Name: Karthikeyan G

Roll No: CB.SC.P2CYS24008

Secure Coding Lab -9

---

1) Malloc:

```
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
  int *i_ptr = malloc(sizeof(int) * 10); // Dynamic memory allocation
  if (i_ptr != NULL)  // Check if memory was allocated successfully
  {
    for (int i = 0; i < 10; i++)
    {
      i_ptr[i] = i; // Assign values to the array
    }
    for (int i = 0; i < 10; i++)
    {
      printf("Element %d: %d\n", i, i_ptr[i]);  // Print array elements
    }
    free(i_ptr); // Free allocated memory
  }
  else
  {
    printf("Could not allocate memory\n");  // Error message for memory allocation
failure
  }
  return 0; // Return success
}
```

Output:

It allocates dynamic memory of 10*4bytes (40 bytes) in heap memory

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ gcc malloc.c -o malloc

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ ./malloc
Element 0: 0
Element 1: 1
Element 2: 2
Element 3: 3
Element 4: 4
Element 5: 5
Element 6: 6
Element 7: 7
Element 8: 8
Element 9: 9
```

2) Exception Handling:

```
#include <iostream>
#include <new>
int main()
{
   try
   {
      int *pn = new int;         // Allocate memory for a single integer
      int *pi = new int(5);      // Allocate memory and initialize to 5
      double *pd = new double(55.9);  // Allocate memory and initialize to 55.9
      int *buf = new int[10];    // Allocate memory for an array of 10 integers

      *pn = 10;  // Assign value to pn

      std::cout << "Value of pn: " << *pn << std::endl;
      std::cout << "Value of pi: " << *pi << std::endl;
      std::cout << "Value of pd: " << *pd << std::endl;
      // Initialize the array to avoid garbage values
      for (int i = 0; i < 10; i++)
      {
         buf[i] = i;  // Assign values to the array
      }
      // Print array elements
      for (int i = 0; i < 10; i++)
      {
         std::cout << "buf[" << i << "]: " << buf[i] << std::endl;
      }
      // Free allocated memory
      delete pn;
      delete pi;
      delete pd;
      delete[] buf;
   }
   catch (std::bad_alloc& e)
   {
      std::cerr << "Memory allocation failed: " << e.what() << std::endl;
   }

   return 0;
}
```

Output:
Using exceptional handling allocate and free the memory after use

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ exhand.cpp -o exhand

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ ./exhand
Value of pn: 10
Value of pi: 5
Value of pd: 55.9
buf[0]: 0
buf[1]: 1
buf[2]: 2
buf[3]: 3
buf[4]: 4
buf[5]: 5
buf[6]: 6
buf[7]: 7
buf[8]: 8
buf[9]: 9
```

3) Freeing the same memory multiple times:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
  int n = 10;
  // Allocate memory for array x
  int *x = malloc (n * sizeof(int));
  if (x == NULL) // Check for successful allocation
  {
    printf("Memory allocation for x failed\n");
    return 1;
  }
  // Initialize and print values of array x
  for (int i = 0; i < n; i++)
  {
    x[i] = i;
    printf("x[%d] = %d\n", i, x[i]);
  }
  free(x); // Free allocated memory for x
  // Allocate memory for array y
  int *y = malloc (n * sizeof(int));
  if (y == NULL) // Corrected comparison
  {
    printf("Memory allocation for y failed\n");
    return 1;
  }
```

Output:

The segmentation fault happens because the memory for `x` is freed twice; to fix it, replace the second `free(x)` with `free(y)` to properly free the memory allocated for `y`.

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ ./free
x[0] = 0
x[1] = 1
x[2] = 2
x[3] = 3
x[4] = 4
x[5] = 5
x[6] = 6
x[7] = 7
x[8] = 8
x[9] = 9
y[0] = 0
y[1] = 2
y[2] = 4
y[3] = 6
y[4] = 8
y[5] = 10
y[6] = 12
y[7] = 14
y[8] = 16
y[9] = 18
```

4) Improper usage of keywords:

```
#include <iostream>
int main() {
   // Allocate memory for an integer using new
   int *ip = new int(12);
   std::cout << "Value of ip before delete: " << *ip << std::endl;
   // Use delete to free the allocated memory
   delete ip;

   // Allocate memory for an integer using new
   ip = new int(12);
   std::cout << "Value of ip after new allocation: " << *ip << std::endl;

   // Free the memory using delete
   delete ip;
   return 0;
}
```

Output:

In the code, there is improper usage of memory management keywords. In C++, dynamic memory is allocated using new and deallocated using delete, but here the C keyword free(ip) is used, which is incorrect. Mixing new with free leads to undefined behavior since new allocates memory differently than malloc in C, and the memory must be deallocated with delete, not free. To fix this, consistently use new for allocation and delete for deallocation.

