++ supports function overloading, additional information has to be added to function names (called name mangling) to avoid conflicts in binary code.
2. Function names may not be changed in C as C doesn't support function overloading. To avoid linking problems, C++ supports extern "C" block. C++ compiler makes sure that names inside extern "C" block are not changed.

Some important differences between the C and C++ structures:

1. Member functions inside structure: Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.

2. Direct Initialization: We cannot directly initialize structure data members in C but we can do it in C++.

3. Using struct keyword: In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record. In C, we must use "struct

Record" for Record variables. In C++, we need not use struct and using 'Record' only would work.

4. Static Members: C structures cannot have static members but is allowed in C++.

5. sizeof operator: This operator will generate 0 for an empty structure in C whereas 1 for an empty structure in C++.
6. Data Hiding: C structures does not allow concept of Data hiding but is permitted in C++ as C++ is an object-oriented language whereas C is not.

7. Access Modifiers: C structures does not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language.

- Conclusion is scope resolution operator is for accessing static or class members and this pointer is for accessing object members when there is a local variable with same name.

Casting operators in C++ | Set 1 (const_cast)

C++ supports following 4 types of casting operators:

1. const_cast
2. static_cast
3. dynamic_cast
4. reinterpret_cast

1. const_cast

const_cast is used to cast away the constness of variables. Following are some interesting facts about const_cast.

1) const_cast can be used to change non-const class members inside a const member function. Consider the following code snippet.
Inside const member function fun(), 'this' is treated by the compiler as 'const student* const this', i.e. 'this' is a constant pointer to a constant object, thus compiler doesn't allow to change the data members through 'this' pointer.
const_cast changes the type of 'this' pointer to 'student* const this'.

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

class student
{
private:
    int roll;
public:
    // constructor
    student(int r):roll(r) {}

    // A const function that changes roll with the help of const_cast
    void fun() const
```

```
    {
        ( const_cast <student*> (this) )->roll = 5;
    }
     int getRoll()  { return roll; }
};

int main(void)
{
    student s(3);
    cout << "Old roll number: " << s.getRoll() << endl;
    s.fun();
    cout << "New roll number: " << s.getRoll() << endl;
     return 0;
}
```

Output:

Old roll number: 3
New roll number: 5

2) const_cast can be used to pass const data to a function that doesn't receive const. For example, in the following program fun() receives a normal pointer, but a pointer to a const can be passed with the help of const_cast.

```
int fun(int* ptr)
{
    return (*ptr + 10);
}

const int val = 10;
const int *ptr = &val;
int *ptr1 = const_cast <int *>(ptr);
cout << fun(ptr1);
```

3) It is undefined behavior to modify a value which is initially declared as const. Consider the following program. The output of the program is undefined. The variable 'val' is a const variable and the call 'fun(ptr1)' tries to modify 'val' using const_cast.

```
int fun(int* ptr)
{
    *ptr = *ptr + 10;
    return (*ptr);
}

int main(void)
{
    const int val = 10;
    const int *ptr = &val;
    int *ptr1 = const_cast <int *>(ptr);
    fun(ptr1);
    cout << val;
    return 0;
```

```
}
```

Output: Undefined Behavior

It it fine to modify a value which is not initially declared as const. For example, in the above program, if we remove const from declaration of val, the program will produce 20 as output.

4) const_cast is considered safer than simple type casting. It'safer in the sense that the casting won't happen if the type of cast is not same as original object. For example, the following program fails in compilation because 'int *' is being typecasted to 'char *'

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

int main(void)
{
    int a1 = 40;
    const int* b1 = &a1;
    char* c1 = const_cast <char *> (b1); // compiler error
    *c1 = 'A';
    return 0;
}
```

output:

prog.cpp: In function 'int main()':
prog.cpp:8: error: invalid const_cast from type 'const int*' to type 'char*'

