# Memory Allocation

Sandra Batista and Mark Redekopp

1.1-1.2

# VARIABLES & SCOPE

# A Program View of RAM/Memory

Code

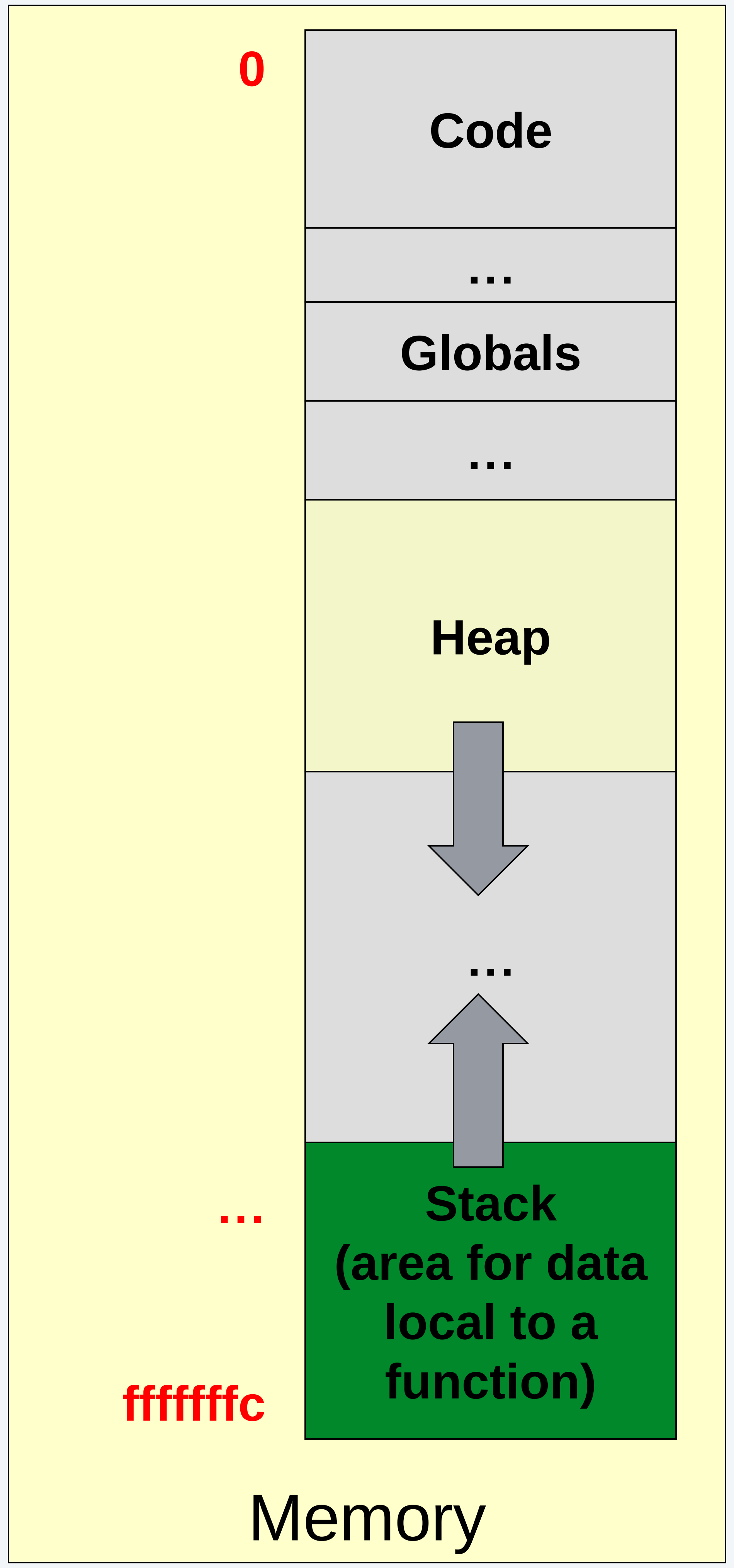
Static memory: Global variables

System stack

- Local variables
- Return link (where to return)
- etc.

Heap or free store: Area of memory that is dynamically allocated

Heap grows downward, stack grows upward…



# Variables and Static Allocation

For every variable in a program there exists

- Name (by which *programmer* references it)
- Address (by which *computer* references it)
- Value

Every variable has **scope**

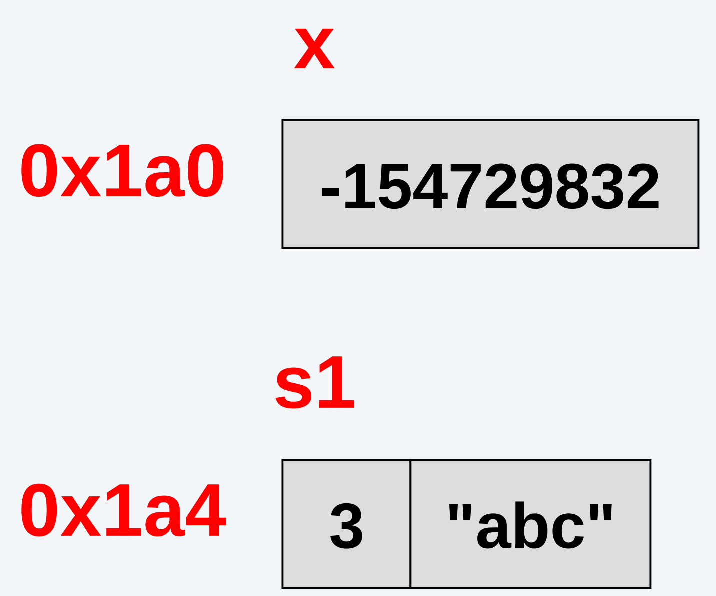
Automatic/Local Scope

- {…} of a function, loop, or if
- On the stack
- Deallocated when the '}' is reached