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ improp.cpp -o improp

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ ./improp
Value of ip before delete: 12
Value of ip after new allocation: 12
```

5) Smart Pointers in C++:

```
#include <iostream>
#include <memory>
int main() {
   std::shared_ptr<int> p1 = std::make_shared<int>(5);
   std::shared_ptr<int> p2 = p1;
   std::cout << "Value: " << *p1 << std::endl;
   p1.reset();
   if (!p1) {
      std::cout << "p1 is null after reset." << std::endl;
   }
   std::cout << "Value after p1 reset, accessed via p2: " << *p2 << std::endl;
   p2.reset();
   if (!p2) {
      std::cout << "p2 is null after reset." << std::endl;
   }
   return 0;
}
```

OUTPUT:

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ ismart.cpp -o ismart

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ ./ismart
Value: 5
p1 is null after reset.
Value after p1 reset, accessed via p2: 5
p2 is null after reset.
```

Usage of smart pointer are often safer choice than raw pointers, which provides provide augmented behaviour like checking null, garbage collection etc

Valgrind

6) Interpreting Memcheck's output:

```
#include <stdlib.h>
void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0; // problem 1: heap block overrun
} // problem 2: memory leak -- x not freed
int main(void)
{
   f();
   return 0;
}
```

Output:

In this valgrind output, it is showing that the invalid write of size 4 what kind of error. And it shows where the error is present and called by which function and what function cause the problem f() Is displayed.

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./valgr1
==105823== Memcheck, a memory error detector
==105823== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==105823== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==105823== Command: ./valgr1
==105823==
==105823== Invalid write of size 4
==105823==    at 0x109157: f (val1.c:5)
==105823==    by 0x109168: main (val1.c:9)
==105823==  Address 0x4a5b068 is 0 bytes after a block of size 40 alloc'd
==105823==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==105823==    by 0x10914A: f (val1.c:4)
==105823==    by 0x109168: main (val1.c:9)
==105823==
==105823==
==105823== HEAP SUMMARY:
==105823==     in use at exit: 40 bytes in 1 blocks
==105823==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==105823==
==105823== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==105823==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==105823==    by 0x10914A: f (val1.c:4)
==105823==    by 0x109168: main (val1.c:9)
```

The heap memory and leaked memory summary are displayed, but Memcheck cannot provide the root cause of the memory leak.

"Definitely lost": This indicates that your program is indeed leaking memory, and the issue needs to be fixed.

"Probably lost": This suggests that your program is likely leaking memory, unless there is a misconfiguration in pointer handling.

Fixed:

```
#include <stdlib.h>
void f(void)
{
int* x = malloc(10 * sizeof(int));
x [9] = 0; // problem 1: heap block overrun
free(x);
} // problem 2: memory leak -- x not freed
int main(void)
{
f();
return 0;
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ gcc -g val2.c -o valgr2

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./valgr2
==2725== Memcheck, a memory error detector
==2725== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2725== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==2725== Command: ./valgr2
==2725==
==2725==
==2725== HEAP SUMMARY:
==2725==     in use at exit: 0 bytes in 0 blocks
==2725==   total heap usage: 1 allocs, 1 frees, 40 bytes allocated
==2725==
==2725== All heap blocks were freed -- no leaks are possible
==2725==
==2725== For lists of detected and suppressed errors, rerun with: -s
==2725== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Example 2

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
    char *source = "computing"; // Allocates 10 bytes
    char *copy = (char *) malloc(strlen(source));
    strcpy(copy, source); // source is larger than copy
    printf("%s\n", copy);

