CSCI 104
Memory Allocation

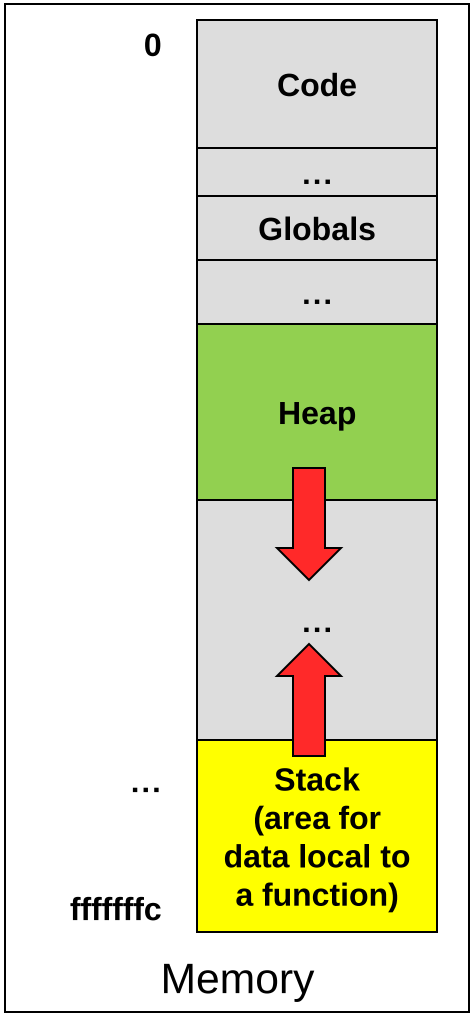
Mark Redekopp

Revised: 01/13/2020

POINTERS, REFERENCES, AND SCOPING REVIEW

A Program View of RAM/Memory

- Code usually sits at low addresses
- Global variables somewhere after code
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.
- Heap: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
- Heap grows downward, stack grows upward…
  - In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error



Variables and Static Allocation

- Every variable/object in a computer has a:
  - Name (by which *programmer* references it)
  - Address (by which *computer* references it)
  - Value
- Let's draw these as boxes
- Every variable/object has **scope** (its lifetime and visibility to other code)
- Automatic/Local Scope
  - {…} of a function, loop, or if
  - Lives on the stack
  - Dies/Deallocated when the '}' is reached
- Logically, let's draw these as nested container boxes