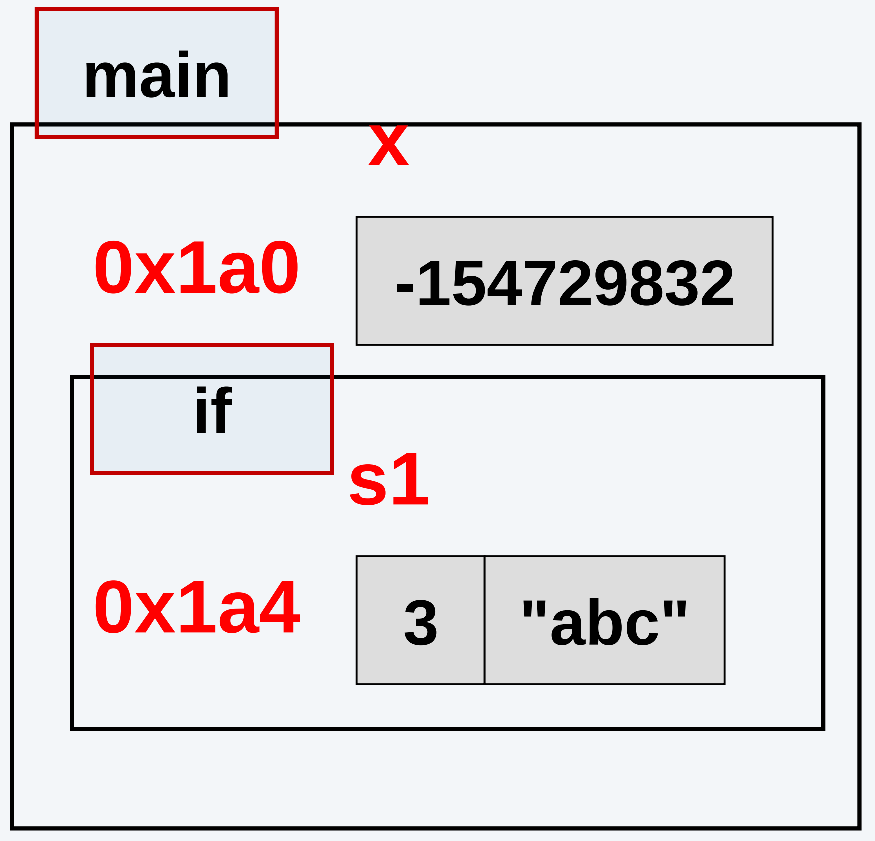
**Code**

```
int x;

string s1("abc");
```

