C++: Pointer & Dynamic Memory

C++: Pointers

- **Pointer**
  - Variable whose value is the **address** of another variable

    type *ptrName

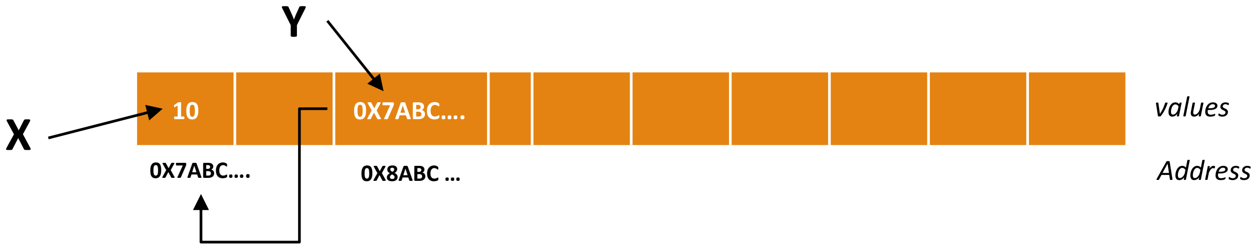
    int *intPtr , double *doublePtr, char *charPtr
  - How to use them

    int value= 20;

    int *valuePtr = &value (read as address of value variable)

C++: Pointers

Example



```
#include <iostream>

using namespace std;

int main() {

          int x = 10;
          int* y = &x;
          cout << “Address of X is: " << y << endl;
          cout << "Value that address is:“<< *y << endl;
}
```

Dereferencing the pointer

Deference pointer Y ➔ Y holds a value, which is an address location
Get me the value held at the address **0X7ABC....**

Y is also a variable – has its own address

C++: Pointers

Refer to: Pointer.cpp

- **Pointer**
  - How to access the value

    ```
    int ptrValue = *valuePtr (value pointed by valuePtr pointer)
    ```

  - Pointer is similar to Array

    ```
    int array[5] = {1,2,3,4,5}
    int *ptr;
    ptr = array;
    ```

C++: Pointer

- **Pointer**
  - Arithmetic
    - *p++
      - same as *(p++): increment pointer, and dereference un-incremented address
    - *++p
      - same as *(++p): increment pointer, and dereference incremented address
    - ++*p
      - same as ++(*p): dereference pointer, and increment the value it points to
    - (*p)++
      - dereference pointer, and post-increment the value it points to

C++: Pointers

Refer to: Pointer.cpp

Pointer

- Constant Pointer
  - They can access/read the value, but they cannot change the value

```
 int x = 10;
const int *valuePtr = &x
int temp = *valuePtr
*valuePtr = 30; X Not allowed, as *valuePtr is constant
```

C++: Pointers

Refer to: Pointer.cpp

Pointer

- Constant Pointer
  - They can access/read/change the value, but they cannot change reference

```
 int x = 10;
int* const valuePtr = &x
int temp = 20;
valuePtr = &temp; X Not allowed, as *valuePtr is constant to a reference
```

C++: Pointers

Refer to: Pointer.cpp

— Pointer

» Constant Pointer

› They cannot change the value and reference

```
 int value = 10;
const int* const valuePtr = &value
 *valuePtr = 30; X Not allowed
int temp = 20;
valuePtr = &temp; X Not allowed, as *valuePtr is constant to reference
```

C++: Pointers vs Reference

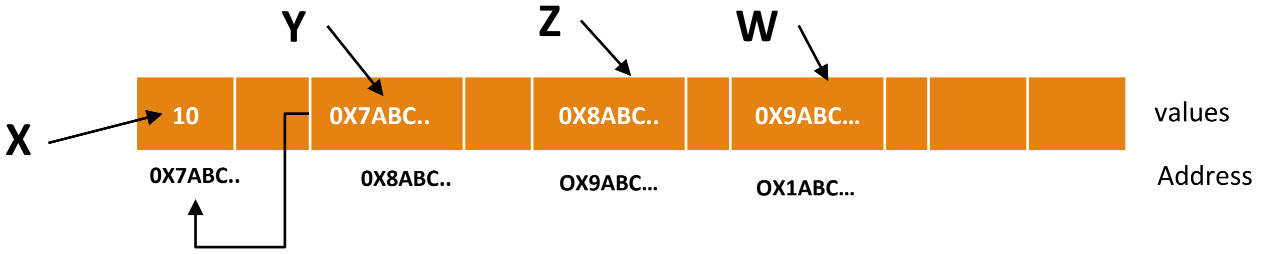
- **What is the difference ?**
  - Pointer can be re-assigned
  - Pointer can be initialized without pointing to any variable
  - Pointer has its own memory address
  - Pointer can be NULL → called null pointer
  - Pointer can point to another pointer
  - Pointer arithmetic is valid