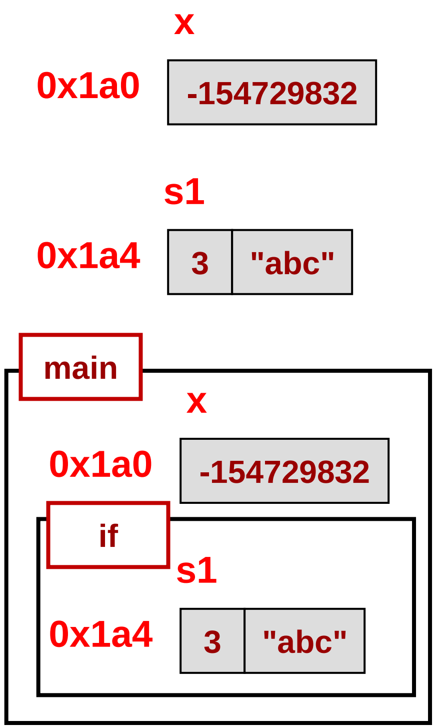
Code

```
int x;

string s1("abc");
```

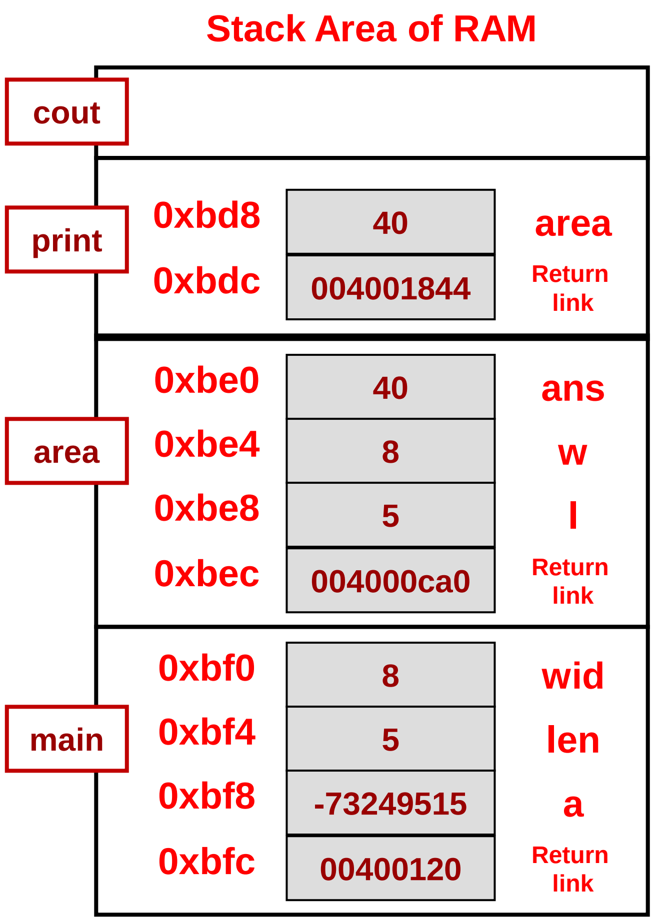
```
int main()
{
  int x; cin >> x;
  if( x ){
    string s1("abc");
  }
}
```

Computer



Automatic/Local Variables

- Physcially, local variables (i.e. those declared inside {…}) are allocated on the stack
- Each function has an area of memory on the stack




```
// 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;
}
```

Kinds of References

Pointers

- A variable (like any other) which occupies memory and stores an address of another variable and can be updated (like any other variable) to store a new address to some other variable
- Declared with the `type*` syntax (e.g. `int*`, `char*`, `Item*`)

C++ Reference Variable

- A special variable that simply gives a second (or third, or fourth) name to an already-declared variable
- Declared with the `type&` syntax (e.g. `int&`, `string&`, `Item&`)
- Does not occupy any memory (just tells the compiler to allow another name to reference some other variable)

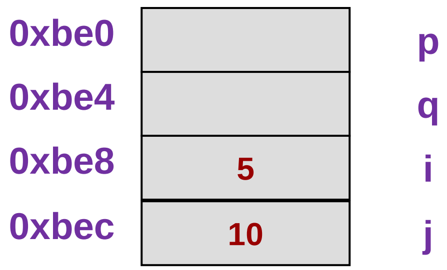
Important Note: When we use the general term "reference" as in "pass-by-reference" we can use EITHER **pointers <u>OR</u> C++ Reference Variables.**
Lets' take a look at each…

Review of Pointers in C/C++

- Pointer (type *)
  - Really just the 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)**
    - `*(&object)` => object [i.e. * and & are inverse operators of each other]
- Example: Indicate what each line prints or what variable is modified. Use **NA** for any invalid operation.

```
int* p, *q;
int i, j;

i = 5; j = 10;
p = &i;
cout << p << endl;
cout << *p << endl;
*p = j;
*q = *p;
q = p;
```



Pointer Notes

- **NULL** (defined in <cstdlib>) or now **`nullptr`** (in C++11) are keywords for values you can assign to a pointer when it doesn't point to anything
  - NULL is effectively the value 0 so you can write:

```
int* p = nullptr;
if( p )
{ /* will never get to this code */ }
```

  - To use **`nullptr`** compile with the C++11 version:

```
$ g++ -std=c++11 -g -o test test.cpp
```

- An uninitialized pointer is a pointer waiting to cause a SEGFAULT
- Beware of SEGFAULTS! What are they and what causes them?
- What tool can help find what is causing SEGFAULTS?