5) const_cast can also be used to cast away volatile attribute. For example, in the following program, the typeid of b1 is PVKi (pointer to a volatile and constant integer) and typeid of c1 is Pi (Pointer to integer)

```
int main(void)
{
    int a1 = 40;
    const volatile int* b1 = &a1;
    cout << "typeid of b1 " << typeid(b1).name() << '\n';
    int* c1 = const_cast <int *> (b1);
    cout << "typeid of c1 " << typeid(c1).name() << '\n';
    return 0;
}
```

Output:

typeid of b1 PVKi
typeid of c1 Pi

reinterpret_cast

reinterpret_cast casts a pointer to any other type of pointer. It also allows casting from a pointer to an integer type and vice versa.
This operator can cast pointers between non-related classed. The operation results is a simple binary copy of the value from one pointer to the other. The content pointed does not pass any kind of check nor transformation between types.

In the case that the copy is performed from a pointer to an integer, the interpretation of its content is system dependent and therefore any implementation is non portable. A pointer casted to an integer large enough to fully contain it can be casted back to a valid pointer.

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

reinterpret_cast treats all pointers exactly as traditional type-casting operators do.

static_cast

static_cast performs any casting that can be implicitly performed as well as the inverse cast (even if this is not allowed implicitly).
Applied to pointers to classes, that is to say that it allows to cast a pointer of a derived class to its base class (this is a valid conversion that can be implicitly performed) and it can also perform the inverse: cast a base class to its derivate class.

In this last case, the base class that is being casted is not checked to determine wether this is a complete class of the destination type or not.

```
class Base {};
class Derived: public Base {};
Base * a = new Base;
Derived * b = static_cast<Derived*>(a);
```

static_cast, aside from manipulating pointers to classes, can also be used to perform conversions explicitly defined in classes, as well as to perform standard conversions between fundamental types:

```
double d=3.14159265;
int i = static_cast<int>(d);
```

dynamic_cast

dynamic_cast is exclusively used with pointers and references to objects. It allows any type-casting that can be implicitly performed as well as the inverse one when used with polymorphic classes, however, unlike static_cast, dynamic_cast checks, in this last case, if the operation is valid. That is to say, it checks if the casting is going to return a valid complete object of the requested type.
Checking is performed during run-time execution. If the pointer being casted is not a pointer to a valid complete object of the requested type, the value returned is a NULL pointer.

```
class Base {virtual dummy(){}; };
class Derived: public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived* d1 = dynamic_cast<Derived*>(b1);   // succeeds
Derived* d2 = dynamic_cast<Derived*>(b2);   // fails: returns NULL
```

If the type-casting is performed to a reference type and this casting is not possible an exception of type bad_cast is thrown:

```
class Base { virtual dummy(){}; };
class Derived : public Base { };

Base* b1 = new Derived;
Base* b2 = new Base;
Derived d1 = dynamic_cast<Derived&*>(b1);   // succeeds
Derived d2 = dynamic_cast<Derived&*>(b2);   // fails: exception thrown
```

Returning multiple values from a function using Tuple and Pair in C++

There can be some instances where you need to return multiple values (may be of different data types) while solving a problem. One method to do the same is by using pointers, structures or global variables, already discussed here
There is another interesting method to do the same without using the above methods, using tuples (for returning multiple values) and pair (for two values).

We can declare the function with return type as pair or tuple (whichever required) and can pack the values to be returned and return the packed set of values. The returned values can be unpacked in the calling function.

