**References in C++**

# Lvalue Reference variables

A reference variable provides an alias (alternative name) for an existing variable.

Declaration

data-type & reference-name = vaiable-name;

#include <iostream>

int main() {

int x = 10;

int & ref = x;

std::cout << "x: " << x << " ref: " << ref << std::endl;

ref = 20;

std::cout << "x: " << x << " ref: " << ref << std::endl;

x = 30;

std::cout << "x: " << x << " ref: " << ref << std::endl;

return 0;

}

Output

x: 10 ref: 10

x: 20 ref: 20

x: 30 ref: 30

**Applications**

1. **Modify the passed parameters in a function:** If a function receives a reference to a variable, it can modify the value of the variable.
2. **Avoiding copy of large structures:** Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes wastage of CPU time and memory. We can use references to avoid this.
3. **In For Each Loops to modify all objects:** We can use references in for each loops to modify all elements.
4. **In For Each Loops to avoid copy of objects:** We can use references in for each loops to avoid copy of individual objects when objects are large.

#include <iostream>

#include <string>

#include <vector>

struct Student {

std::string name;

std::string address;

int rollNo;

};

void swap (int& first, int& second) {

int temp = first;

first = second;

second = temp;

}

void print\_student(const Student &s) {

std::cout << "Student name: " << s.name << " address: " << s.address << " roll No " << s.rollNo << std::endl;

}

int main() {

// example-01

int a = 2, b = 3;

std::cout << "Before swap a: " << a << " b: " << b << std::endl;

swap(a, b);

std::cout << "After swap a: " << a << " b: " << b << std::endl;

std::cout << std::endl;

// example-02

Student st = {"Student\_01", "Addr of student\_01", 12};

print\_student(st);

std::cout << std::endl;

// example-03

std::vector<int> vect{ 10, 20, 30, 40 };

for (int &x : vect)

x = x + 5;

std::cout << "vector elements: ";

for (int &x : vect)

std::cout << x << " ";

std::cout << std::endl;

std::cout << std::endl;

// example-04

std::vector<std::string> vect\_st{"geeksforgeeks practice",

"geeksforgeeks write",

"geeksforgeeks ide"};

std::cout << "vector elements: " << std::endl;

for (const auto &x : vect\_st)

std::cout << x << std::endl;

return 0;

}

Output

Before swap a: 2 b: 3

After swap a: 3 b: 2

Student name: Student\_01 address: Addr of student\_01 roll No 12

vector elements: 15 25 35 45

vector elements:

geeksforgeeks practice

geeksforgeeks write

geeksforgeeks ide

# Lvalue references to const

By using the const keyword when declaring an lvalue reference, we tell an lvalue reference to treat the object it is referencing as const. Such a reference is called an lvalue reference to a const value (sometimes called a reference to const or a const reference).

int x { 5 }; // x is a modifiable lvalue

const int& ref { x }; // okay: we can bind a const reference to a modifiable lvalue

std::cout << ref; // okay: we can access the object through our const reference

ref = 7; // error: we can not modify an object through a const reference

x = 6; // okay: x is a modifiable lvalue, we can still modify it through the original identifier

**Initializing an lvalue reference to const with an rvalue**

lvalues references to const can also bind to rvalues

const int& ref { 5 }; // okay: 5 is an rvalue

std::cout << ref; // prints 5

When this happens, a temporary object is created and initialized with the rvalue, and the reference to const is bound to that temporary object

*When a const lvalue reference is bound to a temporary object, the lifetime of the temporary object is extended to match the lifetime of the reference*

const int& ref { 5 }; // The temporary object holding value 5 has its lifetime extended to match ref

std::cout << ref; // Therefore, we can safely use it here

# R-value references

C++11 adds a new type of reference called an r-value reference. An r-value reference is a reference that is designed to be initialized with an r-value (only). While an l-value reference is created using a single ampersand, an r-value reference is created using a double ampersand:

int x{ 5 };

int &lref{ x }; // l-value reference initialized with l-value x

int &&rref{ 5 }; // r-value reference initialized with r-value 5

R-values references cannot be initialized with l-values.

|  |  |  |
| --- | --- | --- |
| **R-value reference** | **Can be initialized with** | **Can modify** |
| Modifiable l-values | No | No |
| Non-modifiable l-values | No | No |
| R-values | Yes | Yes |

|  |  |  |
| --- | --- | --- |
| **R-value reference to const** | **Can be initialized with** | **Can modify** |
| Modifiable l-values | No | No |
| Non-modifiable l-values | No | No |
| R-values | Yes | No |

R-value references have two properties that are useful

1. r-value references extend the lifespan of the object they are initialized with to the lifespan of the r-value reference (l-value references to const objects can do this too).
2. non-const r-value references allow you to modify the r-value

int &&rref{ 5 }; // because we're initializing an r-value reference with a literal, a temporary with value 5 is created here

rref = 10;

std::cout << rref << '\n';

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

Reference variables are safer than pointers because reference variables must be initialized and they cannot be changed to refer to something else once they are initialized.

