TATA ELXSI

# Module 5 - Pointer and References

Vikas karanth

20 Sept 2020

# Objective

- Dynamic memory Concepts using new and delete
- Pointers to objects
- memory leaks and  Destructors
- What Is a Reference?
- Creating and using references.
- References to objects
- Null Pointers and Null References
- Passing Function Arguments by Reference
- passing by value vs pass by reference vs Passing by reference using pointers
- Returning Values by Reference
- Passing a const Pointer and const References

# Dynamic memory Concepts using new and delete

## new and delete Operators

- The new operation returns a pointer to the memory allocated.
- The delete operator takes a pointer as its sole argument, to remove the storage (i.e., to return the storage to the heap).
- // Static allocation
- int number = 88;
- int * p1 = &number;  // Assign a "valid" address into pointer

- // Dynamic Allocation
- int * p2=nullptr;  // Not initialize, points to somewhere which is invalid
- p2 = new int;  // Dynamically allocate an int and assign its address to pointer

- *p2 = 99;
- delete p2;  // Remove the dynamically allocated storage
- P2=nullptr;

# Dynamic memory allocation and initialization

- To initialize the allocated memory, you can use an initializer for fundamental types, or invoke a constructor for an object.

- // use an initializer to initialize a fundamental type (such as int, double)
- int * p1 = new int(88);
- double * p2 = new double(1.23);

- // C++11 brace initialization syntax
- int * p1 = new int {88};
- double * p2 = new double {1.23};

- // invoke a constructor to initialize an object (such as Date, Time)
- Date * date1 = new Date(1999, 1, 1);
- Time * time1 = new Time(12, 34, 56);

# new[] and delete[] Operators

- Dynamic array is allocated at runtime by the new[] operator.
- To remove the storage, use the delete[] operator (instead of simply delete).

```
int main() {
  const int SIZE = 5;
  int * pArray = nullptr;

  pArray = new int[SIZE];  // Allocate array via new[] operator
     ……
     ……..
  delete[] pArray;  // Deallocate array via delete[] operator
  return 0;
}
```

## Pointers to classes

- Objects can also be pointed to by pointers: Once declared, a class becomes a valid type, so it can be used as the type pointed to by a pointer.

```
int main() {
  Rectangle obj (3, 4);
  Rectangle * foo, * bar, * baz;
  foo = &obj;
  bar = new Rectangle (5, 6);
  baz = new Rectangle[2] { {2,5}, {3,6} };
  /* Do your stuff here  */
  delete bar;
  delete[] baz;
  return 0;
}
```

# Pointers to objects

- Pointer to object is used to create object dynamically, Can be de-allocated dynamically
- lazy instantiation is a advantage of pointers to objects

```
Vector *v1;
cout<<"\nHow many Vector objects?\n";
cin >> count;
v1 = new Vector[count];

(v1+0)->alloc(5);
(v1+0)->read();   // v1 -> read();
(v1+0)->showsum();
```

# Memory leaks

- **Null pointer**

```
Rectangle obj (3, 4);
Rectangle * foo=nullptr;
Rectangle * bar=nulllptr;
 Rectangle * baz=nullptr;

foo = &obj;
bar = new Rectangle (5, 6);
baz = new Rectangle[2] { {2,5}, {3,6} };
```

- …….
- …….

# Dangling pointers and wild pointers.

- Dangling pointers and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type.

- These are special cases of memory safety violations. [From Wikipedia]

- Dangling pointer
  - Deleting an object from memory explicitly or by destroying the stack frame on return does not alter associated pointers.
  - The pointer still points to the same location in memory even though it may now be used for other purposes.

Example:

```
{
  char *dp = NULL;
  /* ... */
  {
     char c;
     dp = &c;
  }
   /* c falls out of scope */
   /* dp is now a dangling pointer */
}
```

# Dangling pointers and wild pointers.

- Another frequent source of dangling pointers is a jumbled combination of malloc() and free() library calls: a pointer becomes dangling when the block of memory it points to is freed.

```
void func()
{
   char *dp = malloc(A_CONST);
   /* ... */
   free(dp);       /* dp now becomes a dangling pointer */
   dp = NULL;      /* dp is no longer dangling */
   /* ... */
}
```

## Dangling pointers and wild pointers.

- An all too common misstep is returning addresses of a stack-allocated local variable: once a called function returns, the space for these variables gets deallocated and technically they have "garbage values".

```
 int *func(void)
{
   int num = 1234;
   /* ... */
   return &num;
}

int main()
{
   int iPtr=nullptr;
  iPtr=func();
   .....
}
```

## Cause of wild pointers

- Wild pointers are created by omitting necessary initialization prior to first use.
- Thus, strictly speaking, every pointer in programming languages which do not enforce initialization begins as a wild pointer.