C++: Pointers

Pointers

…… .

```
int main() {
          int x = 10;
          int* y = &x;
          cout << "Value that address is:"<< *y << endl;

          int** z = &y;
          int*** w = &z;
          cout << "Value of Z is :"<< z << endl;
          cout << "Dereferencing Z: "<< *z << endl;
          cout << "Dereferencing Z twice: "<< **z << endl;

          cout << "Value of W is :"<< w << endl;
          cout << "Dereferencing W: "<< *w << endl;
          cout << "Dereferencing W twice: "<< **w << endl;
          cout << "Dereferencing W thrice: "<< ***w << endl;

}
```



Dereferencing the pointer

Deference pointer Y ➔ Y holds a value, which is an address location
Get me the value held at that address (0X7ABC…).

Dereferencing Z twice

**z➔ *(*z)

* (*z)

0X7ABC…

Deference pointer Z ➔
Gives you the value of Y

* (0X7ABC…)

Deference 0X7ABC… ➔
Gives you the value at the
Location 0X7ABC….

C++: Dynamic Memory Allocation

Previously

```
int numbers[1000];
```

- Problem
  - Memory requirements are generally not known ahead of time or at compile time
    - Each execution might need different memory capacity
    - Leads to either memory wastage or insufficiency

C++: Dynamic Memory Allocation

Memory at Runtime

- The ***new*** operator
  - Creates a dynamic variable of a specified type and size
  - **ALWAYS** Returns a pointer
- Request for memory allocation
  - If sufficient memory is available
    - new initializes the memory by calling object constructors
    - Return a pointer to the memory location
  - If sufficient memory is *not* available
    - Throws a std::bad_alloc memory exception or returns NULL (based on the compiler edition)

C++: Dynamic Memory Allocation

Dynamic Array

- The ***new []*** operator
  - Creates a dynamic variable of a specified type
  - **ALWAYS** Returns a pointer

```
int* dArray = new int[20]
```

Will try to create a dynamic memory space to store 20 array elements of type integer

- You can also create array of user defined data types → Struct and Classes

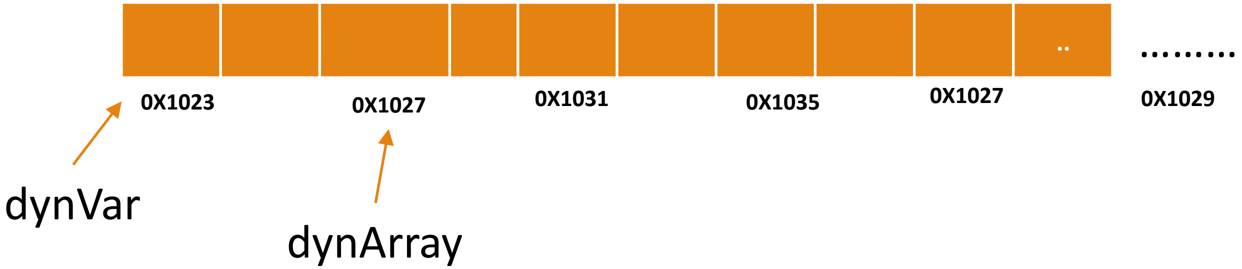
C++: Dynamic Memory Allocation

Refer to: DynamicMemory.cpp

Examples

```
#include <iostream>
#include <stdlib.h>
using std::endl;using std::cout;using std::cin;
int main(){
      /*
       * Either you might waste the memory
       * or
       * It might not be sufficient
       */
      int random[100];
      int size;
      cout<<"How many random # do you want: ";cin>>size;
      for(int i=0;i<size;i++){
            random[i] = rand()%200;
      }
      /*
       * create memory on demand.
       */
      int* dynVar = new int;
      //create what is required.
      int* dynArray = new int[size];
return 0;
}
```