Tuple

A tuple is an object capable to hold a collection of elements where each element can be of a different type.
Class template std::tuple is a fixed-size collection of heterogeneous values

Pair

This class couples together a pair of values, which may be of different types
A pair is a specific case of a std::tuple with two elements
Note : Tuple can also be used to return two values instead of using pair .

```
#include<bits/stdc++.h>
using namespace std;

// A Method that returns multiple values using
// tuple in C++.
tuple<int, int, char> foo(int n1, int n2)
{
    // Packing values to return a tuple
    return make_tuple(n2, n1, 'a');
}

// A Method returns a pair of values using pair
std::pair<int, int> foo1(int num1, int num2)
{
    // Packing two values to return a pair
    return std::make_pair(num2, num1);
}

int main()
{
    int a,b;
    char cc;

    // Unpack the elements returned by foo
    tie(a, b, cc) = foo(5, 10);

    // Storing  returned values in a pair
    pair<int, int> p = foo1(5,2);

    cout << "Values returned by tuple: ";
    cout << a << " " << b << " " << cc << endl;

    cout << "Values returned by Pair: ";
    cout << p.first << " " << p.second;
    return 0;
}
```

Functors in C++

Please note that the title is Functors (Not Functions)!!

A functor is pretty much just a class which defines the operator(). That lets you create objects which "look like" a function.
Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

Functors are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

```
#include <bits/stdc++.h>
using namespace std;

int increment(int x) {  return (x+1); }

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Apply increment to all elements of
    // arr[] and store the modified elements
    // back in arr[]
    transform(arr, arr+n, arr, increment);

    for (int i=0; i<n; i++)
        cout << arr[i] << S" ";

    return 0;
}
```

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what's happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the operator().

The line,
MyFunctor(10);

Is same as
MyFunctor.operator()(10);
Let's delve deeper and understand how this can be used in conjunction with STLs.

```
#include <bits/stdc++.h>
using namespace std;

// A Functor
class increment
{
```

```
private:
    int num;
public:
    increment(int n) : num(n) {  }

    // This operator overloading enables calling
    // operator function () on objects of increment
    int operator () (int arr_num) const {
        return num + arr_num;
    }
};

// Driver code
int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    int to_add = 5;

    transform(arr, arr+n, arr, increment(to_add));

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}
```

Output: 6 7 8 9 10
Thus, here, Increment is a functor, a c++ class that acts as a function.

The line,
transform(arr, arr+n, arr, increment(to_add));

is the same as writing below two lines,
// Creating object of increment
increment obj(to_add);

// Calling () on object
transform(arr, arr+n, arr, obj);
Thus, an object a is created that overloads the operator(). Hence, functors can be used effectively in conjunction with C++ STLs

What is Array Decay?

The loss of type and dimensions of an array is known as decay of an array.This generally occurs when we pass the array into function by value or pointer. What it does is, it sends first address to the array which is a pointer, hence the size of array is not the original one, but the one occupied by the pointer in the memory.

When do we pass arguments by reference or pointer?

In C++, variables are passed by reference due to following reasons:

1) To modify local variables of the caller function: A reference (or pointer) allows called function to modify a local variable of the caller function. For example, consider the following example program where fun() is able to modify local variable x of main().

2) For passing large sized arguments: If an argument is large, passing by reference (or pointer) is more efficient because only an address is really passed, not the entire object. For example, let us consider the following Employee class and a function printEmpDetails() that prints Employee details.

The problem with above code is: every time printEmpDetails() is called, a new Employee abject is constructed that involves creating a copy of all data members. So, a better implementation would be to pass Employee as a reference.
This point is valid only for struct and class variables as we don't get any efficiency advantage for basic types like int, char.. etc.

3) To avoid Object Slicing: If we pass an object of subclass to a function that expects an object of superclass then the passed object is sliced if it is pass by value. For example, consider the following program, it prints "This is Pet Class".

```
class Pet {
public:
    virtual string getDescription() const {
        return "This is Pet class";
    }
};

class Dog : public Pet {
public:
    virtual string getDescription() const {
        return "This is Dog class";
    }
};

void describe(Pet p) { // Slices the derived class object
    cout<<p.getDescription()<<endl;
}

int main() {
    Dog d;
    describe(d);
    return 0;
}
```

If we use pass by reference in the above program then it correctly prints "This is Dog Class". See the following modified program.

```
class Pet {
public:
    virtual string getDescription() const {
        return "This is Pet class";
    }
```

```
};

class Dog : public Pet {
public:
    virtual string getDescription() const {
        return "This is Dog class";
    }
};

void describe(const Pet &p) { // Doesn't slice the derived class object.
    cout<<p.getDescription()<<endl;
}

int main() {
    Dog d;
    describe(d);
    return 0;
}
```

Output:
This is Dog Class

This point is also not valid for basic data types like int, char, .. etc.

