C is a procedural programming language, but C++ supports both procedural and Object Oriented programming.

C++ supports object oriented programming, it supports features like function overloading, templates, inheritance, virtual functions, and friend functions.

C++ supports exception handling at language level, in C exception handling is done in traditional if-else style.

C++ supports [references](https://www.geeksforgeeks.org/references-in-c/)**,**C doesn’t.

**What are the differences between references and pointers?**  
Both references and pointers can be used to change local variables of one function inside another function. Both of them 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. 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.

**What are virtual functions – Write an example?**  
[Virtual functions](https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/)are used with inheritance, they are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime. Virtual keyword is used to make a function virtual. Necessary to write a C++ program with runtime polymorphism (use of virtual functions)  
1) A base class and a derived class.  
2) A function with same name in base class and derived class.  
3) A pointer or reference of base class type pointing or referring to an object of derived class.

For example, in the following program bp is a pointer of type Base, but a call to bp->show() calls show() function of Derived class, because bp points to an object of Derived class.

|  |
| --- |
| #include<iostream>  using namespace std;    class Base {  public:      virtual void show() { cout<<" In Base \n"; }  };    class Derived: public Base {  public:      void show() { cout<<"In Derived \n"; }  };    int main(void) {      Base \*bp = new Derived;      bp->show();  // RUN-TIME POLYMORPHISM      return 0;  } |

Output:

In Derived

**What is this pointer?**  
The [‘this’ pointer](https://www.geeksforgeeks.org/this-pointer-in-c/)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).

**What are VTABLE and VPTR?**  
vtable is a table of function pointers. It is maintained per class.  
vptr is a pointer to vtable. It is maintained per object.

Compiler adds additional code at two places to maintain and use vtable and vptr.  
1) Code in every constructor. This code sets vptr of the object being created. This code sets vptr to point to vtable of the class.  
2) Code with polymorphic function call (e.g. bp->show() in above code). Wherever a polymorphic call is made, compiler inserts code to first look for vptr using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once vptr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function show() is accessed and called.

**What are main features of OOP?**  
Encapsulation  
Polymorphism  
Inheritance

**What is encapsulation?**  
Encapsulation is referred to one of the following two notions.  
1) Data hiding: A language feature to restrict access to members of an object. For example, private and protected members in C++.  
2) Bundling of data and methods together: Data and methods that operate on that data are bundled together.

**What is Polymorphism? How is it supported by C++?**  
Polymorphism means that some code or operations or objects behave differently in different contexts. In C++, following features support polymorphism.

Compile Time Polymorphism: Compile time polymorphism means compiler knows which function should be called when a polymorphic call is made.  C++ supports compiler time polymorphism by supporting features like templates, function overloading and default arguments.

Run Time Polymorphism: Run time polymorphism is supported by virtual functions. The idea is, [virtual functions](http://en.wikipedia.org/wiki/Virtual_function) are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

**What is**[**Inheritance**](http://en.wikipedia.org/wiki/Inheritance_%28object-oriented_programming%29)**? What is the purpose?**  
The idea of inheritance is simple, a class is based on another class and uses data and implementation of the other class.  
The purpose of inheritance is Code Reuse.

**What is Abstraction?**  
The first thing with which one is confronted when writing programs is the problem. Typically we are confronted with “real-life” problems and we want to make life easier by providing a program for the problem. However, real-life problems are nebulous and the first thing we have to do is to try to understand the problem to separate necessary from unnecessary details: We try to obtain our own abstract view, or model, of the problem. This process of modeling is called abstraction.

img7

In C, static and global variables are initialized by the compiler itself. Therefore, they must be initialized with a constant value.

References in C++

When a variable is declared as reference, it becomes an alternative name for an existing variable. A variable can be declared as reference by putting ‘&’ in the declaration.

#include<iostream>

using namespace std;

int main()

{

  int x = 10;

  // ref is a reference to x.

  int& ref = x;

  // Value of x is now changed to 20

  ref = 20;

  cout << "x = " << x << endl ;

  // Value of x is now changed to 30

  x = 30;

  cout << "ref = " << ref << endl ;

  return 0;

}

Output:

x = 20

ref = 30

Following is one more example that uses references to swap two variables.

|  |
| --- |
| void swap (int& first, int& second)  {      int temp = first;      first = second;      second = temp;  }    int main()  {      int a = 2, b = 3;      swap( a, b );      cout << a << " " << b;      return 0;  } |

Output:

3 2

**References vs Pointers**  
Both references and pointers can be used to change local variables of one function inside another function. Both of them 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

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](https://www.geeksforgeeks.org/archives/4979) 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 ++.