**Computer**




```
int main()
{
  int x; cin >> x;
  if( x ){
    string s1("abc");
  }
}
```

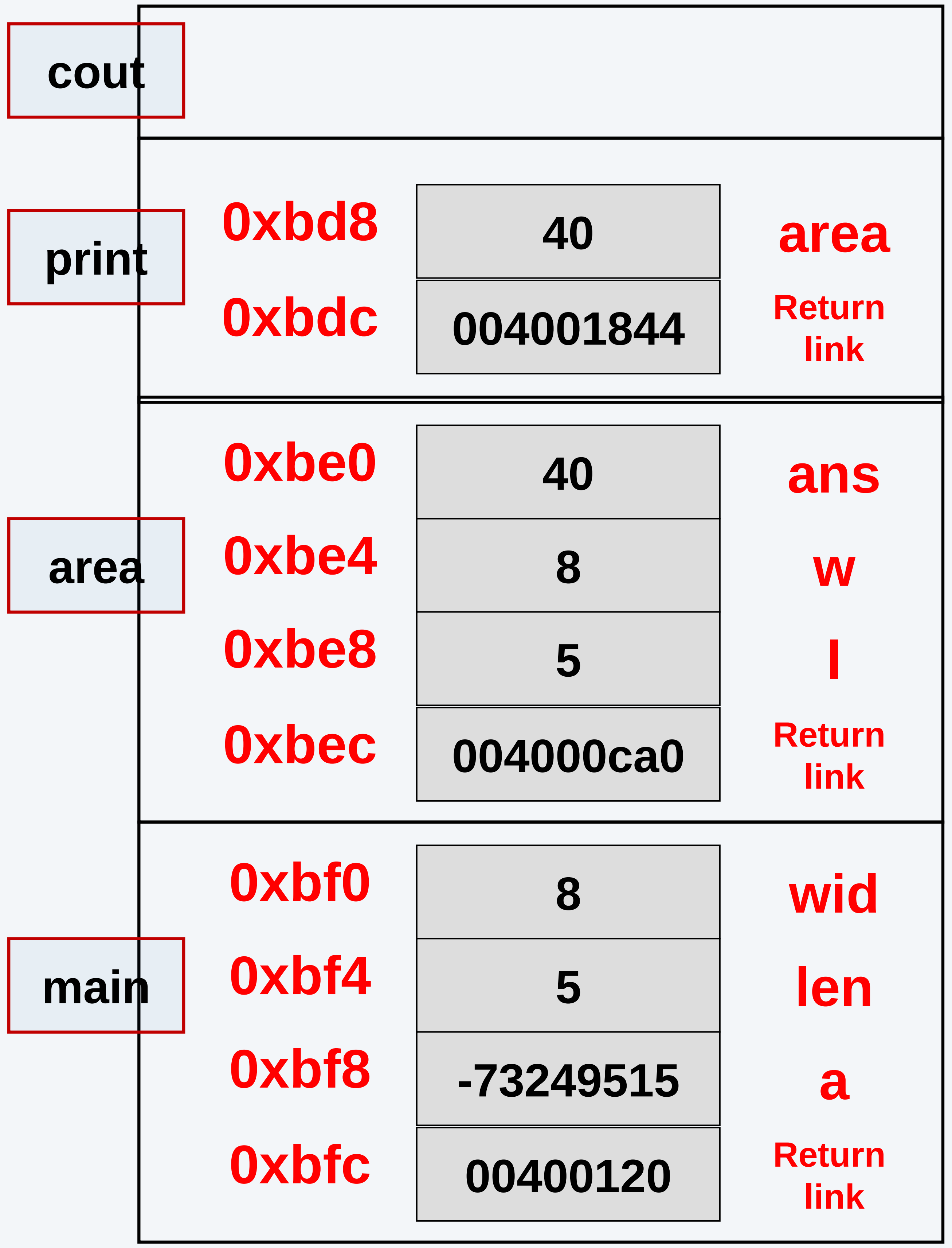


# Automatic/Local Variables

Variables declared inside {…} are allocated on the stack

This includes functions

**Stack Area of RAM**




```
// Computes rectangle area,
//  prints it, & returns it
int area(int, int);
void print(int);
int main()
{
  int wid = 8, len = 5, a;
  a = area(wid,len);
}

int area(int w, int l)
{
  int ans = w * l;
  print(ans);
  return ans;
}

void print(int area)
{
  cout << “Area is “ << area;
  cout << endl;
}
```

# POINTERS & REFERENCES

## Pointers

A variable that stores an address of another variable

Declared with the `type*` syntax (e.g. `int*`, `char*`, `Item*`)

A pointer occupies memory the size of an address on the machine

## C++ Reference Variable

A special variable that provides an **alias** to an already-declared variable

Declared with the `type&` syntax (e.g. `int&`, `string&`, `Item&`)

A reference does not occupy any memory. The compiler uses it to access another variable.

**Important Note**: "Pass-by-reference" can mean **pointers <u>OR</u> C++ Reference Variables.**

Tip: prefer using C++ Reference Variables

# Review of Pointers in C/C++

## Pointer (type *)

- Memory address of a variable
- Pointer to a data-type is specified as *type* * (e.g. `int *`)
- Operators: & and *

  *&object* => ***address-of object (Create a link to an object)***

  **ptr* => ***object located at address given by ptr (Follow a link to an object)***

## Example:

```
int* p, *q; //1
int i, j;   //2

i = 5; j = 10; //3
p = &i; //4
cout << p << endl; //5
cout << *p << endl; //6
*p = j; //7
q = nullptr; //8
```

| | | |
|---|---|---|
| 0xbe0 | 0xbe8 | p |
| 0xbe4 | nullptr | q |
| 0xbe8 | 5 | i |
| 0xbec | 10 | j |

# Pointer Notes

**`A pointer can be set to`** **`nullptr`** (in C++11 and later) to mean that it does not point to any memory

To use **`nullptr`** compile with the C++17 version:

```
$ g++ -std=c++17 -g -o test test.cpp
```

An uninitialized pointer is a pointer waiting to cause a SEGFAULT

- What are they and what causes them?