The pointer (dynVar or dynArray) points to the memory location of the assigned memory



C++: Dynamic Memory Allocation

Refer to: DynamicMemory2.cpp

Examples

```
#include <iostream>
#include "Employee.h"
using std::string;
using std::endl;using std::cout;using std::cin;
int main(){

            Employee e;
            cout<<"Size of Employee is :"<<sizeof(e)<<" bytes"<<endl;

            int noEmp;
            cout<<"# of Employees: "; cin>>noEmp;

            Employee* emp = new Employee[noEmp];

            for(int i=0;i<noEmp;i++){
                  cout<<"First Name : ";cin>>emp[i].firstName;
                  cout<<"Last Name : ";cin>>emp[i].lastName;
                  cout<<"Address: ";cin>>emp[i].lastName;
            }

            delete [] emp;
            emp = NULL;

}
```

C++: Delete Operator

— Delete

- Will free the memory associated with a dynamic variable

  delete *pointer*

  will delete or “free” the memory allocated to dynamic variable p1

  delete [] *arrayPoiner*

  will delete or “free” the memory allocated to dynamic variable dArray

Stack vs. Heap

- What is the difference between

```
int a = 10;
or
Int* p= new int ;

int a[100];
or
int* p= new int[100]  ;


Person p;
or
Person* p= new Person;
```

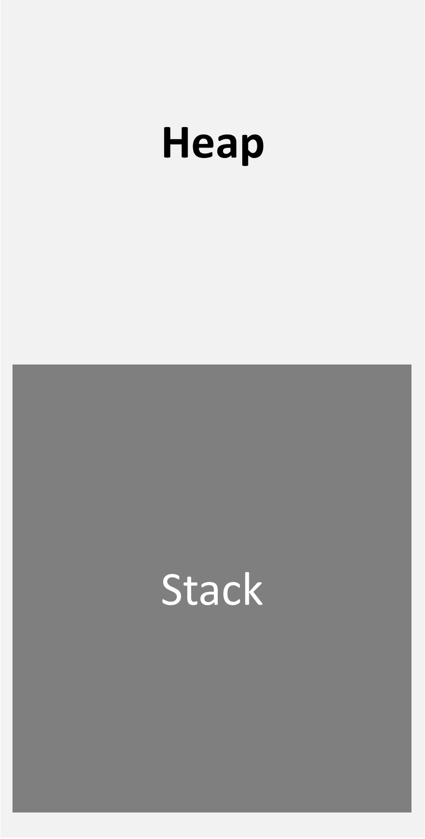
Stack
vs
Heap

Stack vs. Heap

— Stack Memory

- » Keeps track of the program and its execution
- » Follows Stacks data structure (LIFO)
- » Fixed Size
- » Small capacity
- » Fast performance