**NOTE**

* Due to above limitations, references in C++ cannot be used for implementing data structure.
* In Java, references do not have above restrictions, and can be used to implement all data structures.
* Main reason Java does not need pointers, because references are more powerful in Java.

**Exceptions**

1. **Reference to value at uninitialized pointer**

int \*ptr;

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

1. **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 be invalid.

**Examples**

|  |  |
| --- | --- |
| #include <iostream>  int & fun() {  static int x = 10;  return x;  }  int main() {  fun() = 30;  std::cout << fun();  return 0;  }  Output  30 | #include <iostream>  int fun(int & x) {  return x;  }  int main() {  int val = 10;  std::cout << fun(val);    //std::cout << fun(10); // CE  // Compilation error: invalid initialization of non-const reference of type 'int&' from an rvalue of type 'int'    return 0;  }  Output  10 |
|  |  |
| #include <iostream>  int & fun() {  int x = 10; // CE  return x;  }  int main() {  fun() = 30;  std::cout << fun();  return 0;  }  Compilation warning: reference to local variable 'x' returned [-Wreturn-local-addr]  Output  Runtime Errors:  Segmentation Fault (SIGSEGV) | #include <iostream>  void swap(char\* & str1, char\* & str2) {  char\* temp = str1;  str1 = str2;  str2 = temp;  }  int main() {  char\* str1 = "Geeks";  char\* str2 = "for Geeks";  swap(str1, str2);  std::cout << "str1: " << str1 << std::endl;  std::cout << "str2: " << str2 << std::endl;  return 0;  }  Output  str1: for Geeks  str2: Geeks |

# References are safer and easier to use

**Safer:** wild references like wild pointers are unlikely to exist (It is still possible to have references that don’t refer to a valid location

**Easier:** can be used like normal variables

‘&’ operator is needed only at the time of declaration

don’t need dereferencing operator to access the value

members of an object reference can be accessed with dot operator (‘.’), unlike pointers where arrow operator (->) is needed

**NOTE**

Reference must be used for overloading some operator like ++

In copy constructor argument pointer cannot be used

In copy constructor passing by reference ensures an actual object is passed, whilst a pointer can have NULL value and make the constructor fail.

Reference is also use for consistency. Operator like =, ==, <=, += would normally be defined using references.

# Pointers vs Reference

|  |  |
| --- | --- |
| **Pointer** | **Reference** |
| A variable that holds memory address of another variable. | An alias (another name) for an already existing variable, implemented by storing the address of an object |
| Needs to be dereferenced with \* operator to access memory location it points to | Automatic indirection, i.e., the compiler will apply the \* operator for you |
| Can be reassigned | Cannot be reassigned |
| Has its own memory address and size on the stack | References shares the same memory address but also takes up some space on the stack |
| Pointer can be assigned new address value | Reference can never refer to an new object until the variable goes out of scope |
| Pointers may be passed to function stored in classes | Reference may be passed to function stored in classes, similar to pointer |
| Pointer can be assigned NULL | Reference cannot be assigned NULL  to ensured that underlying operations do not run into exception situation |
| Pointer have multilevel of indirection  Pointer to pointer to pointer | Reference only offer one level of indirection |
| Arithmetic operations can be performed on pointers | No such thing call reference arithmetic |

* Both can be used to change local variable of one function inside another function
* Efficiency gain: both can be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain
* Pointer can be declared as void

Reference can never be void

* References are less powerful than pointers

References are safer and easier to use

## References are less powerful than pointers

|  |  |
| --- | --- |
| **Reference** | **Pointers** |
| Once a reference is created, it cannot be later made reference another object | This is done with pointers |
| References cannot be null | Pointers are made null to indicate that they are not pointing to any valid thing |
| A reference must be initialized when declared | No such restrictions with pointers |