4) To achieve Run Time Polymorphism in a function
We can make a function polymorphic by passing objects as reference (or pointer) to it. For example, in the following program, print() receives a reference to the base class object. print() calls the base class function show() if base class object is passed, and derived class function show() if derived class object is passed.

```
class base {
public:
    virtual void show() {  // Note the virtual keyword here
        cout<<"In base \n";
    }
};


class derived: public base {
public:
    void show() {
        cout<<"In derived \n";
    }
};

// Since we pass b as reference, we achieve run time polymorphism here.
void print(base &b) {
    b.show();
}

int main(void) {
    base b;
    derived d;
```

```
    print(b);
    print(d);
    return 0;
}
```

Output:
In base
In derived

Smart Pointer

Consider the following simple C++ code with normal pointers.

```
MyClass *ptr = new MyClass();
ptr->doSomething();
//  We must do delete(ptr) to avoid memory leak
```

Using smart pointers, we can make pointers to work in way that we don't need to explicitly call delete. Smart pointer is a wrapper class over a pointer with operator like * and -> overloaded. The objects of smart pointer class look like pointer, but can do many things that a normal pointer can't like automatic destruction (yes, we don't have to explicitly use delete), reference counting and more.

The idea is to make a class with a pointer, destructor and overloaded operators like * and ->. Since destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically have deleted (or reference count can be decremented). Consider the following simple smartPtr class.

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

class SmartPtr
{
   int *ptr;  // Actual pointer
public:
   // Constructor: Refer http://www.geeksforgeeks.org/g-fact-93/
   // for use of explicit keyword
   explicit SmartPtr(int *p = NULL) { ptr = p; }

   // Destructor
   ~SmartPtr() { delete(ptr); }

   // Overloading dereferencing operator
   int &operator *() {  return *ptr; }
};

int main()
{
    SmartPtr ptr(new int());
    *ptr = 20;
```

```
    cout << *ptr;

    // We don't need to call delete ptr: when the object
    // ptr goes out of scope, destructor for it is automatically
    // called and destructor does delete ptr.

    return 0;
}
```

Output:

20

99

Yes, we can use templates to write a generic smart pointer class. Following C++ code demonstrates the same.

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

// A generic smart pointer class
template <class T>
class SmartPtr
{
   T *ptr;  // Actual pointer
public:
   // Constructor
   explicit SmartPtr(T *p = NULL) { ptr = p; }

   // Destructor
   ~SmartPtr() { delete(ptr); }

   // Overloading dereferncing operator
   T & operator * () {  return *ptr; }

   // Overloding arrow operator so that members of T can be accessed
   // like a pointer (useful if T represents a class or struct or
   // union type)
   T * operator -> () { return ptr; }
};

int main()
{
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;
    return 0;
}
```

Output:20

Smart pointers are also useful in management of resources, such as file handles or network sockets.

C++ libraries provide implementations of smart pointers in the form of auto_ptr, unique_ptr, shared_ptr and weak_ptr

- **What is a smart pointer?**
  It's a type which can be used like a pointer, but provides the additional feature of automatic memory management: When the pointer is no longer in use, the memory it points to is deallocated (see also the more detailed definition on Wikipedia).

- **When should I use one?**
  In code, which involves tracking the ownership of a piece of memory, allocating or de-allocating; the smart pointer often saves you the need to do these things explicitly.