Check Yourself

- Consider these declarations:
  - `int k, x[3] = {5, 7, 9};`
  - `int *myptr = x;`
  - `int **ourptr = &myptr;`
- Indicate the formal type that each expression evaluates to (i.e. `int`, `int *`, `int **`)

To figure out the type of data a pointer expression will yield…
- **Each * in the expression cancels a * from the variable type.**
- **Each & in the expression adds a * to the variable type.**

Orig. Type	Expr	Yields
myptr = int*	*myptr	int
ourptr = int**	**ourptr	int
	*ourptr	int*
k = int	&k	int*
	&myptr	int**

Expression	Type
&x[0]	
x	
myptr	
*myptr	
(*ourptr) + 1	
myptr + 2	
&ourptr	

Using C++ References

- Reference type (type &) creates an alias (another name) the programmer/compiler can use for some other variable
  - Is **NOT** another variable; does **NOT** require memory
- "Syntactic sugar" (i.e. make programmer's life easy) to avoid using pointers
- 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!

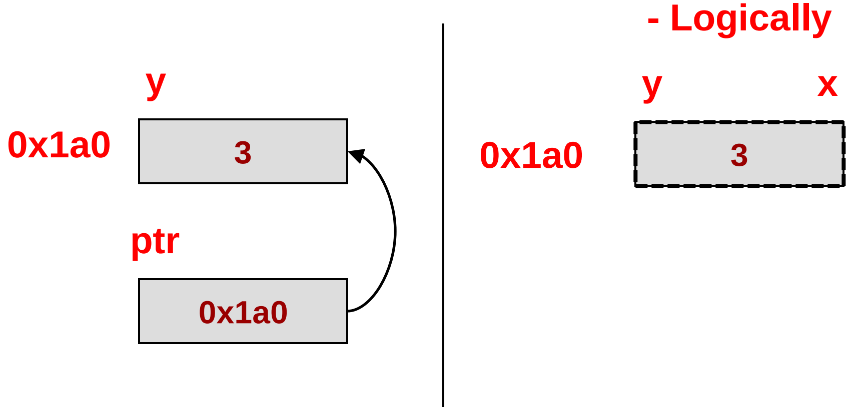

  x++;    // y just got incr.

  cout << y << endl;

  int &z;      // NO! must assign

  int w = 5;
  x = w;  // doesn't make x
          // reference w...copies
          // w into y;
  return 0;
}
```

With Pointers | **With References - Logically**

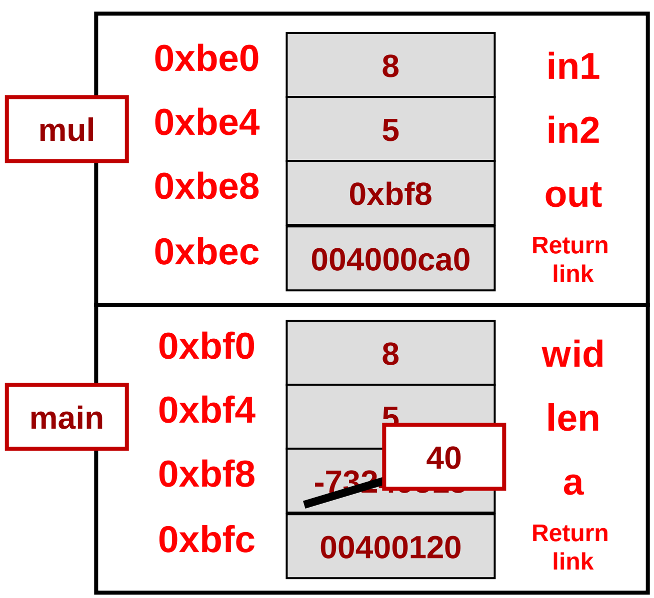


POINTERS, REFERENCES, AND SCOPING ASSESSMENT

Correct Usage of Pointers

- Commonly functions will take some inputs and produce some outputs
  - We'll use a simple 'multiply' function for now even though we can easily compute this without a function
  - We could use the return value from the function but let's practice with pointers
- Can use a pointer to have a function modify the variable of another

Stack Area of RAM