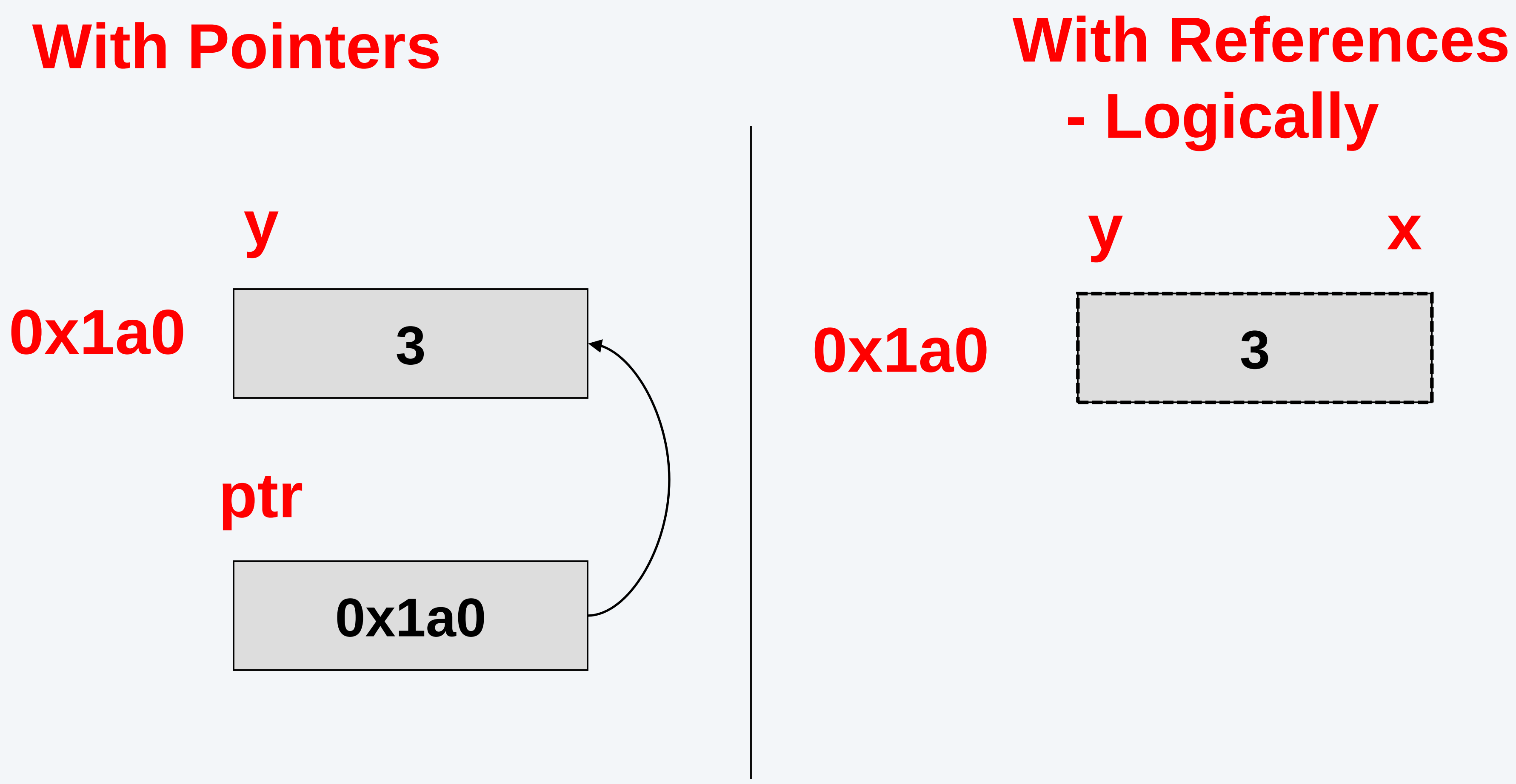
- What tool can help find what is causing SEGFAULTS?

# Using C++ References

Reference type (type &) creates an alias (another name) the programmer/compiler can use for some other variable

A variable declared with an ‘int &’ doesn’t store an int, but is an alias for an actual variable

MUST assign to the reference variable when you declare it.




```
int main()
{
  int y = 3, *ptr;
  ptr = &y;  // address-of
             //  operator


 int &x = y; // reference
             // declaration
  // We’ve not copied y into x.
  // Rather, we’ve created an alias.
  // What we do to x happens to y.
  // Now x can never reference
  //   any other int…only y!


  x++;    // y just got incr.

  int &z;     // NO! must assign

  cout << y << endl;
  return 0;
}
```

# Argument Passing Examples

Pass-by-value => Passes a copy

Pass-by-reference =>

- Pass-by-pointer/address => Passes address of actual variable
- Pass-by-reference => Passes an alias to actual variable

```
int main()
{
  int x=5,y=7;
  swapit(x,y);
  cout <<“x,y=“<< x<<“,”<< y;
  cout << endl;
}

void swapit(int x, int y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

**program output: x=5,y=7**

```
int main()
{
  int x=5,y=7;
  swapit(&x,&y);
  cout <<“x,y=“<< x<<“,”<< y;
  cout << endl;
}