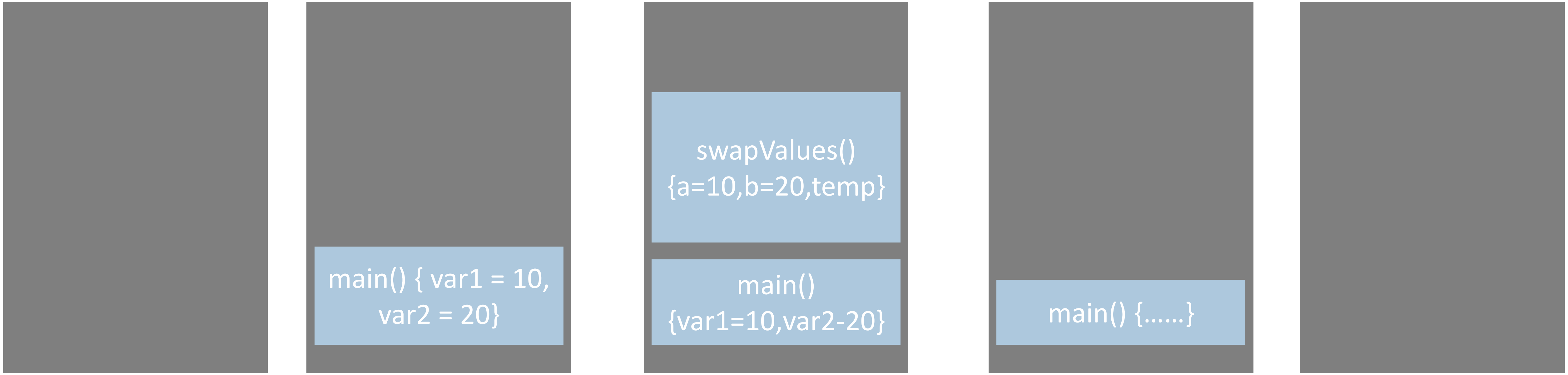
Memory



```
void swapValues(int a, int b){
     int temp = a;
     a= b;
     b = temp;
     cout<<"Inside the swapping function, a: "<<a<<",
     b:"<<b<<endl;

          return;
}



int main(){

     int var1 = 10;
     int var2 = 20;

     swapValues(var1,var2);

}
```

Stack vs. Heap

Stack Memory



Before the program is executed, stack is empty

main() { var1 = 10, var2 = 20}

When main function executed, it is pushed to the stack

swapValues() {a=10,b=20,temp}

main() {var1=10,var2-20}

swapValues function is pushed to the stack

main() {……}

swapValues stack-frame is popped to the when the function is completed

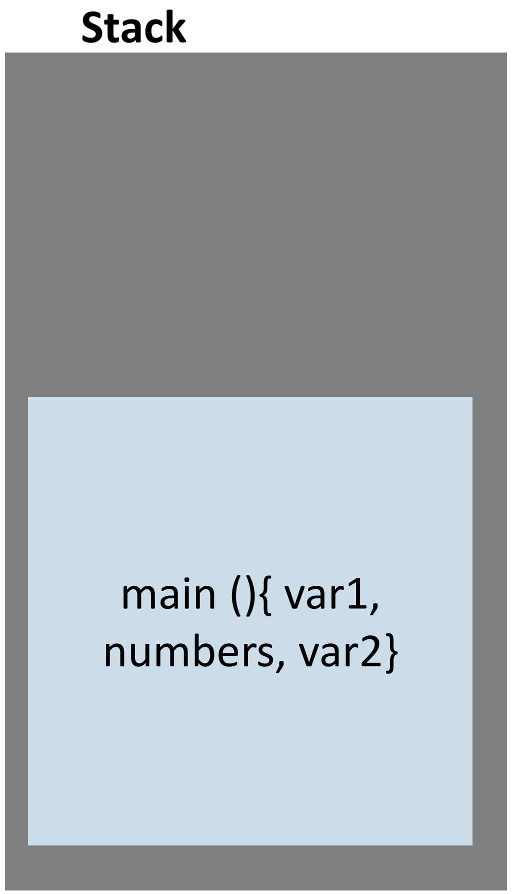
main stack-frame is popped to the stack when completed

Stack vs. Heap

Stack Memory

- Avoid allocating large chucks of memory

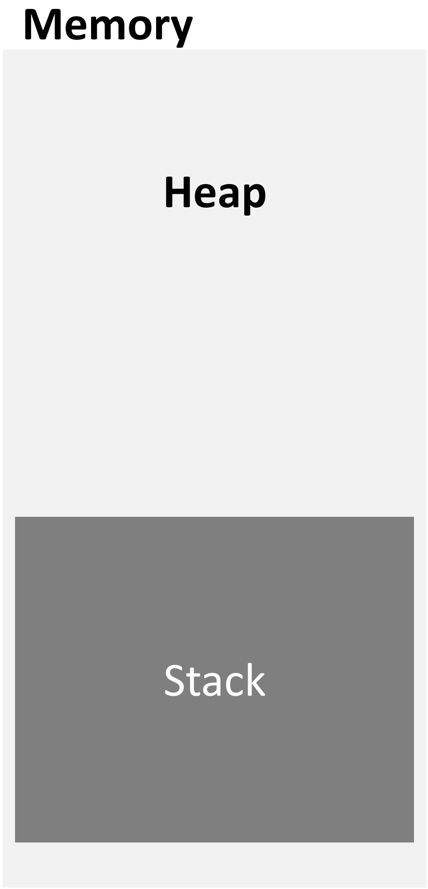
main function takes a large chuck of space in the stack to allocate the required memory for “numbers” (int)



```
int main(){

     int var1 = 10;
     int numbers[300];
     int var2 = 20;

     swapValues(var1,var2);

}
```