```
// Computes the product of in1 & in2
int mul1(int in1, int in2);
void mul2(int in1, int in2, int* out);


int main()
{
  int wid = 8, len = 5, a;
  mul2(wid,len,&a);
  cout << "Ans. is " << a << endl;
  return 0;
}

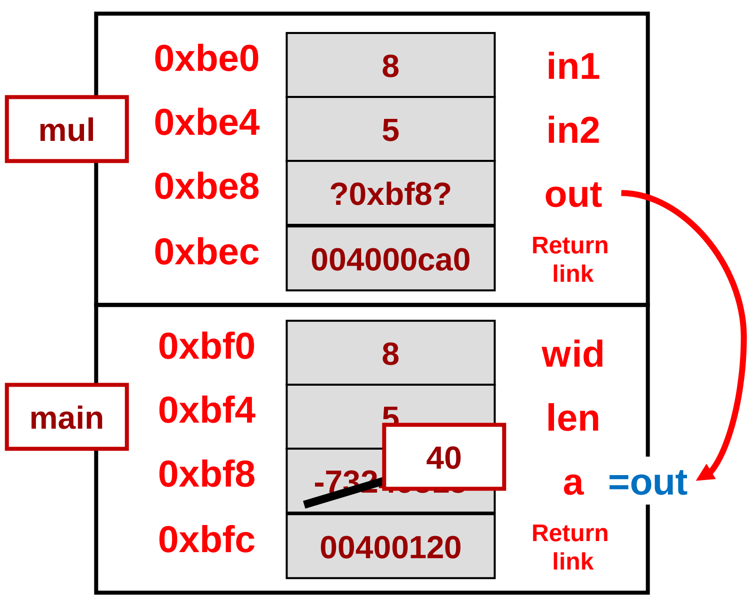
int mul1(int in1, int in2)
{
  return in1 * in2;
}

void mul2(int in1, int in2, int* out)
{
  *out = in1 * in2;
}
```

Now with C++ References

- We can pass using C++ reference
- The reference 'out' is just an alias for 'a' back in main
  - In memory, it might actually be a pointer, but you don't have to dereference (the kind of stuff you have to do with pointers)

Stack Area of RAM




```
// Computes the product of in1 & in2
void mul(int in1, int in2, int& out);

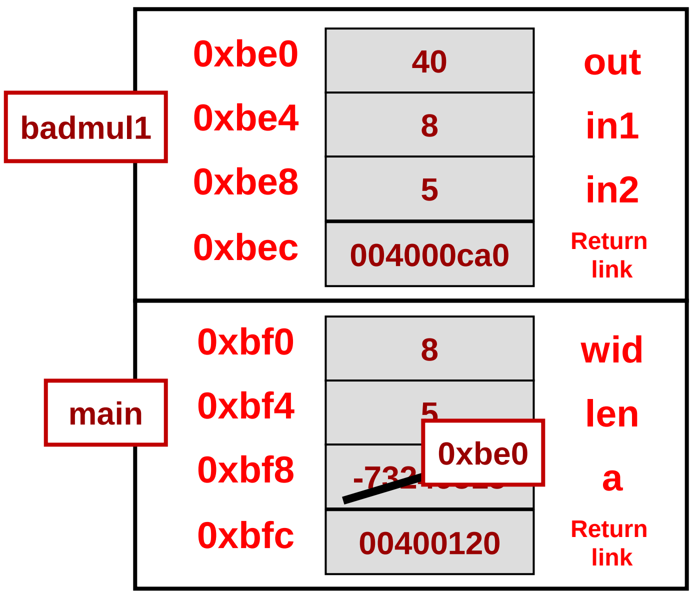
int main()
{
  int wid = 8, len = 5, a;
  mul(wid,len,a);
  cout << "Ans. is " << a << endl;
  return 0;
}

void mul(int in1, int in2, int& out)
{
  out = in1 * in2;
}
```

Misuse of Pointers/References

- Make sure you don't return a pointer or reference to a dead variable
- You might get lucky and find that old value still there, but likely you won't

Stack Area of RAM