void swapit(int *x, int *y)
{
   int temp;
   temp = *x;
   *x = *y;
   *y = temp;
}
```

**program output: x=7,y=5**

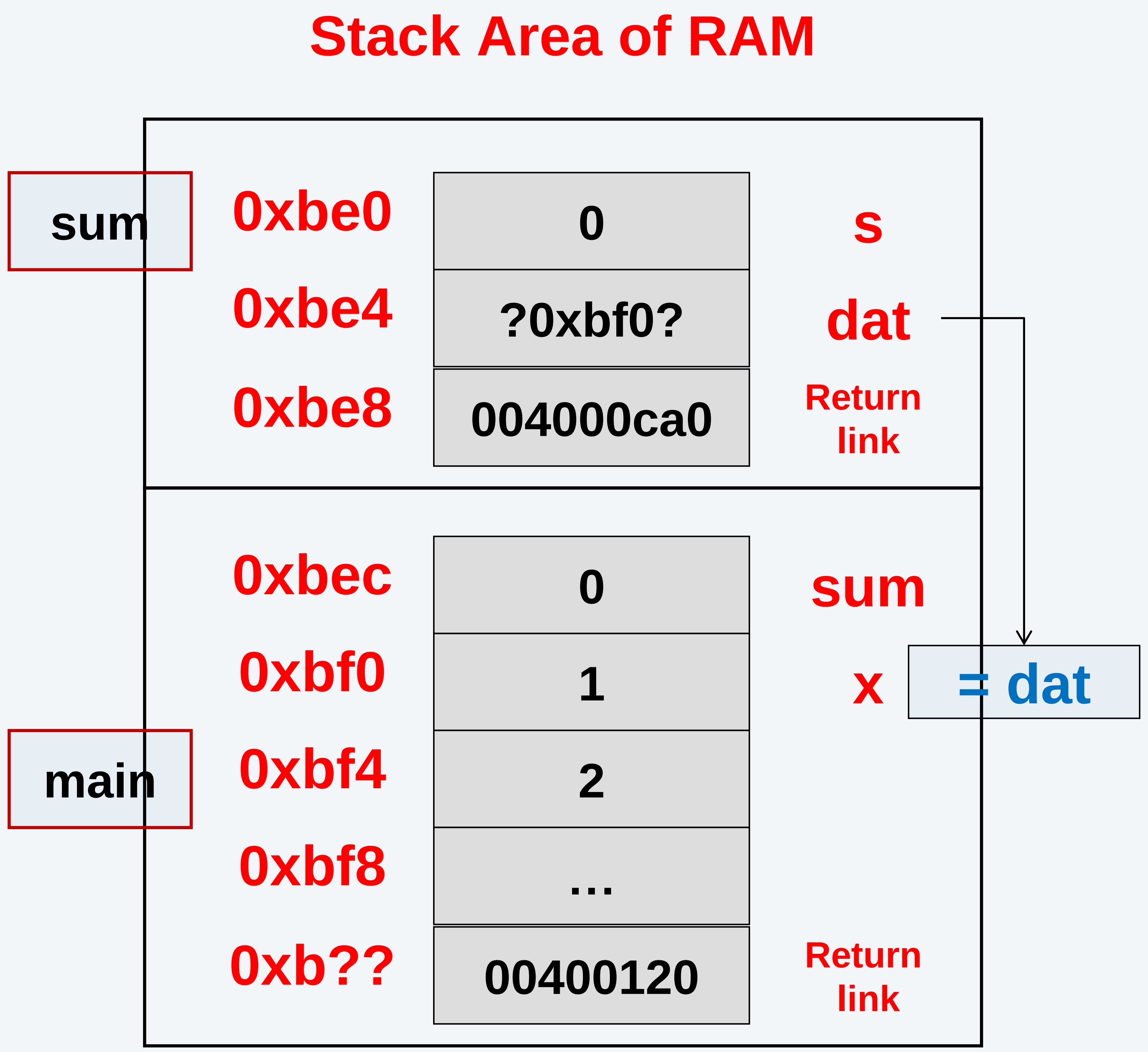
```
int main()
{
  int x=5,y=7;
  swapit(x,y);
 cout <<“x,y=“<< x<<“,”<< y;
  cout << endl;
}

void swapit(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

**program output: x=7,y=5**

# Pass by Reference

## Notice no copy of x need be made since we pass it to sum() by reference

- The **const** keyword tells the compiler to not permit the vector to be modified




```cpp
// Computes the sum of a vector
int sum(const vector<int>&);

int main()
{
  int result;
  vector<int> x = {1,2,3,4};
  result = sum(x);
}

int sum(const vector<int>& dat)
{
  int s = 0;
  for(int i=0; i < dat.size(); i++)
  {
     s += dat[i];
  }
  return s;
}
```