    // free(copy);

    return 0;
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ gcc valex2.c -o valex2

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./valex2
==4116== Memcheck, a memory error detector
==4116== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4116== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4116== Command: ./valex2
==4116==
==4116== Invalid write of size 1
==4116==    at 0x4846DFE: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==4116==    by 0x1091A6: main (in /home/karthikeyan/Desktop/lab9/valex2)
==4116==  Address 0x4a5b049 is 0 bytes after a block of size 9 alloc'd
==4116==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==4116==    by 0x10918F: main (in /home/karthikeyan/Desktop/lab9/valex2)
==4116==
==4116== Invalid read of size 1
==4116==    at 0x4846CF4: __strlen_sse2 (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==4116==    by 0x48EA653: puts (ioputs.c:35)
==4116==    by 0x1091B2: main (in /home/karthikeyan/Desktop/lab9/valex2)
==4116==  Address 0x4a5b049 is 0 bytes after a block of size 9 alloc'd
==4116==    at 0x4840808: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==4116==    by 0x10918F: main (in /home/karthikeyan/Desktop/lab9/valex2)
==4116==
computing
==4116==
==4116== HEAP SUMMARY:
==4116==     in use at exit: 9 bytes in 1 blocks
==4116==   total heap usage: 2 allocs, 1 frees, 1,033 bytes allocated
==4116==
```

Fixed:

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{

    char *source = "computing"; // Allocates 10 bytes
    char *copy = (char *) malloc(strlen(source)+1);
    strcpy(copy, source); // source is larger than copy
    printf("%s\n", copy);

    free(copy);
    return 0;

}
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ gcc valex2_fix.c -o valex2_fix

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./valex2_fix
==4350== Memcheck, a memory error detector
==4350== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4350== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4350== Command: ./valex2_fix
==4350==
computing
==4350==
==4350== HEAP SUMMARY:
==4350==     in use at exit: 0 bytes in 0 blocks
==4350==   total heap usage: 2 allocs, 2 frees, 1,034 bytes allocated
==4350==
==4350== All heap blocks were freed -- no leaks are possible
==4350==
==4350== For lists of detected and suppressed errors, rerun with: -s
==4350== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

To correct the memory allocation and avoid undefined behavior, ensure that enough memory is allocated for copy to store the entire source string, including the null terminator. This can be done by modifying the allocation line to: char *copy = (char *) malloc(strlen(source) + 1);. Additionally, uncommenting free(copy); will properly release the allocated memory, preventing memory leaks.

7) **Delete function twice error**

```cpp
#include<iostream>
int main()
{
        int *p = new int(10);
        delete p;
        delete p;
        return 0;

}
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ val3.cpp -o val3

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./val3
==4720== Memcheck, a memory error detector
==4720== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4720== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4720== Command: ./val3
==4720==
==4720== Invalid free() / delete / delete[] / realloc()
==4720==    at 0x4843ADF: operator delete(void*, unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64
-linux.so)
==4720==    by 0x109190: main (in /home/karthikeyan/Desktop/lab9/val3)
==4720==  Address 0x4de3080 is 0 bytes inside a block of size 4 free'd
==4720==    at 0x4843ADF: operator delete(void*, unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64
-linux.so)
==4720==    by 0x10917A: main (in /home/karthikeyan/Desktop/lab9/val3)
==4720==  Block was alloc'd at
==4720==    at 0x4840F83: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==4720==    by 0x10915A: main (in /home/karthikeyan/Desktop/lab9/val3)
==4720==
==4720==
==4720== HEAP SUMMARY:
==4720==     in use at exit: 0 bytes in 0 blocks
==4720==   total heap usage: 2 allocs, 3 frees, 73,732 bytes allocated
==4720==
==4720== All heap blocks were freed -- no leaks are possible
==4720==
```

Fixed:

```cpp
#include<iostream>
int main()
{
        int *p = new int(10);
        delete p;
        return 0;
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ val3_fix.cpp -o val3_fix

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./val3_fix
==4769== Memcheck, a memory error detector
==4769== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4769== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4769== Command: ./val3_fix
==4769==
==4769==
==4769== HEAP SUMMARY:
==4769==     in use at exit: 0 bytes in 0 blocks
==4769==   total heap usage: 2 allocs, 2 frees, 73,732 bytes allocated
==4769==
==4769== All heap blocks were freed -- no leaks are possible
==4769==
==4769== For lists of detected and suppressed errors, rerun with: -s
==4769== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