- **But which smart pointer should I use in which of those cases?**
  - Use `std::unique_ptr/std::auto_ptr` when you don't intend to hold multiple references to the same object. For example, use it for a pointer to memory which gets allocated on entering some scope and de-allocated on exiting the scope. The auto_ptr template class describes an object that stores a pointer to a single allocated object that ensures that the object to which it points gets destroyed automatically when control leaves a scope.
  - Use `std::shared_ptr` when you do want to refer to your object from multiple places - and do not want it to be de-allocated until all these references are themselves gone.
  - Use `std::weak_ptr` when you do want to refer to your object from multiple places - for those references for which it's ok to ignore and deallocate (so they'll just note the object is gone when you try to dereference).
  - Don't use the `boost::` smart pointers or `std::auto_ptr` except in special cases which you can read up on if you must.

- **Hey, I didn't ask which one to use!**
  Ah, but you really wanted to, admit it.

- **So when should I use regular pointers then?**
  In code that is oblivious to memory ownership. This would typically be in functions which get a pointer from someplace else and do not allocate, de-allocate or store a copy of the pointer which outlasts their execution.

auto_ptr, unique_ptr, shared_ptr and weak_ptr

C++ libraries provide implementations of smart pointers in following types:

auto_ptr
unique_ptr
shared_ptr
weak_ptr

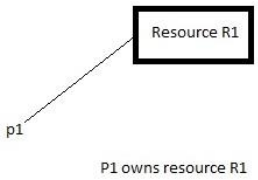
They all are declared in **memory** header file.

auto_ptr

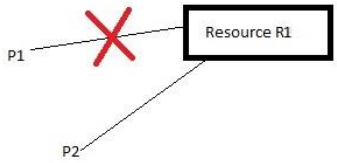
This class template is deprecated as of C++11. unique_ptr is a new facility with a similar functionality, but with improved security.

auto_ptr is a smart pointer that manages an object obtained via new expression and deletes that object when auto_ptr itself is destroyed.

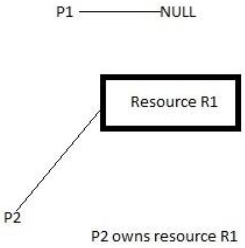
An object when described using auto_ptr class it stores a pointer to a single allocated object which ensures that when it goes out of scope, the object it points to must get automatically destroyed. It is based on exclusive ownership model i.e. two pointers of same type can't point to the same resource at the same time. As shown in below program, copying or assigning of pointers changes the ownership i.e. source pointer has to give ownership to the destination pointer.



P1 owns resource R1



P2 is assigned ownership of R1



P2 owns resource R1

```
// C++ program to illustrate the use of auto_ptr
#include<iostream>
#include<memory>
using namespace std;

class A
{
public:
    void show() {  cout << "A::show()" << endl; }
};

int main()
{
    // p1 is an auto_ptr of type A
    auto_ptr<A> p1(new A);
    p1 -> show();

    // returns the memory address of p1
    cout << p1.get() << endl;

    // copy constructor called, this makes p1 empty.
    auto_ptr <A> p2(p1);
    p2 -> show();

    // p1 is empty now
```

```
    cout << p1.get() << endl;

    // p1 gets copied in p2
    cout<< p2.get() << endl;

    return 0;
}
```

Output:

A::show()
0x1b42c20
A::show()
0
0x1b42c20

The copy constructor and the assignment operator of auto_ptr do not actually copy the stored pointer instead they transfer it, leaving the first auto_ptr object empty. This was one way to implement strict ownership, so that only one auto_ptr object can own the pointer at any given time i.e. auto_ptr should not be used where copy semantics are needed.

Why is auto_ptr deprecated?

It takes ownership of the pointer in a way that no two pointers should contain the same object. Assignment transfers ownership and resets the rvalue auto pointer to a null pointer. Thus, they can't be used within STL containers due to the aforementioned inability to be copied.

unique_ptr

std::unique_ptr was developed in C++11 as a replacement for std::auto_ptr.

unique_ptr is a new facility with a similar functionality, but with improved security (no fake copy assignments), added features (deleters) and support for arrays. It is a container for raw pointers. It explicitly prevents copying of its contained pointer as would happen with normal assignment i.e. it allows exactly one owner of the underlying pointer.