# DYNAMIC ALLOCATION

# Dynamic Memory & the Heap

Code

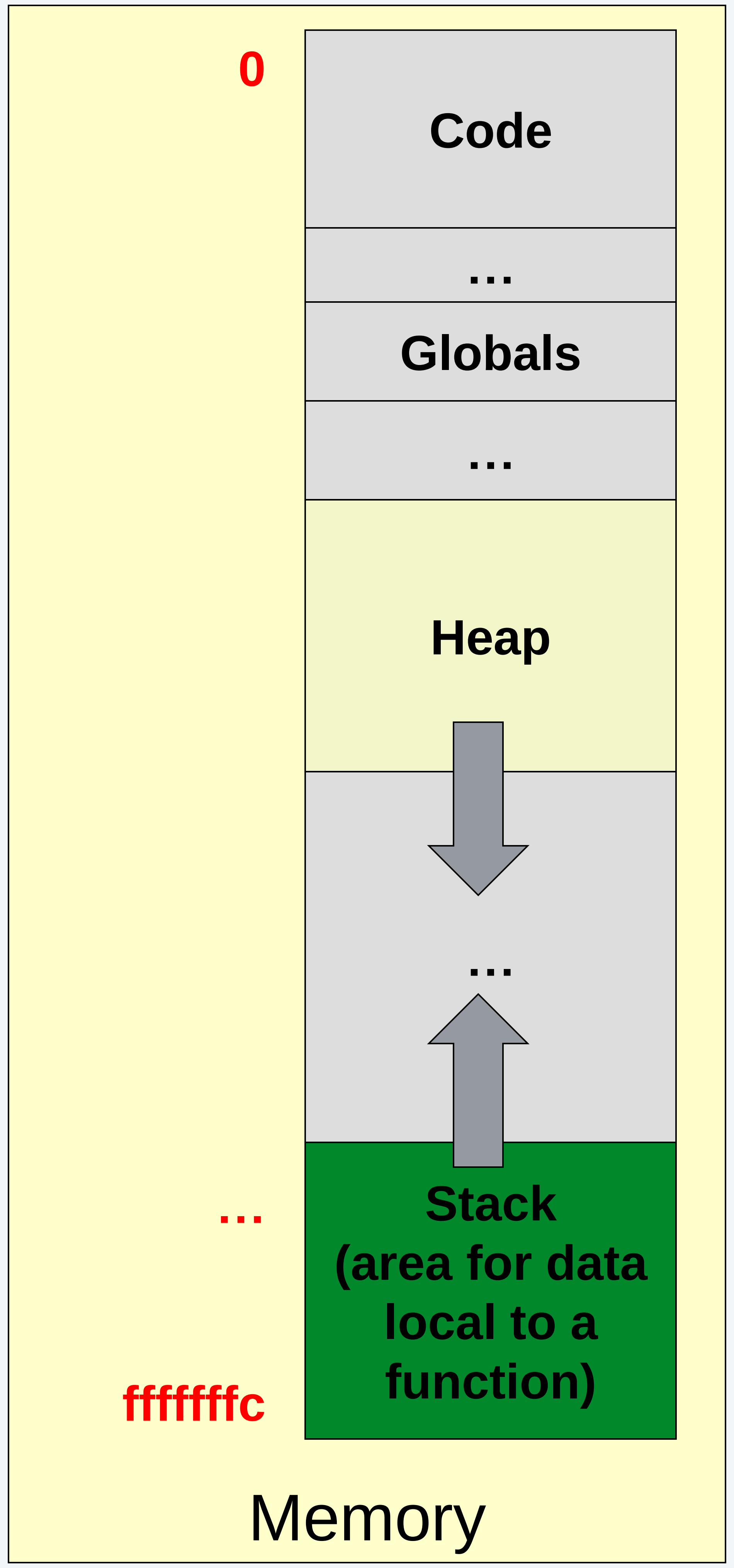
Static memory: Global variables

System stack

- Local variables
- Return link (where to return)
- etc.

Heap or free store: Area of memory that is dynamically allocated

Heap grows downward, stack grows upward…



# C++ `new` & `delete` operators

## new allocates memory from heap

- followed with the type of the variable you want or an array type declaration

```
double *dptr = new double;

int *myarray = new int[100];
```

- returns a pointer of the appropriate type

## delete returns memory to heap

- followed by the pointer to the data you want to de-allocate

```
delete dptr;
```

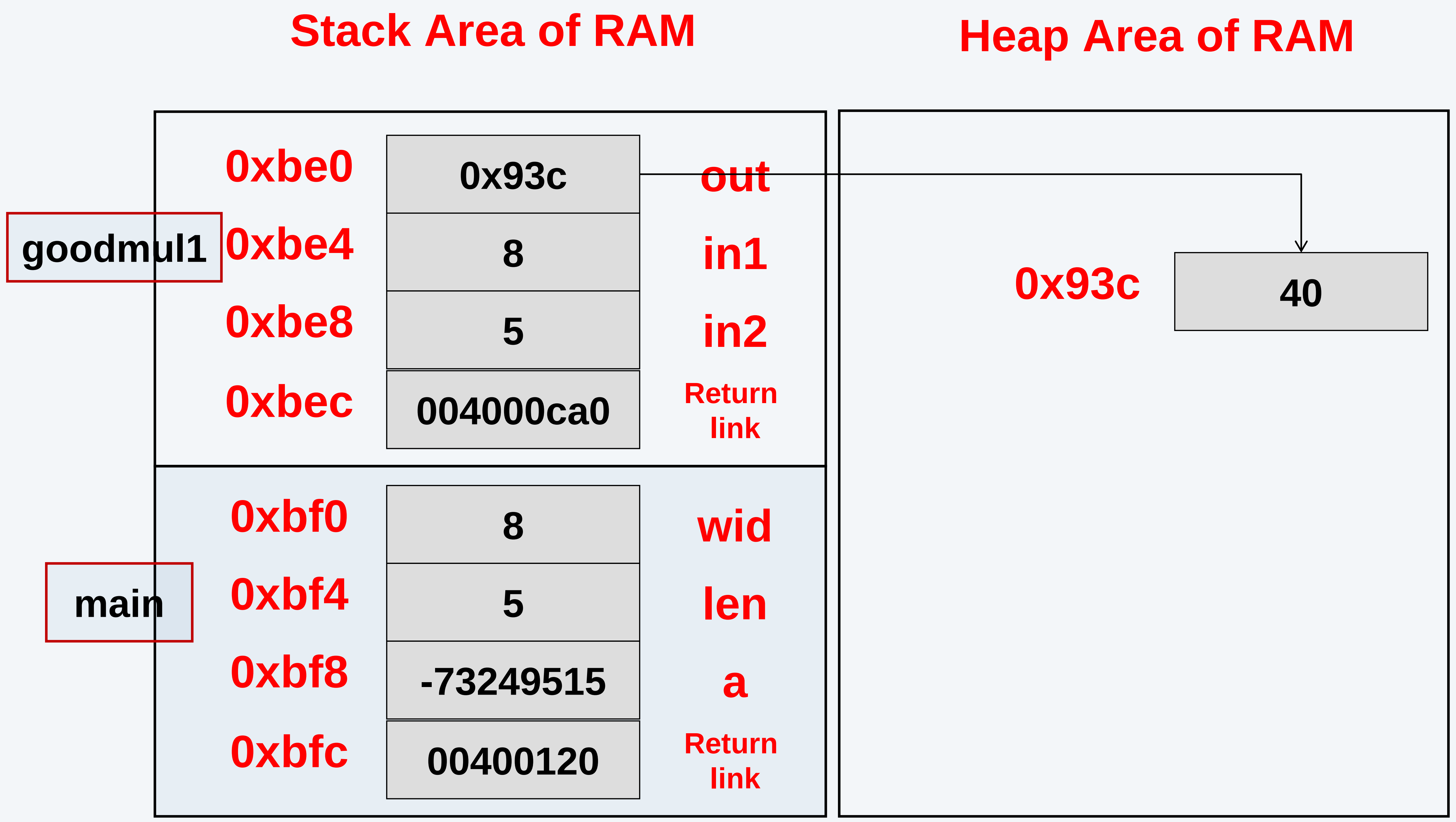
- use `delete []` for pointers to arrays

```
delete [] myarray;
```