To fix the issue of double deletion, remove the second delete p; statement since it attempts to free the same memory twice, leading to undefined behavior. The corrected code should only call delete once after dynamically allocating memory with new

8) Not freeing a memory allocated dynamically is an error

```
#include <iostream>
#include  <memory>
int main()
{

      int *p = new int[10];
      p[10] = 1;
      return 0;

}
```

To fix the out-of-bounds access issue, change p [10] = 1; to a valid index within the allocated array, such as p[9] = 1;, since the valid indices for an array of size 10 are from 0 to 9

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./val4
==4829== Memcheck, a memory error detector
==4829== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4829== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4829== Command: ./val4
==4829==
==4829== Invalid write of size 4
==4829==    at 0x109157: main (in /home/karthikeyan/Desktop/lab9/val4)
==4829==  Address 0x4de30a8 is 0 bytes after a block of size 40 alloc'd
==4829==    at 0x4842263: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.s
o)
==4829==    by 0x10914A: main (in /home/karthikeyan/Desktop/lab9/val4)
==4829==
==4829==
==4829== HEAP SUMMARY:
==4829==     in use at exit: 40 bytes in 1 blocks
==4829==   total heap usage: 2 allocs, 1 frees, 73,768 bytes allocated
==4829==
==4829== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==4829==    at 0x4842263: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.s
o)
==4829==    by 0x10914A: main (in /home/karthikeyan/Desktop/lab9/val4)
==4829==
==4829== LEAK SUMMARY:
==4829==    definitely lost: 40 bytes in 1 blocks
==4829==    indirectly lost: 0 bytes in 0 blocks
==4829==      possibly lost: 0 bytes in 0 blocks
==4829==    still reachable: 0 bytes in 0 blocks
==4829==         suppressed: 0 bytes in 0 blocks
==4829==
==4829== For lists of detected and suppressed errors, rerun with: -s
==4829== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Fixed:

```cpp
#include <iostream>
#include <memory>

int main()
{
    int *p = new int[10];
    p[9] = 1; // Correctly access the last element of the array
    delete[] p; // Always free dynamically allocated memory return
    0;
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ nano val4_fix.cpp

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ val4_fix.cpp -o val4_fix

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./val4_fix
==4885== Memcheck, a memory error detector
==4885== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4885== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4885== Command: ./val4_fix
==4885==
==4885==
==4885== HEAP SUMMARY:
==4885==     in use at exit: 0 bytes in 0 blocks
==4885==   total heap usage: 2 allocs, 2 frees, 73,768 bytes allocated
==4885==
==4885== All heap blocks were freed -- no leaks are possible
==4885==
==4885== For lists of detected and suppressed errors, rerun with: -s
==4885== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

9) Uninitialized variable

```cpp
#include <iostream>
using namespace std; int
main()

{

        int x;
        cout << x << endl;
        return 0;