# Can references refer to invalid location in C++?

In C++, [Reference variables](http://en.wikipedia.org/wiki/Reference_%28C%2B%2B%29) are safer than pointers because reference variables must be initialized and they cannot be changed to refer to something else once they are initialized. But there are exceptions where we can have invalid references.

1) Reference to value at uninitialized pointer.

|  |
| --- |
| int \*ptr;  int &ref = \*ptr;  // Reference to value at some random memory location |

2) Reference to a local variable is returned.

|  |
| --- |
| int& fun()  {     int a = 10;     return a;  } |

Once fun() returns, the space allocated to it on stack frame will be taken back. So the reference to a local variable will not be valid.

# Passing by pointer Vs Passing by Reference in C++

In C++, we can pass parameters to a function either by pointers or by reference. In both the cases, we get the same result. So the following questions are inevitable; when is one preferred over the other? What are the reasons we use one over the other?

**Passing by Pointer:**

#include <iostream>

using namespace std;

void swap(int\* x, int\* y)

{

    int z = \*x;

    \*x = \*y;

    \*y = z;

}

int main()

{

    int a = 45, b = 35;

    cout << "Before Swap\n";

    cout << "a = " << a << " b = " << b << "\n";

    swap(&a, &b);

    cout << "After Swap with pass by pointer\n";

    cout << "a = " << a << " b = " << b << "\n";

}

Output:

Before Swap

a = 45 b = 35

After Swap with pass by pointer

a = 35 b = 45

**Passing by Reference:**

#include <iostream>

using namespace std;

void swap(int& x, int& y)

{

    int z = x;

    x = y;

    y = z;

}

int main()

{

    int a = 45, b = 35;

    cout << "Before Swap\n";

    cout << "a = " << a << " b = " << b << "\n";

    swap(a, b);

    cout << "After Swap with pass by reference\n";

    cout << "a = " << a << " b = " << b << "\n";

}

Output:

Before Swap

a = 45 b = 35

After Swap with pass by reference

a = 35 b = 45

**Difference in Reference variable and pointer variable**  
References are generally implemented using pointers. A reference is same object, just with a different name and reference must refer to an object. Since references can’t be NULL, they are safer to use.

1. A pointer can be re-assigned while reference cannot, and must be assigned at initialization only.
2. Pointer can be assigned NULL directly, whereas reference cannot.
3. Pointers can iterate over an array, we can use ++ to go to the next item that a pointer is pointing to.
4. A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
5. A pointer to a class/struct uses ‘->'(arrow operator) to access it’s members whereas a reference uses a ‘.'(dot operator)
6. A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly.

// C++ program to demonstrate differences between pointer

// and reference.

#include <iostream>

using namespace std;

struct demo

{

    int a;

};

int main()

{

    int x = 5;

    int y = 6;

    demo d;

    int \*p;

    p =  &x;

    p = &y;                     // 1. Pointer reintialization allowed

    int &r = x;

    // &r = y;                  // 1. Compile Error

    r = y;                      // 1. x value becomes 6

    p = NULL;

    // &r = NULL;               // 2. Compile Error

    p++;                        // 3. Points to next memory location

    r++;                        // 3. x values becomes 7

    cout << &p << " " << &x << endl;    // 4. Different address

    cout << &r << " " << &x << endl;    // 4. Same address

    demo \*q = &d;

    demo &qq = d;

    q->a = 8;

    // q.a = 8;                 // 5. Compile Error

    qq.a = 8;

    // qq->a = 8;               // 5. Compile Error

    cout << p << endl;        // 6. Prints the address

    cout << r << endl;        // 6. Print the value of x

    return 0;

}

Output (May be different in different runs as we print addresses in program):

0x7ffd09172c20 0x7ffd09172c18

0x7ffd09172c18 0x7ffd09172c18

0x4

7

**Usage in parameter passing:**  
References are usually preferred over pointers whenever we don’t need “reseating”.

Overall, **Use references when you can, and pointers when you have to**. But if we want to write C code that compiles with both C and a C++ compiler, you’ll have to restrict yourself to using pointers.

# 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()*.

|  |
| --- |
| void fun(int &x) {      x = 20;  }    int main() {      int x = 10;      fun(x);      cout<<"New value of x is "<<x;      return 0;  } |

Run on IDE

Output:  
New value of x is 20

**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.

|  |
| --- |
| class Employee {  private:      string name;      string desig;        // More attributes and operations  };    void printEmpDetails(Employee emp) {       cout<<emp.getName();       cout<<emp.getDesig();        // Print more attributes  } |

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.

void printEmpDetails(const Employee &emp) {

     cout<<emp.getName();

     cout<<emp.getDesig();

    // Print more attributes

}

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.