So, when using unique_ptr there can only be at most one unique_ptr at any one resource and when that unique_ptr is destroyed, the resource is automatically claimed. Also, since there can only be one unique_ptr to any resource, so any attempt to make a copy of unique_ptr will cause a compile time error.

```
unique_ptr<A> ptr1 (new A);

// Error: can't copy unique_ptr
```

```
unique_ptr<A> ptr2 = ptr1;
```

But, unique_ptr can be moved using the new move semantics i.e. using std::move() function to transfer ownership of the contained pointer to another unique_ptr.

```
// Works, resource now stored in ptr2

unique_ptr<A> ptr2 = move(ptr1);
```

So, it's best to use unique_ptr when we want a single pointer to an object that will be reclaimed when that single pointer is destroyed.

```
// C++ program to illustrate the use of unique_ptr
#include<iostream>
#include<memory>
using namespace std;

class A
{
public:
    void show()
    {
        cout<<"A::show()"<<endl;
    }
};

int main()
{
    unique_ptr<A> p1 (new A);
    p1 -> show();

    // returns the memory address of p1
    cout << p1.get() << endl;

    // transfers ownership to p2
    unique_ptr<A> p2 = move(p1);
    p2 -> show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;

    // transfers ownership to p3
    unique_ptr<A> p3 = move (p2);
    p3->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;
    cout << p3.get() << endl;

    return 0;
}
```

Output:

A::show()
0x1c4ac20

```
A::show()
0       // NULL
0x1c4ac20
A::show()
0       // NULL
0       // NULL
0x1c4ac20
```

The below code returns a resource and if we don't explicitly capture the return value, the resource will be cleaned up. If we do, then we have exclusive ownership of that resource. In this way, we can think of unique_ptr as safer and better replacement of auto_ptr.

```
unique_ptr<A> fun()
{
   unique_ptr<A> ptr(new A);

   /* ...
      ... */

   return ptr;
}
```

When to use unique_ptr?

Use unique_ptr when you want to have single ownership(Exclusive) of resource. Only one unique_ptr can point to one resource. Since there can be one unique_ptr for single resource it's not possible to copy one unique_ptr to another.

shared_ptr

A shared_ptr is a container for raw pointers. It is a reference counting ownership model i.e. it maintains the reference count of its contained pointer in cooperation with all copies of the shared_ptr. So, the counter is incremented each time a new pointer points to the resource and decremented when destructor of object is called.

Reference Counting: It is a technique of storing the number of references, pointers or handles to a resource such as an object, block of memory, disk space or other resources.

An object referenced by the contained raw pointer will not be destroyed until reference count is greater than zero i.e. until all copies of shared_ptr have been deleted.
So, we should use shared_ptr when we want to assign one raw pointer to multiple owners.

```cpp
// C++ program to demonstrate shared_ptr
#include<iostream>
#include<memory>
using namespace std;

class A
{
public:
    void show()
    {
        cout<<"A::show()"<<endl;
    }
};

int main()
{
    shared_ptr<A> p1 (new A);
    cout << p1.get() << endl;
    p1->show();
    shared_ptr<A> p2 (p1);
    p2->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;

    // Returns the number of shared_ptr objects
    //referring to the same managed object.
    cout << p1.use_count() << endl;
    cout << p2.use_count() << endl;

    // Relinquishes ownership of p1 on the object
    //and pointer becomes NULL
    p1.reset();
    cout << p1.get() << endl;
    cout << p2.use_count() << endl;
    cout << p2.get() << endl;

    return 0;
}
```

Output:

```
0x1c41c20
A::show()
A::show()
0x1c41c20
0x1c41c20
2
2
0      // NULL
1
0x1c41c20
```