}
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ val5.cpp -o val5

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./val5
==4962== Memcheck, a memory error detector
==4962== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4962== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4962== Command: ./val5
==4962==
==4962== Conditional jump or move depends on uninitialised value(s)
==4962==    at 0x49A37FD: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::ostreamb
uf_iterator<char, std::char_traits<char> > >::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<
char> >, std::ios_base&, char, long) const (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
==4962==    by 0x49B288C: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/libstdc+
+.so.6.0.33)
==4962==    by 0x109164: main (in /home/karthikeyan/Desktop/lab9/val5)
==4962==
==4962== Use of uninitialised value of size 8
==4962==    at 0x49A370B: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
==4962==    by 0x49A3827: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::ostreamb
uf_iterator<char, std::char_traits<char> > >::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<
char> >, std::ios_base&, char, long) const (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
==4962==    by 0x49B288C: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/libstdc+
+.so.6.0.33)
==4962==    by 0x109164: main (in /home/karthikeyan/Desktop/lab9/val5)
==4962==
==4962== Conditional jump or move depends on uninitialised value(s)
==4962==    at 0x49A371D: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
==4962==    by 0x49A3827: std::ostreambuf_iterator<char, std::char_traits<char> > std::num_put<char, std::ostreamb
uf_iterator<char, std::char_traits<char> > >::_M_insert_int<long>(std::ostreambuf_iterator<char, std::char_traits<
char> >, std::ios_base&, char, long) const (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.33)
==4962==    by 0x49B288C: std::ostream& std::ostream::_M_insert<long>(long) (in /usr/lib/x86_64-linux-gnu/libstdc+
+.so.6.0.33)
```

Fixed:

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

int main()
{

    int x = 0; // Initialize x with a value

    cout << x << endl;
    return 0;

}
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ val5_fix.cpp -o val5_fix

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./val5_fix
==5025== Memcheck, a memory error detector
==5025== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==5025== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==5025== Command: ./val5_fix
==5025==
0
==5025==
==5025== HEAP SUMMARY:
==5025==     in use at exit: 0 bytes in 0 blocks
==5025==   total heap usage: 2 allocs, 2 frees, 74,752 bytes allocated
==5025==
==5025== All heap blocks were freed -- no leaks are possible
==5025==
==5025== For lists of detected and suppressed errors, rerun with: -s
==5025== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

To fix the issue of using an uninitialized variable, initialize x before using it.

10) Valgrind

- Linege.cpp

```
// Adapted from http://people.cs.ksu.edu/~sherrill/labs/lab05/lineage.cpp
#include "person.h"
#include "personList.h"

int main() {
  PersonList theList;

  theList.addPerson("Bob", "Mark", "Betty");
  theList.addPerson("Jim", "Bob", "Sally");
  theList.addPerson("Frank", "Jim", "Mary");
  theList.addPerson("Leonard", "Jim", "Mary");
  theList.addPerson("Kim", "Leonard", "Sarah");

  theList.printLineage("Jim");
  theList.printLineage("Kim");
  theList.printLineage("Betty");

  return 0;
}
```

```
/* Output (your revised solution should give the same output)
Ancestors of Jim
mother: Sally
father: Bob
grand mother: Betty
grand father: Mark

Decendents of Jim
child: Frank
child: Leonard
grand child: Kim

Ancestors of Kim
mother: Sarah
father: Leonard
grand mother: Mary
grand father: Jim
great grand mother: Sally
great grand father: Bob
great great grand mother: Betty
great great grand father: Mark

Decendents of Kim

Ancestors of Betty

Decendents of Betty
child: Bob
grand child: Jim
great grand child: Frank
great grand child: Leonard
great great grand child: Kim
*/
```

- person.cpp

```
#include "person.h"
#include <iostream>
#include <string.h>
using std::cout;
using std::endl;
Person::Person(char *name, Person* father, Person* mother){
  this->name = new char[strlen(name)+ 1];
  strcpy(this->name, name);
  this->father = father;
  this->mother = mother;
  capacity = 1;
  numChildren = 0;
  children = new Person*[capacity];
}
Person::~Person(){
  delete[] children;
  delete[] name;
}
void Person::addChild(Person *newChild){
  if(numChildren == capacity) expand(&children, &capacity);
  children[numChildren++] = newChild;
}
void Person::printAncestors(){
  cout << endl << "Ancestors of " << name << endl;
  printLineage('u', 0);
}
void Person::printDecendents(){
  cout << endl << "Decendents of " << name << endl;
  printLineage('d', 0);
}
void Person::printLineage(char dir, int level){
  char *temp = compute_relation(level);

  if(dir == 'd'){
    for(int i = 0; i < numChildren; i++){
      cout << temp << "child: " << children[i]->getName() << endl;
      children[i]->printLineage(dir, level + 1);
    }
  } else {
    if(mother){
      cout << temp << "mother: " << mother->getName() << endl;
      mother->printLineage(dir, level + 1);
    }
    if(father){
      cout << temp << "father: " << father->getName() << endl;
      father->printLineage(dir, level + 1);
    }
  }
 delete[] temp;