Stack vs. Heap

- **Heap Memory**
  - Allocates memory for dynamic variables
  - Large capacity
  - Slow performance

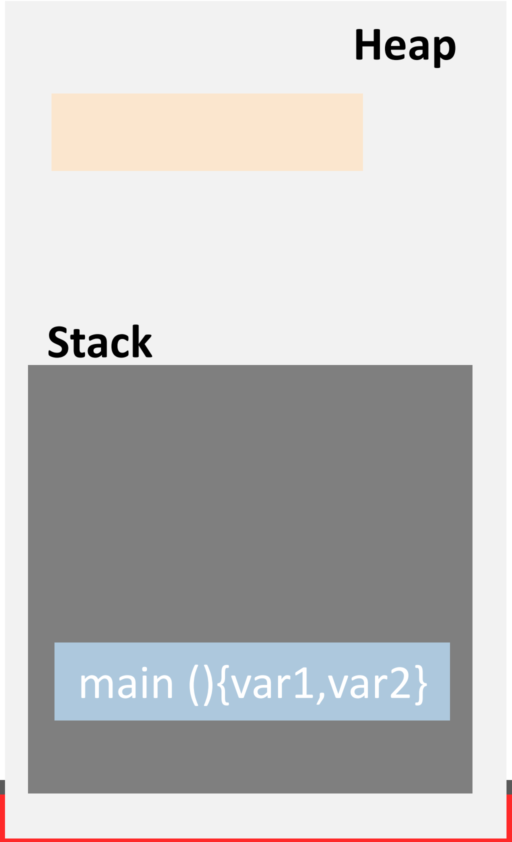


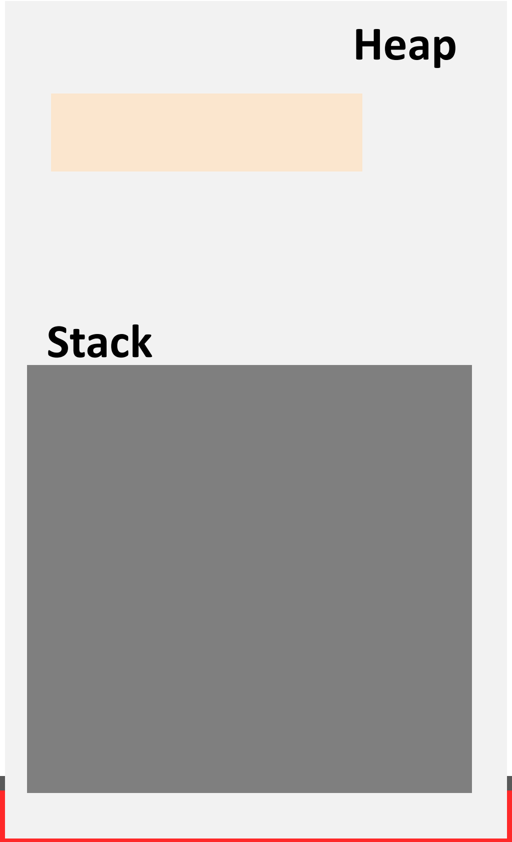
Stack vs. Heap

- **Heap Memory- when you execute this program**

```
int main(){

    int var1 = 10;
    Int* numbers = new int[300];
    int var2 = 20;
}
```