# A Paradigm Shift

What can go wrong when a function returns memory from the heap?

Who owns memory returned from the heap and who must delete it?




```
// Computes the product of in1 & in2
int* badmul1(int in1, int in2);
int* goodmul1(int in1, int in2);

int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  //delete a;
  return 0;
}

// Bad! Returns a pointer to a var.
// that will go out of scope
int* badmul1(int in1, int in2)
{
  int out = in1 * in2;
  return &out;
}

// Good! Returns a pointer to a var.
// that will continue to live
int* goodmul1(int in1, int in2)
{
  int* out = new int;
  *out = in1 * in2;
  return out;
}
```

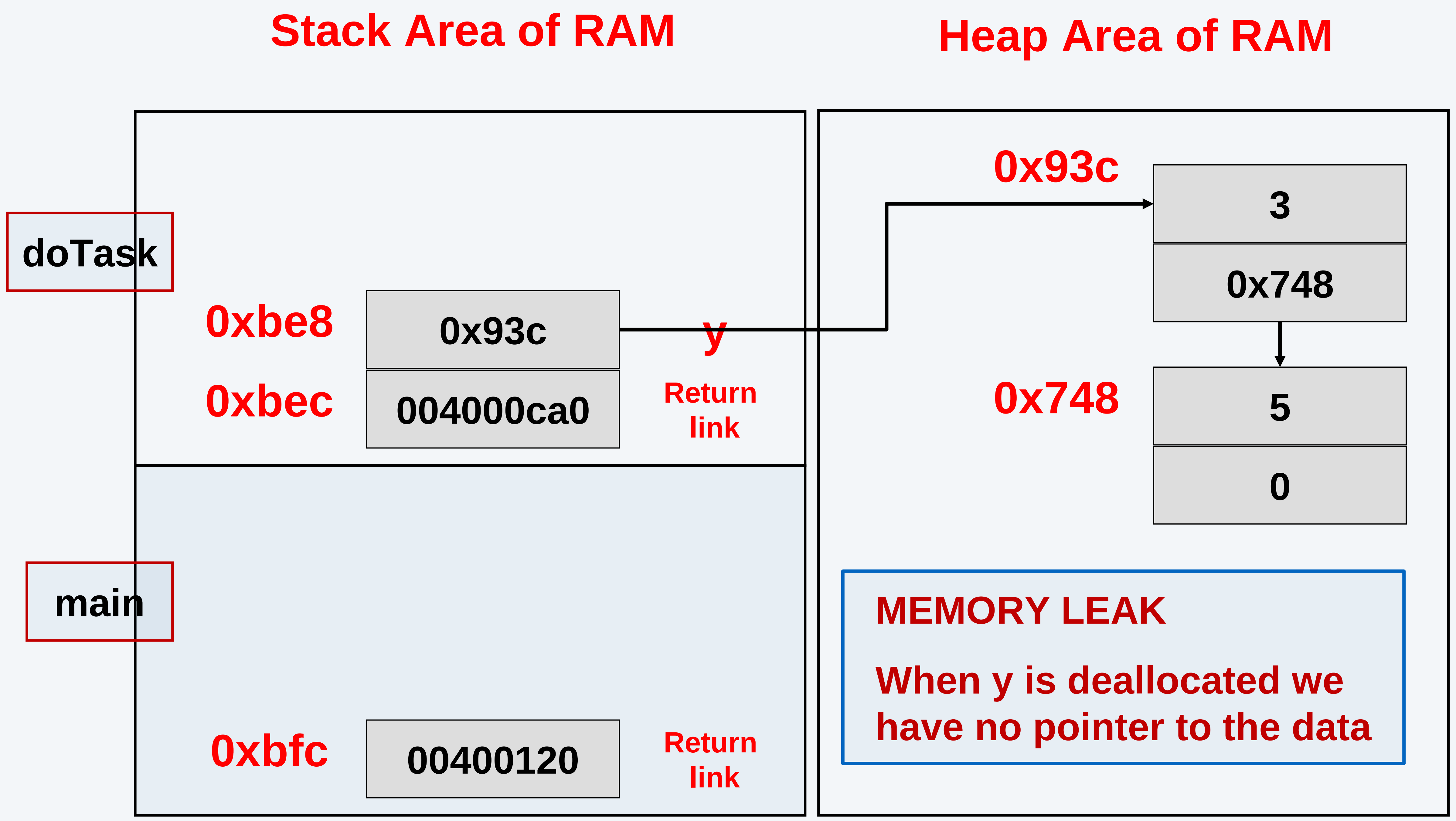
# Dynamic Allocation

The LinkedList object is allocated as a static/local variable

- But each element is allocated on the heap

When y goes out of scope only the data members are deallocated

- You may have a memory leak




```
struct Item {
  int val;  Item* next;
};
class LinkedList {
  public:
   // create a new item
   // in the list
   void push_back(int v);
  private:
   Item* head;
};

int main()
{
  doTask();
}

void doTask()
{
  LinkedList y;
  y.push_back(3);
  y.push_back(5);
  /* other stuff */
}
```