}
```

```
/* helper function to compute the lineage
* if level = 0 then returns the empty string
* if level >= 1 then returns ("great ")^(level - 1) + "grand "
*/
char* Person::compute_relation(int level){
   if(level == 0) return strcpy(new char[1], "");

   char *temp = strcpy(new char[strlen("grand ") + 1], "grand ");;

   for(int i = 2; i <= level; i++){
      char *temp2 = new char[strlen("great ") + strlen(temp) + 1];
      strcat(strcpy(temp2, "great "), temp);
      delete[] temp;
      temp = temp2;
     // delete temp2;
   }
   return temp;

}

/* non-member function which doubles the size of t
 * NOTE: t's type will be a pointer to an array of pointers
 */
void expand(Person ***t, int *MAX){
  Person **temp = new Person*[2 * *MAX];
  memcpy(temp, *t, *MAX * sizeof(**t));
  *MAX *= 2;

  *t = temp;
  //delete temp;
}
```

- personList.cpp

```
#include "personList.h"
#include <iostream>
#include <string.h>

using std::cout;
using std::endl;

PersonList::PersonList(){
   capacity = 2;
   numPeople = 0;
   theList = new Person*[capacity];
}

PersonList::~PersonList(){
  for(int i = 0; i <  numPeople; i++)
   {
     delete theList[i];
    }
   delete[] theList;
}
```

```
void PersonList::addPerson(char* child_name, char* father_name, char* mother_name){
  Person *father = 0;
  Person *mother = 0;

  // try to find the three names in the theList
  for(int i = 0; i < numPeople; i++){
    if(!strcmp(theList[i]->getName(), child_name)){
      cout << "ERROR: " << child_name << " already has parents!!!";
      return;
    } else if(!strcmp(theList[i]->getName(), father_name)) {
      father = theList[i];
    } else if(!strcmp(theList[i]->getName(), mother_name)) {
      mother = theList[i];
    }
  }

  if(father == 0){
   // father_name is not in the theList so create a new person
   father = new Person(father_name, 0, 0);
   insertIntoList(father);


  }
  if(mother == 0){
   // mother_name is not in the theList so create a new person
   mother = new Person(mother_name, 0, 0);
   insertIntoList(mother);

  }
  Person *newChild = new Person(child_name, father, mother);
  insertIntoList(newChild);
  father->addChild(newChild);
  mother->addChild(newChild);


}

void PersonList::insertIntoList(Person *newPerson){
  if(numPeople == capacity) expand(&theList, &capacity);

  theList[numPeople++] = newPerson;
}

void PersonList::printLineage(char* person){
  for(int i = 0; i < numPeople; i++) {
    if(!strcmp(theList[i]->getName(), person)){
      theList[i]->printAncestors();
      theList[i]->printDecendents();
      return;
    }
  }
  cout << endl << person << " is not in the list!" << endl;
}
```

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ linege.cpp person.cpp personList.cpp -o linege -w

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./linege
==2898== Memcheck, a memory error detector
==2898== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==2898== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==2898== Command: ./linege
==2898==

Ancestors of Jim
mother: Sally
father: Bob
grand mother: Betty
grand father: Mark

Decendents of Jim
child: Frank
child: Leonard
grand child: Kim

Ancestors of Kim
mother: Sarah
father: Leonard
grand mother: Mary
grand father: Jim
great grand mother: Sally
```

```
Ancestors of Betty

Decendents of Betty
child: Bob
grand child: Jim
great grand child: Frank
great grand child: Leonard
great great grand child: Kim
==2898==
==2898== HEAP SUMMARY:
==2898==     in use at exit: 128 bytes in 5 blocks
==2898==   total heap usage: 87 allocs, 82 frees, 76,081 bytes allocated
==2898==
==2898== 8 bytes in 1 blocks are definitely lost in loss record 1 of 5
==2898==    at 0x4842263: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2898==    by 0x1093E2: Person::Person(char*, Person*, Person*) (in /home/karthikeyan/Desktop/lab9/linege)
==2898==    by 0x109BFF: PersonList::addPerson(char*, char*, char*) (in /home/karthikeyan/Desktop/lab9/linege)
==2898==    by 0x109255: main (in /home/karthikeyan/Desktop/lab9/linege)
==2898==
==2898== 8 bytes in 1 blocks are definitely lost in loss record 2 of 5
==2898==    at 0x4842263: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==2898==    by 0x1093E2: Person::Person(char*, Person*, Person*) (in /home/karthikeyan/Desktop/lab9/linege)
==2898==    by 0x109BBE: PersonList::addPerson(char*, char*, char*) (in /home/karthikeyan/Desktop/lab9/linege)
==2898==    by 0x109279: main (in /home/karthikeyan/Desktop/lab9/linege)
==2898==
==2898== 16 bytes in 1 blocks are definitely lost in loss record 3 of 5
==2898==    at 0x4842263: operator new[](unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
```