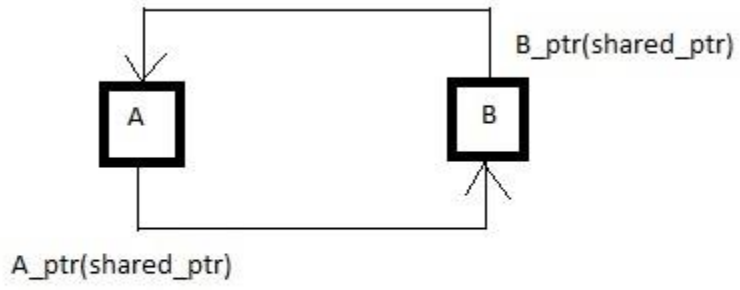
When to use shared_ptr?

Use shared_ptr if you want to share ownership of resource . Many shared_ptr can point to single resource. shared_ptr maintains reference count for this propose. when all shared_ptr's pointing to resource goes out of scope the resource is destroyed.

weak_ptr

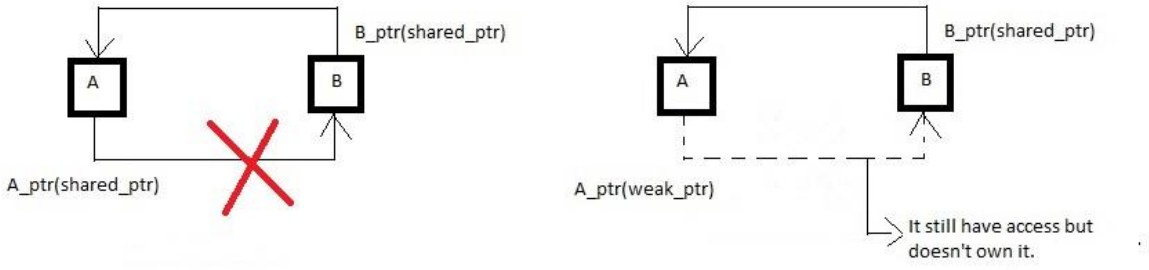
A weak_ptr is created as a copy of shared_ptr. It provides access to an object that is owned by one or more shared_ptr instances, but does not participate in reference counting. The existence or destruction of weak_ptr has no effect on the shared_ptr or its other copies. It is required in some cases to break circular references between shared_ptr instances.

Cyclic Dependency (Problems with shared_ptr): Let's consider a scenario where we have two classes A and B, both have pointers to other classes. So, it's always be like A is pointing to B and B is pointing to A. Hence, use_count will never reach zero and they never get deleted.



Circular Reference

This is the reason we use weak pointers(weak_ptr) as they are not reference counted. So, the class in which weak_ptr is declared doesn't have strong hold of it i.e. the ownership isn't shared, but they can have access to these objects.



So, in case of shared_ptr because of cyclic dependency use_count never reaches zero which is prevented using weak_ptr, which removes this problem by declaring A_ptr as weak_ptr, thus class A does not own it, only have access to it and we also need to check the validity of object as it may go out of scope. In general, it is a design issue.

When to use weak_ptr?

When you do want to refer to your object from multiple places - for those references for which it's ok to ignore and deallocate (so they'll just note the object is gone when you try to dereference).

Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are three different ways where Pointer acts as dangling pointe

1. De-allocation of memory

```
// Deallocating a memory pointed by ptr causes
// dangling pointer
#include <stdlib.h>
#include <stdio.h>
int main()
{
    int *ptr = (int *)malloc(sizeof(int));

    // After below free call, ptr becomes a
    // dangling pointer
    free(ptr);

    // No more a dangling pointer
    ptr = NULL;
}
```