- This most often occurs due to jumping over the initialization, not by omitting it. [ Most compilers are able to warn about this.]

```
int f(int i)
{
    char *dp;    /* dp is a wild pointer */
    static char *scp;  /* scp is not a wild pointer:
                * static variables are initialized to 0 at start and retain their values from the last call afterwards.
                * Using this feature may be considered bad style if not commented */
}
```

# Creation and using references.

# References

- A **reference** is a type of C++ variable that acts as an alias to another object or value.

- A reference is essentially an implicit pointer.

- There are three ways that a reference can be used:
  - as a function parameter,
  - as a function return value, or
  - as a stand-alone reference.

## call-by-value

- Probably the most important use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing.
- When using call-by-value, a copy of the argument is passed to the function.
- Call-by-reference passes the address of the argument to the function.
- First, can explicitly pass a pointer to the argument.
- Second, can use a reference parameter. [ For most circumstances the best way is to use a reference parameter]

## call-by-value

```
void fnnegate(int ival);
int main()
{
        int xval;
        xval = 10;
        cout << xval << " negated is ";
        fnnegate(xval);
        cout << xval << "\n";
        return 0;
}
void fnnegate(int ival)
{
        ival = -ival;
}
```

## Call-by-reference (using pointers)

```
void fnnegate(int *ival);
int main()
{
        int xval;
        xval = 10;
        cout << xval << " negated is ";
        fnnegate(&xval);
        cout << xval << "\n";
        return 0;
}
void fnnegate(int *ival)
{
        *ival = -*ival;
}
```

## Call-by-reference  using  reference parameter.

- The main use of references is acting as function formal parameters to support pass-by-reference.
- In an reference variable is passed into a function, the function works on the original copy (instead of a clone copy in pass-by-value).
- Changes inside the function are reflected outside the function to create a reference parameter, precede the parameter's name with an &.

## Call-by-reference using reference parameter : example

```
void fnnegate(int &ival);
int main()
{
        int xval;
        xval = 10;
        cout << xval << " negated is ";
        fnnegate(xval);
        cout << xval << "\n";
        return 0;
}
void fnnegate(int &ival)
{
        ival = -ival;
}
```

## Call-by-reference using reference parameter

- When we create a reference parameter, it automatically refers to (implicitly points to) the argument used to call the function.

- Therefore, in the preceding program, the statement
- ival = -ival ;

- actually operates on xval, not on a copy of xval.

- In general, when you assign a value to a reference, you are actually assigning that value to the variable that the reference points to.

# References are not pointers

| References | Pointers |
|---|---|
| Reference must be initialized when it is created. | Pointers can be initialized any time. |
| Once initialized, we cannot reinitialize a reference. | Pointers can be reinitialized any number of time. |
| You can never have a NULL reference. | Pointers can be NULL. |
| Reference is automatically dereferenced. | * is used to dereference a pointer. |

## Independent References

- By far the most common uses for references are to pass an argument using call-by-reference and to act as a return value from a function.
- However, you can declare a reference that is simply a variable. This type of reference is called an independent reference.
- All independent references must be initialized when they are created.
- Aside from initialization, you cannot change what object a reference variable points to.

# Independent References

```
int main()
{
        int a;
        int &ref = a; // independent reference
        a = 10;
        cout << a << " " << ref << "\n";
        ref = 100;
        cout << a << " " << ref << "\n";
        int b = 19;
        ref = b; // this puts b's value into a
        cout << a << " " << ref << "\n";
        ref--;   // this decrements “a”   it does not affect what ref refers to
        cout << a << " " << ref << "\n";
        return 0;
}
```

## Return by reference

- Return by reference is very different from Call by reference.
- Functions behaves a very important role when variable or pointers are returned as reference.

- A C++ program can be made easier to read and maintain by using references rather than pointers.

- When a function returns a reference, it returns an implicit pointer to its return value.

- This way, a function can be used on the left side of an assignment statement.

- dataType& functionName(parameters);
- where, dataType is the return type of the function, and parameters are the passed arguments to it.