Fixed:

linege.cpp

```
#include "person.h"
#include "personList.h"
int main() {
   // Create an instance of PersonList
   PersonList theList;
   // Add persons to the list with their parents
   theList.addPerson("Bob", "Mark", "Betty");
   theList.addPerson("Jim", "Bob", "Sally");
   theList.addPerson("Frank", "Jim", "Mary");
   theList.addPerson("Leonard", "Jim", "Mary");
   theList.addPerson("Kim", "Leonard", "Sarah");
   // Print the lineage of specific individuals
   theList.printLineage("Jim");
   theList.printLineage("Kim");
   theList.printLineage("Betty");

   return 0;
}
```

person.cpp

```
#include "person.h"
#include <iostream>
#include <cstring> // Prefer <cstring> over <string.h> in C++
using std::cout;
using std::endl;
// Constructor
Person::Person(char *name, Person* father, Person* mother) {
  this->name = new char[strlen(name) + 1];
  strcpy(this->name, name);
  this->father = father;
  this->mother = mother;
  capacity = 1;
  numChildren = 0;
  children = new Person*[capacity];
}
// Helper function to compute the relationship lineage
// If level = 0, it returns an empty string.
// If level >= 1, it returns ("great ")^(level - 1) + "grand ".
char* Person::compute_relation(int level) {
  if (level == 0) {
    return strcpy(new char[1], ""); // Return an empty string
  }
  char *temp = strcpy(new char[strlen("grand ") + 1], "grand ");
  for (int i = 2; i <= level; ++i) {
    char *temp2 = new char[strlen("great ") + strlen(temp) + 1];
    strcat(strcpy(temp2, "great "), temp);
    delete[] temp;
    temp = temp2;
  }
  return temp;
}
// Expands the size of an array of pointers to accommodate more elements.
// t is a pointer to an array of pointers, and MAX is the current capacity.
void expand(Person ***t, int *MAX) {
  Person **temp = new Person*[2 * *MAX];
  memcpy(temp, *t, *MAX * sizeof(**t)); // Copy existing elements to the new array
  delete[] *t; // Free the old array
  *MAX *= 2; // Update capacity
  *t = temp; // Assign new array to original pointer
}
// Destructor to free dynamically allocated memory
Person::~Person() {
  delete[] children;
  delete[] name;
}
```

```
// Adds a child to the current person. If capacity is reached, the array is expanded.
void Person::addChild(Person *newChild) {
   if (numChildren == capacity) {
      expand(&children, &capacity);
   }
   children[numChildren++] = newChild;
}

// Prints the ancestors of the current person.
void Person::printAncestors() {
   cout << endl << "Ancestors of " << name << endl;
   printLineage('u', 0); // 'u' represents upward lineage (ancestors)
}

// Prints the descendants of the current person.
void Person::printDescendants() {
   cout << endl << "Descendants of " << name << endl;
   printLineage('d', 0); // 'd' represents downward lineage (descendants)
}

// Recursively prints the lineage of the person.
// dir = 'd' for descendants, 'u' for ancestors.
// level is used to compute the relationship (grand/great-grand).
void Person::printLineage(char dir, int level) {
   char *relation = compute_relation(level);

   if (dir == 'd') { // Printing descendants
      for (int i = 0; i < numChildren; ++i) {
         cout << relation << "child: " << children[i]->getName() << endl;
         children[i]->printLineage(dir, level + 1); // Recursively print descendants
      }
   } else { // Printing ancestors
      if (mother) {
         cout << relation << "mother: " << mother->getName() << endl;
         mother->printLineage(dir, level + 1); // Recursively print mother's lineage
      }
      if (father) {
         cout << relation << "father: " << father->getName() << endl;
         father->printLineage(dir, level + 1); // Recursively print father's lineage
      }
   }

   delete[] relation; // Clean up dynamically allocated memory
}
```