**NOTE**

* Due to above limitations, references in C++ cannot be used for implementing data structure
* In Java references don’t have above restrictions, and can be used to implement all data structures
* Main reason Java does not need pointers, because references are more powerful in Java

# When to use pointer and reference

Performances are the same

References are implemented internally as pointers

**Reference** In function parameters and return type

**Pointers** To implement data structure

If pointer arithmetic or passing NULL pointer is needed

**NOTE**

* Use references when you can, and pointers when you have to
* References are preferred over pointers, whenever you do not need “reseating”
* References are most useful in a class public interface
* References typically appear on the skin of an object, and pointers on the side

**Exception**

Where a function parameter or return value needs a “sentinel” reference – a reference that does not refer to an object

This is usually best done by returning/taking a pointer and giving the NULL pointer this special significance (references must always alias objects, not a dereferenced NULL pointer)

# pass arguments by reference or pointer?

Variables are passed by reference due to following reasons:

1. **To modify local variables of the caller function**

#include <iostream>

void fun(int &x) {

x = 20;

}

int main() {

int x = 10;

std::cout << "Value of x is " << x << std::endl;

fun(x);

std::cout << "New value of x is " << x << std::endl;

return 0;

}

Output

Value of x is 10

New value of x is 20

1. **For passing large sized arguments**

because only an address is really passed, not the entire object

(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.)

void printEmpDetails(Employee emp) {

// statements

}

void printEmpDetails(const Employee &emp) {

// statements

}

1. **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.

#include <iostream>

#include <string>

class Pet {

public:

virtual std::string getDescription() const {

return "This is Pet class";

}

};

class Dog : public Pet {

public:

virtual std::string getDescription() const {

return "This is Dog class";

}

};

void describe1(Pet p) { // Slices the derived class object

std::cout << p.getDescription() << std::endl;

}

void describe2(const Pet & p) { // Do not slices the derived class object

std::cout << p.getDescription() << std::endl;

}

int main() {

Dog d;

describe1(d);

describe2(d);

return 0;

}

Output

This is Pet class

This is Dog class

1. **To achieve Run Time Polymorphism in a function**

by passing objects as reference (or pointer) to it

#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

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.

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

* References are generally implemented using pointers
* Reference must refer to an object
* References can’t be NULL, they are safer to use
* Pointer can be re-assigned while reference cannot
* Pointer can be assigned NULL directly, whereas reference cannot
* Pointers can iterate over an array
* A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references
* A pointer to a class/struct uses ‘->'(arrow operator) to access it’s members whereas a reference uses a ‘.'(dot operator)
* A pointer needs to be dereferenced with \* to access the memory location it points to, whereas a reference can be used directly
* References are usually preferred over pointers whenever we don’t need “reseating”

#include "stdio.h"

#include <iostream>

using namespace std;

struct demo {

int a;

};

int main() {

int x = 5, y = 6;

demo d;

int \* p;

p = &x; p = &y;

int & r = x;

//&r = y; // error: lvalue required as left operand of assignment

r = y;

p = NULL;

//&r = NULL; // error: lvalue required as left operand of assignment

p++; r++;

cout << &p << " : " << &x << endl; // diff address

cout << &r << " : " << &x << endl; // same address

demo \* q = &d;

demo &qq = d;

q->a = 8;

//q.a = 8;

qq.a = 8;

//qq->a = 8;

cout << p << endl;

cout << r << endl;

return 0;

}

prog.cpp: In function 'int main()':

prog.cpp:17:5: error: lvalue required as left operand of assignment

&r = y;

^

prog.cpp:20:5: error: lvalue required as left operand of assignment

&r = NULL;

^

prog.cpp:30:4: error: request for member 'a' in 'q', which is of pointer type 'demo\*' (maybe you meant to use '->' ?)

q.a = 8;

^

prog.cpp:32:4: error: base operand of '->' has non-pointer type 'demo'

qq->a = 8;

^

Output:

0x7ffdea3e3858 : 0x7ffdea3e3850

0x7ffdea3e3850 : 0x7ffdea3e3850

0x4

7

# References

Pointers and References | <https://www.geeksforgeeks.org/c-plus-plus/>

Chapter 9 Compound Types: References and Pointers | <https://learncpp.com/>

<https://en.cppreference.com/w/cpp/language/pointer>

<https://en.cppreference.com/w/cpp/language/this>

<https://en.cppreference.com/w/cpp/language/reference>

<https://en.cppreference.com/w/cpp/language/reference_initialization>

# END