***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](http://en.wikipedia.org/wiki/Object_slicing) if it is pass by value. For example, consider the following program, it prints “This is Pet Class”.

|  |
| --- |
| #include <iostream>  #include<string>    using namespace std;    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;  } |

Output:  
This is Pet Class

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

|  |
| --- |
| #include <iostream>  #include<string>    using namespace std;    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.

***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.

|  |
| --- |
| #include<iostream>  using namespace std;    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

As a side note, it is a recommended practice to make reference arguments const if they are being passed by reference only due to reason no. 2 or 3 mentioned above. This is recommended to avoid unexpected modifications to the objects.

# Dangling, Void , Null and Wild Pointers

**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 pointer

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 non-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

**3.Variable goes out of scope**

void main()

{

int \*ptr;

.....

.....

{

int ch;

ptr = &ch;

}

.....

// Here ptr is dangling pointer

}

[**Void pointer**](http://quiz.geeksforgeeks.org/void-pointer-c/)

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.
3. **Example:**

|  |
| --- |
| #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**](http://quiz.geeksforgeeks.org/few-bytes-on-null-pointer-in-c/)

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**

1. **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.
2. **NULL vs Void Pointer** – Null pointer is a value, while void pointer is a type

[**Wild pointer**](https://www.geeksforgeeks.org/what-are-wild-pointers-how-can-we-avoid/)

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;  } |

Understanding nullptr in C++

Consider the following C++ program that shows problem with NULL (need of nullptr)

|  |
| --- |
| // C++ program to demontrate problem with NULL  #include <bits/stdc++.h>  using namespace std;    // function with integer argument  int fun(int N)   { cout << "fun(int)"; }    // Overloaded function with char pointer argument  int fun(char\* s)  { cout << "fun(char \*)"; }    int main()  {      // Ideally, it should have called fun(char \*),      // but it causes compiler error.      fun(NULL);  } |

Output:

16:13: error: call of overloaded 'fun(NULL)' is ambiguous

fun(NULL);

**What is the problem with above program?**  
NULL is typically defined as (void \*)0 and conversion of NULL to integral types is allowed. So the function call fun(NULL) becomes ambiguous.

|  |
| --- |
| // This program compiles (may produce warning)  #include<stdio.h>  int main()  {     int x = NULL;  } |

**How does nullptr solve the problem?**  
In the above program, if we replace NULL with nullptr, we get the output as “fun(char \*)”.

nullptr is a keyword that can be used at all places where NULL is expected. Like NULL, nullptr is implicitly convertible and comparable to any pointer type. **Unlike NULL, it is not implicitly convertible or comparable to integral types**.

// This program does NOT compile

#include<stdio.h>

int main()

{

   int x = nullptr;

}

Output:

Compiler Error

As a side note, **nullptr is convertible to bool.**

|  |
| --- |
| // This program compiles  #include<iostream>  using namespace std;    int main()  {     int \*ptr = nullptr;       // Below line compiles     if (ptr) { cout << "true"; }     else { cout << "false"; }  } |

Output:

false

There are some unspecified things when we compare two simple pointers but comparison between two values of type nullptr\_t is specified as, comparison by <= and >= return true and comparison by < and > returns false and comparing any pointer type with nullptr by == and != returns true or false if it is null or non-null respectively.

// C++ program to show comparisons with nullptr

#include <bits/stdc++.h>

using namespace std;

// Driver program to test behavior of nullptr

int main()

{

    // creating two variables of nullptr\_t type

    // i.e., with value equal to nullptr

    nullptr\_t np1, np2;

    // <= and >= comparison always return true

    if (np1 >= np2)

        cout << "can compare" << endl;

    else

        cout << "can not compare" << endl;

    // Initialize a pointer with value equal to np1

    char \*x = np1;  // same as x = nullptr (or x = NULL

                    // will also work)

    if (x == nullptr)

        cout << "x is null" << endl;

    else

        cout << "x is not null" << endl;

    return 0;

}

Output :

can compare

x is null

Program to find sum of elements in a given array

// function to return sum of elements in an array of size n

int sum(int arr[], int n)

{

    int sum = 0; // initialize sum

    // Iterate through all elements

    // and add them to sum

    for (int i = 0; i < n; i++)

    sum += arr[i];

    return sum;

}

# Sum of array elements using recursion

// Return sum of elements in A[0..N-1] using recursion.

int findSum(int A[], int N)

{

    if (N <= 0)

        return 0;

    return (findSum(A, N - 1) + A[N - 1]);

}