2. Function Call

```
// The pointer pointing to local variable becomes
// dangling when local variable is static.
#include<stdio.h>
```

```
int *fun()
{
    // x is local variable and goes out of
    // scope after an execution of fun() is
    // over.
    int x = 5;

    return &x;
}

// Driver Code
int main()
{
    int *p = fun();
    fflush(stdin);

    // p points to something which is not
    // valid anymore
    printf("%d", *p);
    return 0;
}
```

Output: A garbage Address

The above problem doesn't appear (or p doesn't become dangling) if x is a static variable.

```
// The pointer pointing to local variable doesn't
// become dangling when local variable is static.
#include<stdio.h>

int *fun()
{
    // x now has scope throughout the program
    static int x = 5;

    return &x;
}

int main()
{
    int *p = fun();
    fflush(stdin);

    // Not a dangling pointer as it points
    // to static variable.
    printf("%d",*p);
}
```

Output: 5

```
Variable goes out of scope
void main()
{
  int *ptr;
```

```
  .....
  .....
  {
     int ch;
     ptr = &ch;
  }
  .....
  // Here ptr is dangling pointer
}
```

Void pointer

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically, the type of data that it points to is can be any.

If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.
Important Points

1. void pointers cannot be dereferenced. It can however be done using typecasting the void pointer
2. Pointer arithmetic is not possible on pointers of void due to lack of concrete value and thus size.

Example:

```c
#include<stdlib.h>

int main()
{
    int x = 4;
    float y = 5.5;

    //A void pointer
    void *ptr;
    ptr = &x;

    // (int*)ptr - does type casting of void
    // *((int*)ptr) dereferences the typecasted
    // void pointer variable.
    printf("Integer variable is = %d", *( (int*) ptr) );

    // void pointer is now float
    ptr = &y;
    printf("\nFloat variable is= %f", *( (float*) ptr) );

    return 0;
}
```

Output:

Integer variable is = 4
Float variable is= 5.500000

NULL Pointer

NULL Pointer is a pointer which is pointing to nothing. In case, if we don't have address to be assigned to a pointer, then we can simply use NULL.

```
include <stdio.h>
int main()
{
    // Null Pointer
    int *ptr = NULL;

    printf("The value of ptr is %u", ptr);
    return 0;
}
```

Output :

The value of ptr is 0

Important Points

NULL vs Uninitialized pointer – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

NULL vs Void Pointer – Null pointer is a value, while void pointer is a type

Wild pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

```
int main()
{
    int *p;  /* wild pointer */
     int x = 10;

    // p is not a wild pointer now
    p = &x;

    return 0;
}
```

this' pointer in C++

The 'this' pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. 'this' pointer is a constant pointer that holds the memory address of the current object. 'this' pointer is not available in static member functions as static member functions can be called without any object (with class name).

The type of this depends upon function declaration. If the member function of a class X is declared const, the type of this is const X* if the member function is declared volatile, the type of this is volatile X* and if the member function is declared const volatile, the type of this is const volatile X*
For a class X, the type of this pointer is 'X* const'. Also, if a member function of X is declared as const, then the type of this pointer is 'const X *const'

Following are the situations where 'this' pointer is used:

1) **When local variable's name is same as member's name**

2) **To return reference to the calling object**

```
/* Reference to the calling object can be returned */
Test& Test::func ()
{
   // Some processing
   return *this;
}
```

3) **When a reference to a local object is returned, the returned reference can be used to chain function calls on a single object.**

"delete this" in C++

Ideally delete operator should not be used for this pointer. However, if used, then following points must be considered.

1) delete operator works only for objects allocated using operator new If the object is created using new, then we can do delete this, otherwise behavior is undefined.
2) Once delete this is done, any member of the deleted object should not be accessed after deletion.

The best thing is to not do delete this at all.

Type of iterator :
1) Input iterator
2) Output iterator
3) Forward iterator = forward_list

4) Bi-directional iterator = list
5) Random access iterator = vector

Contributor: Santosh Kumar Mishra, SDE @ Microsoft
Reference: Geeksforgeeks