# New Paradigm: RAII

**Resource Acquisition is Initialization (RAII)**

1. Acquire resources in constructor for objects

2. Release the resources in the matching destructor

**Key point: Use C++ classes that manage resources for programmers** (whenever permissible)

Examples:

1. iostreams for I/IO buffers (e.g. cin, cout, cerr)

2. C++ Strings for character buffers

3. STL vector for variable sized array

4. STL containers such as vector, map, unordered_map, list, stack and queue

5. fstreams for files

```
struct Item {
  int val;  Item* next;
};
class LinkedList {
  public:
//destroys items when list is
//out of scope.
  ~LinkedList();
   // create a new item
   // in the list
   void push_back(int v);
  private:
   Item* head;
};

int main()
{
  doTask();
}
void doTask()
{
  LinkedList y;
  y.push_back(3);
  y.push_back(5);
  /* other stuff */
}
```

# RAII for Memory

- The **<memory> library** contains classes for managing pointers: **unique_ptr, shared_ptr, weak_ptr**

- These ptr classes are abstractions for memory management.

- The ptr objects hold raw pointers and can be used syntactically like built-in raw pointers

- Unique_ptr and shared_ptr will destroy memory that it points to when it goes out of scope or no longer used.

# Unique_ptr

- **std::unique_ptr<type> is for *exclusive ownership* of memory at address.**
- **Only one std::unique_ptr can own a  raw pointer (or physical memory address).**
- As a result, unique_ptrs can be moved or returned from functions transferring ownership of the raw pointers.
- Unique_ptrs *cannot be copied or assigned* because two unique_ptrs cannot own same raw pointer.
- Unique_ptrs automatically destroy memory contained in their raw pointers when destroyed using delete by default

# Unique_ptr Declaration

**Instantiate a unique_ptr<type> using constructor and new for the type.**

**Preferably instantiate unique_ptr<type> using make_unique**

Use the unique_ptr as you would a raw pointer on the object.

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

struct Student {
   int id;
   string name;
   Student():id(0), name(""){}
   Student(int i, string n): id(i), name(n){}
};

int main(){
     unique_ptr<Student> sp(new Student(1234, "Jane Doe"));

     unique_ptr<Student> sp2 = make_unique<Student>(2468, "John Clark");

     cout<< "First student ID and name: " << sp->id << " " << sp->name << endl;
     cout<< "Second student ID and name: " << sp2->id << " " << sp2->name << endl;

     return 0;
}
```

# Transferring Unique_ptrs

**Return a unique_ptr<type> from a function**

**Use std::move to move the unique pointer**

Use the reset function of the unique pointer with a raw address

The release function of a unique pointer returns its raw address and sets it to nullptr

```
// include <iostream>, <string>, <memory> and using namespace std;

unique_ptr<Student> add(int ID, string name);

int main(){
    unique_ptr<Student> sp2 = add(12345, "Jane Doe");   //returned from function


    unique_ptr<Student> sp3 = move(sp2);   // sp2 is set to nullptr and the raw pointer is in sp3

    cout<< "student ID and name: " << sp3->id << " " << sp3->name << endl;

    //sp2 = sp3; /*will not compile cannot assign*/

    sp2.reset(sp3.release());

    cout<< "student ID and name: " << sp2->id << " " << sp2->name << endl;         return 0;}


unique_ptr<Student> add(int ID, string name){
          unique_ptr<Student> s = make_unique<Student>(ID,name);
          // unique_ptr<Student> no_copy = s;   /* will not compile cannot copy*/
          return s;
}
```

# Dynamic Arrays using Unique_ptr Declaration

**Instantiate a unique_ptr<type[]> using  new for the type.**

**Preferably instantiate unique_ptr<type[]> using make_unique**

Use the unique_ptr to the array with the subscript operator, operator **[]**

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;

int main(){
   unique_ptr<int[]> int_array(new int[5]);
   for (size_t i =0; i < 5;i++)  int_array[i] = (i+1)*2;

   unique_ptr<string[]>  s_array = make_unique<string[]>(3);
   s_array[0] = "cat";
   s_array[1] = "bird";
   s_array[2] = "dog";

   for (size_t j =0; j <3 ;j++) cout <<s_array[j] << endl;

   int_array.reset(new int[10]);

    return 0;
}
```

# Recommended References for Dynamic Memory

1. Course Lecture Notes Chapter 2 (http://david-kempe.com/teaching/DataStructures.pdf)
2. Lippman,Moo, and Lajoie. C++ Primer. Chapter 12 only sections on unique pointers and dynamic arrays. Available for free from USC library and includes practice exercises. You may skip sections on shared_ptrs and exceptions as we will get back to those in a few weeks. You need only focus on sections 12.1.2, 12.1.5, and 12.2.1 https://uosc.primo.exlibrisgroup.com/permalink/01USC_INST/273cgt/cdi_askewsholts_vlebooks_9780133053036