```
// Computes the product of in1 & in2
int* badmul1(int in1, int in2);
int& badmul2(int in1, int in2);

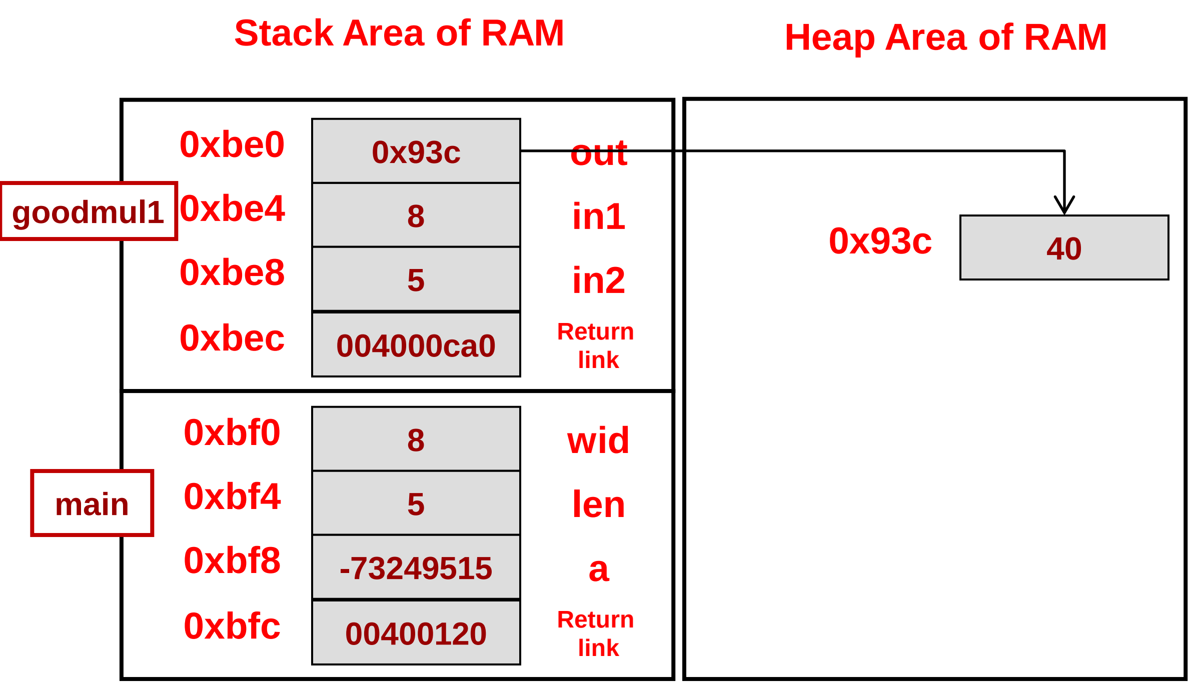
int main()
{
  int wid = 8, len = 5;
  int *a = badmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  return 0;
}

// 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;
}

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

Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw the operation of **goodmul1()**