main stack-frame is created and only takes space for var1 and var2





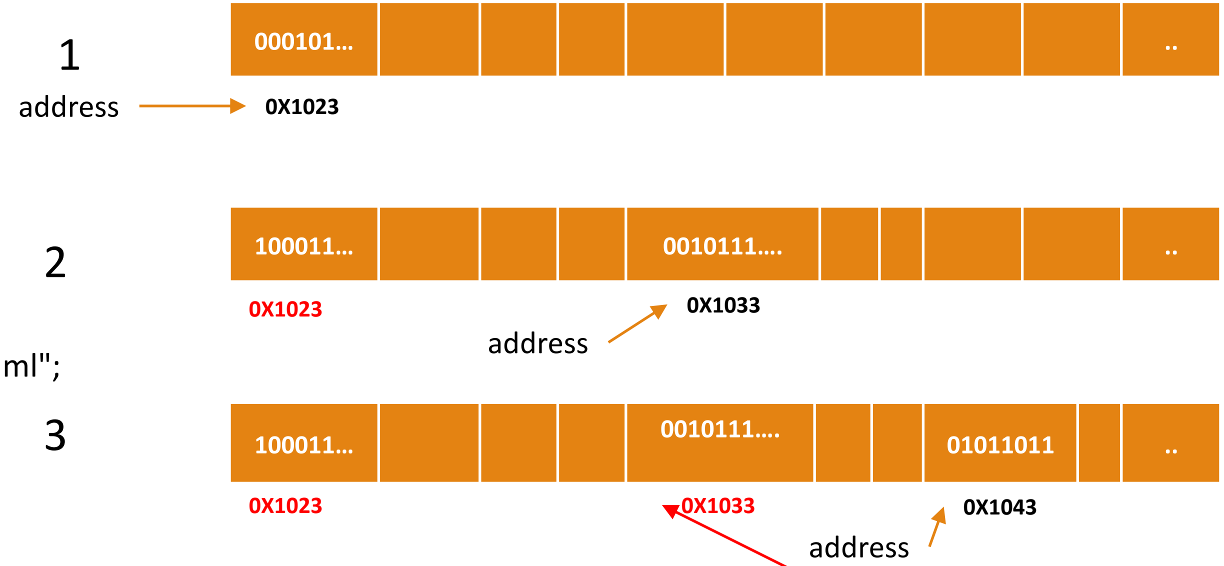
When main is done, the main stack-fame is popped, destroying var1 and v2, *but not* **numbers – as its allocated in heap**

C++: Dynamic Memory Allocation

Refer to: DynamicMemory4.cpp

Memory Leaks

```
int main(){

    1       string* address = new string;
            *address = "5150 School of Information Tech";

    2       address = new string;
            *address = "5150, Box #234, Old Unioun, ISU, Noraml";

    3       address = new string;
            *address = "Lost my job !";

}
```



<u>At Line 1,</u>
address pointer is pointing to **0X1023**
with value
5150 School of Information Tech

<u>At Line 2,</u>
address pointer is pointing to new location **0X1033**
with value **5150, Box #234 ….**
BUT losing the reference to the previous location
NOW, *the previous location cannot be accessed to release the memory, thus causing* ***memory leak***

At Line3,
Reference lost the previous location, as ***address*** pointer is referring the new location