PersonList.cpp:

```
#include "personList.h"
#include <iostream>
#include <cstring> // Prefer <cstring> over <string.h> in C++
using std::cout;
using std::endl;
// Constructor
PersonList::PersonList() {
  capacity = 2;
  numPeople = 0;
  theList = new Person*[capacity];
}
// Destructor
PersonList::~PersonList() {
  // Free memory for all people in the list
  for (int i = 0; i < numPeople; ++i) {
    delete theList[i];
  }
  delete[] theList; // Free the array itself
}
// Adds a person to the list along with their father and mother
void PersonList::addPerson(char* child_name, char* father_name, char* mother_name) {
  Person *father = nullptr;
  Person *mother = nullptr;
  // Try to find the child, father, and mother in the list
  for (int i = 0; i < numPeople; ++i) {
    if (!strcmp(theList[i]->getName(), child_name)) {
      cout << "ERROR: " << child_name << " already has parents!!!" << endl;
      return; // Child already exists, exit early
    } else if (!strcmp(theList[i]->getName(), father_name)) {
      father = theList[i]; // Father found in the list
    } else if (!strcmp(theList[i]->getName(), mother_name)) {
      mother = theList[i]; // Mother found in the list
    }
  }
  // If father was not found, create a new Person for the father
  if (father == nullptr) {
    father = new Person(father_name, nullptr, nullptr);
    insertIntoList(father); // Insert new father into the list
  }
  // If mother was not found, create a new Person for the mother
  if (mother == nullptr) {
    mother = new Person(mother_name, nullptr, nullptr);
    insertIntoList(mother); // Insert new mother into the list
  }
  // Create a new Person for the child
  Person *newChild = new Person(child_name, father, mother);
  insertIntoList(newChild); // Insert the new child into the list
  // Add the child to both parents
  father->addChild(newChild);
  mother->addChild(newChild);
}
```

```
// Inserts a person into the list, expanding the list if needed
void PersonList::insertIntoList(Person *newPerson) {
   // If the list is full, expand its capacity
   if (numPeople == capacity) {
      expand(&theList, &capacity);
   }

   // Add the new person to the list
   theList[numPeople++] = newPerson;
}

// Prints the lineage (ancestors and descendants) of a given person
void PersonList::printLineage(char* person_name) {
   // Find the person in the list
   for (int i = 0; i < numPeople; ++i) {
      if (!strcmp(theList[i]->getName(), person_name)) {
         // Print ancestors and descendants
         theList[i]->printAncestors();
         theList[i]->printDescendants();
         return; // Exit after printing
      }
   }

   // If the person was not found in the list
   cout << endl << person_name << " is not in the list!" << endl;
}
```

Output:

```
┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ g++ linege.cpp person.cpp personList.cpp -o linege -w

┌──(karthikeyan㉿kali)-[~/Desktop/lab9]
└─$ valgrind --leak-check=full ./linege
==4554== Memcheck, a memory error detector
==4554== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==4554== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==4554== Command: ./linege
==4554==

Ancestors of Jim
mother: Sally
father: Bob
grand mother: Betty
grand father: Mark

Decendents of Jim
child: Frank
child: Leonard
grand child: Kim

Ancestors of Kim
mother: Sarah
father: Leonard
grand mother: Mary
grand father: Jim
great grand mother: Sally
great grand father: Bob
great great grand mother: Betty
great great grand father: Mark

Decendents of Kim

Ancestors of Betty

Decendents of Betty
child: Bob
grand child: Jim
great grand child: Frank
great grand child: Leonard
great great grand child: Kim
```

```
==4554==
==4554== HEAP SUMMARY:
==4554==     in use at exit: 0 bytes in 0 blocks
==4554==   total heap usage: 87 allocs, 87 frees, 76,081 bytes allocated
==4554==
==4554== All heap blocks were freed -- no leaks are possible
==4554==
==4554== For lists of detected and suppressed errors, rerun with: -s
==4554== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Resolved memory leaks by ensuring that all dynamically allocated memory is correctly released in the destructors