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

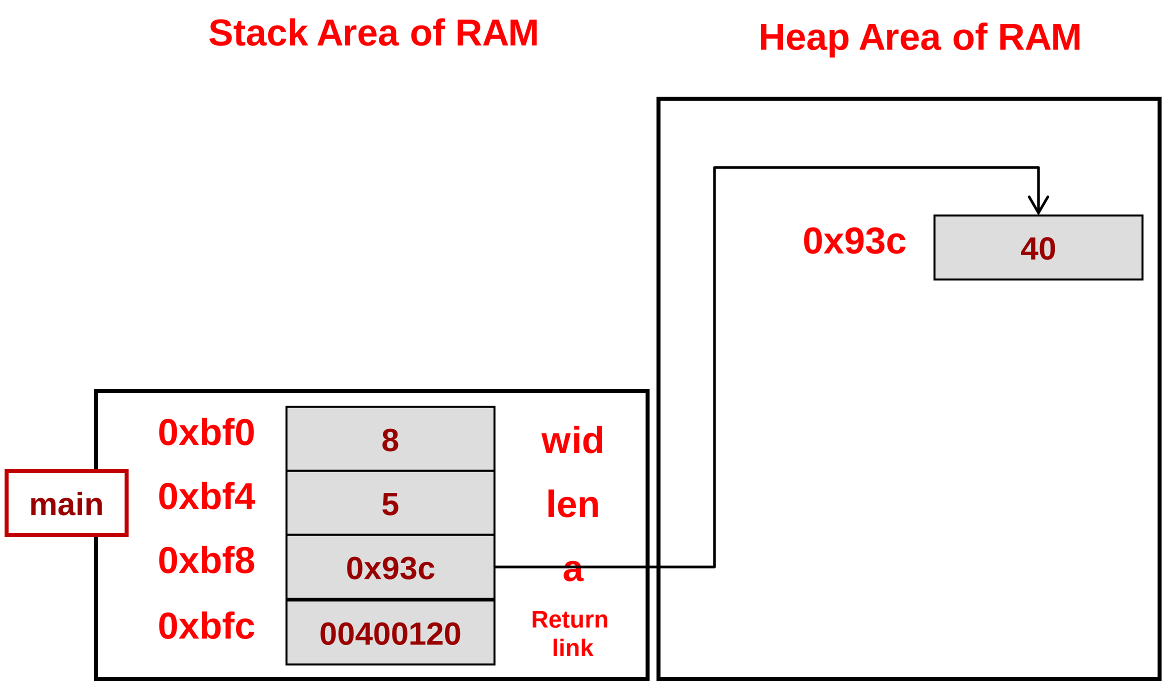
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  delete a;
  return 0;
}

// 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

- When goodmul1() exits, the out pointer goes out of scope
- Thus we need to return the pointer or save it somewhere so that there is a record of our allocation, otherwise we will have a leak




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

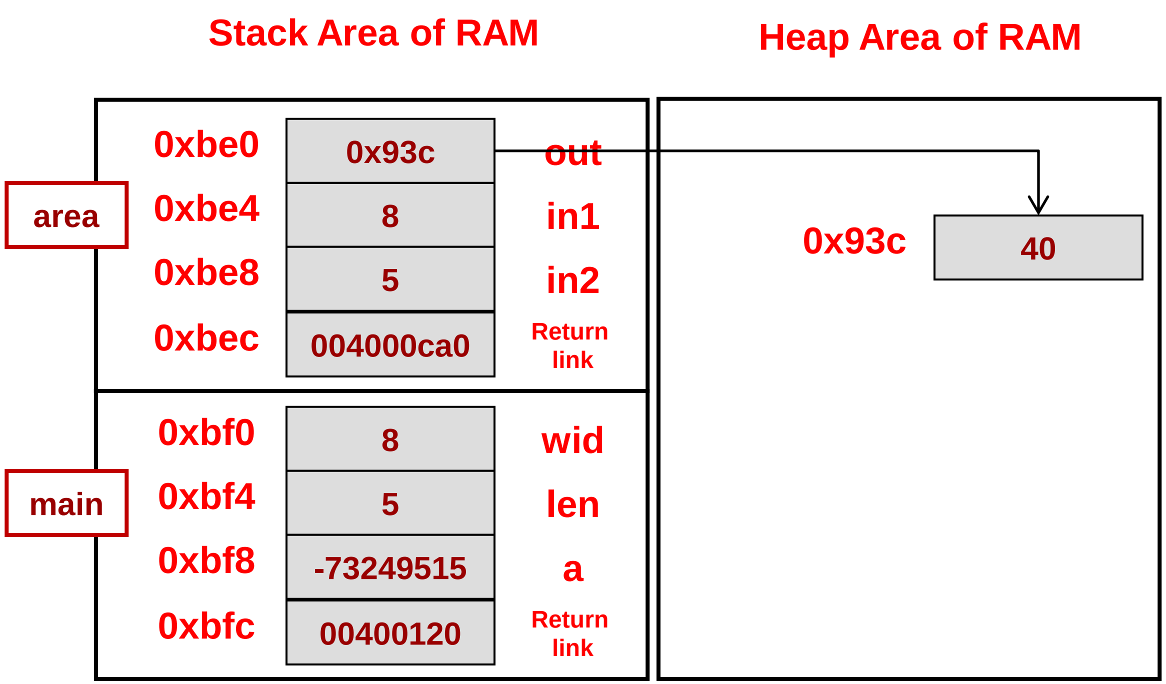
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  delete a;
  return 0;
}