C++: Dangling Pointer

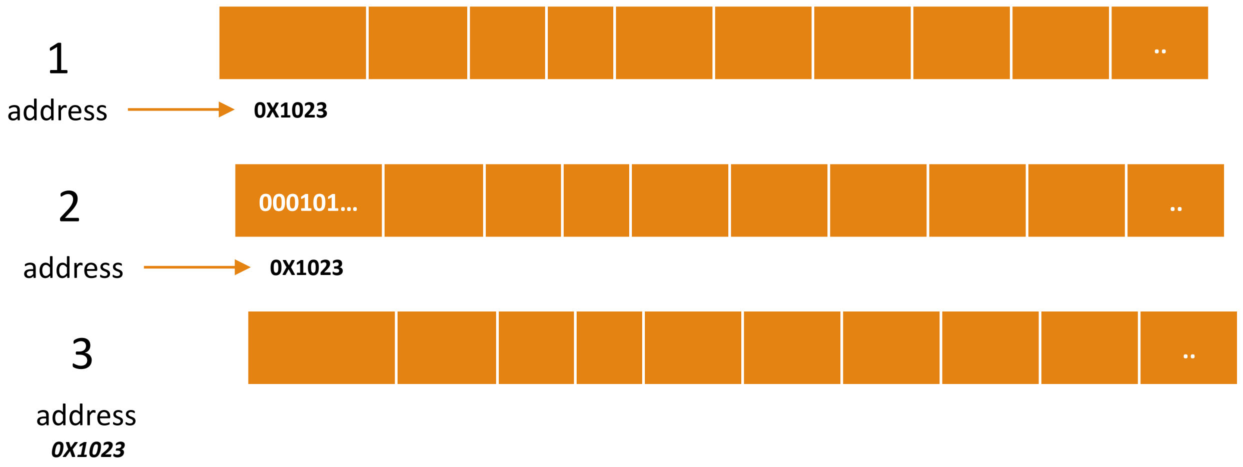
- **Dangling Pointer**
  - When delete is used to free the memory, the memory is returned
    - But, the pointer variable still exists or “dangling” whose value is undefined
    - Any operation (expect assignment) on that pointer is “undefined”
  - No built-in test or algorithm to check for dangling pointer
- **Solution**
  - Set pointer to **nullptr**
- **Dangling Pointer vs. Memory Leak**

C++: Dangling Pointer

Refer to: DynamicMemory3.cpp

Example

```
#include <iostream>
using std::string;
using std::endl;using std::cout;using std::cin;
int main(){

        1       string* address = new string;

        2       *address = "5150 School of Information Tech";

        3       delete address;

        4       cout<<*address<<endl;
}
```



3 Memory is released to the system
BUT the pointer is holding the address value

ERROR: The program is trying to the get the value
using the pointer, but the address doesn’t belong to the current program.

At this line 4, address is called a **dangling pointer**

C++: Dangling Pointer

Refer to: DynamicMemory3.cpp

- Solution

```
#include <iostream>
using std::string;using std::endl;using std::cout;using
std::cin;
int main(){

    string* address = new string;
    *address = "5150 School of Information Tech";

    delete address;

    // Not sure what the outcome will be
    //cout<<"Value is: "<<*address<<endl;

    address = nullptr;

    if(address == nullptr){
         address = new string;
         *address = "PO Box 5150, Normal,IL";
    }

    cout<<"Value is: "<<*address<<endl;

}
```

Note: nullptr was introduced
In C++ 11

Exception Handling

Refer to: Exception.cpp

- **How to handle exceptions**
  - Exception Handling

```
try{
        computation …….
        throw Type
        computation ….
}
catch(Type e){
        computation…….
}
```

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

    std::cout<<"Enter the numerator: ";
    int numerator;
    std::cin>>numerator;

    std::cout<<"Enter the denominator: ";
    int denominator;
    std::cin>>denominator;
    try{
        if(denominator==0)
                throw denominator;
        std::cout<<"Value:
        "<<numerator/denominator<<std::endl;
    }
    catch(int e){ //catch the type of the value thrown
        std::cout<<e<<std::endl;
    }
}
```

Exception Handling

Refer to: Exception.cpp

— Custom Throw type

```
#include<iostream>

class MyException{
    private:
        string msg;
        int n, d;
        void logMessage(){
                cout<<"Message logged:
        "<<msg<<"with vales:"<<n<<d<<endl;
        }
    public:
        MyException(string msg, int n, int d){
            this->msg = msg;
            this->n= n;
            this->d= d;
            logMessage();
        }
        string getMessage(){
            return msg;
        }

};
```

Throw a instance of class.

```
int main(){
    cout<<"Enter the numerator: ";
    int numerator;
    cin>>numerator;

    cout<<"Enter the denominator: ";
    int denominator;
    cin>>denominator;
    try{
        if(denominator<=0){
            MyException exe(string("denominator is
            zero"), numerator, denominator);
            throw exe; // throw any value
        }
        cout<<"Value: "<<numerator/denominator<<endl;
    }
    catch(MyException& e){ //catch the type of the
    value thrown
        cout<<e.getMessage()<<endl;
    }
}
```

Thank You

Question, Comments & Feedback