## Return by reference : example

- We should never return a local variable as a reference, reason being, as soon as the functions returns, local variable will be erased, however, we still will be left with a reference which might be a security bug in the code.

```
// C++ program to illustrate return by reference
int x;
int& retByRef()   {
        return x;
}
int main()    {
        retByRef() = 10;
        cout << x;
        return 0;
}
```

## Restrictions to References

- There are a number of restrictions that apply to references.
- Reference another reference is not possible. OR , you cannot obtain the address of a reference.
- Arrays of references can not be created.
- Pointer to a reference is not possible.
- cannot reference a bit-field.
- A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value.
- Null references are not prohibited.

## passingConstReference

```
void function(int , int &, const int &);
main()
{
        int a=5,b = 10, c = 20;
        cout <<"a is "<<a<<"\t b is :"<<b<<"\t c is :" <<c<<endl;
        function(a,b,c);
        cout <<"a is "<<a<<"\t b is :"<<b<<"\t c is :" <<c<<endl;
}
void function(int x, int &ref, const int &constref)
{
        x = ref + constref;
        ref = constref * 10;
        constref = x + ref;
}
```

## Reference to array

```
#include<iostream>
using namespace std;
main()
{
    int   arr[]={1,2,3,4,5};
    int (&ptr)[5] = arr; //reference to an array
    cout << ptr[0]<< ptr[2]<<ptr[4] <<endl;
    cin.get();
}
```

## Object passing by reference

- when an object is passed as an argument to a function, a copy of that object is made.
- When the function terminates, the copy's  destructor is called.
- If for some reason you do not want the destructor function to be called, simply pass the object by reference.
- **When you pass by reference, no copy of the object is made.**
- This means that no object used as a parameter is destroyed when the function terminates, and the parameter's destructor is not called.

# Passing pointers to objects

- This example is using pointer(pass by reference as pointer)

- Person p1,p2;
- p1.init();
- p1.setname("Vikas Karanth");
- p1.setaddress(" TATA ELXSI");
- p1.setphone("91-9164628338");
- printperson(&p1);
- p1.clear();

- Example:
- const_pointer_parameter.cpp

## Why ?

- Passing pointers to objects made const?

- Passing reference to objects made const?

## Passing reference to objects

- **Passing objects by reference is faster than passing them by value.**
- **Arguments are usually passed on the stack.**
- **Thus, large objects take a considerable number of CPU cycles to push onto and pop from the stack.**

- This example is using pointer(pass by C++ reference)

```
Person p1,p2;
p1.init();
p2.init();
p1.setname("Vikas Karanth");
p1.setaddress(" TATA ELXSI");
p1.setphone("91-9164628338");
printperson(p1);
```

- constant_parameter.cpp

# Memory leak

- Where is the bug in this program ?
- What is the solution?

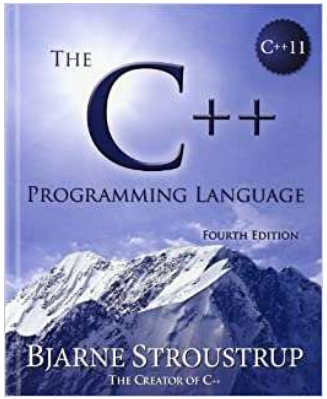
```
CopyConstructor function(CopyConstructor   New)
{
   New.concatenate("constructor");
   return New;
}
/* main function */
int main()
{
    CopyConstructor c1("Copy");
   c1.display();
   function(c1);
   c1.display();
}
```

memory_leak.cpp

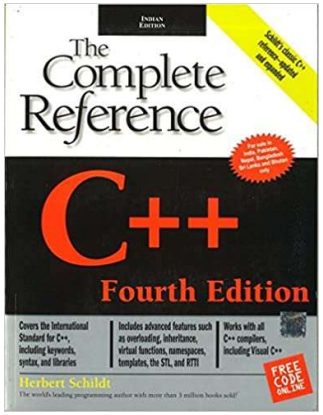
# bibliography and references

- **Online References**
- http://www.cplusplus.com/doc/tutorial/
- https://www3.ntu.edu.sg/home/ehchua/programming/index.html

- **C++ Textbooks**
- Bjarne Stroustrup (Creator of C++), "The C++ Programming Language", 4th ed, 2012.




- The Complete Reference, 4th Edition



**Vikas.karanth@tataelxsi.co.in**

Vikas karanth

L&D Team TATA ELXSI

www.tataelxsi.com