// 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 – Q1

- What happens if we comment the 'delete a' line?




```
// 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 – A1

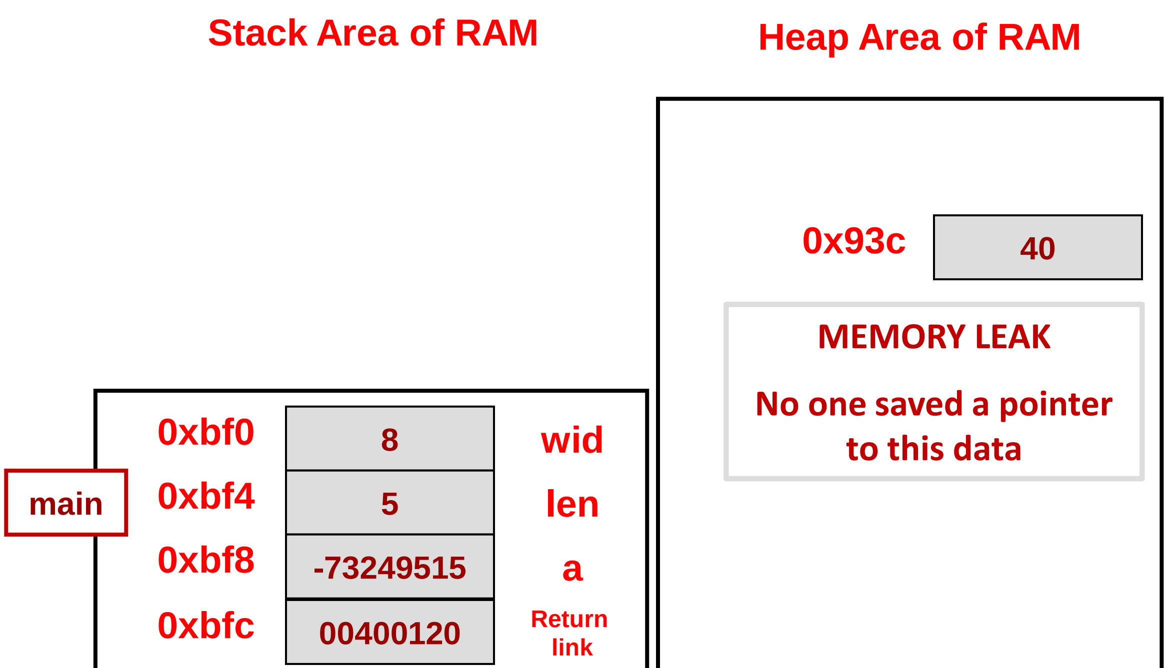
- What happens if we comment the 'delete a' line?
  - **Memory LEAK!!**

```
// 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;
}

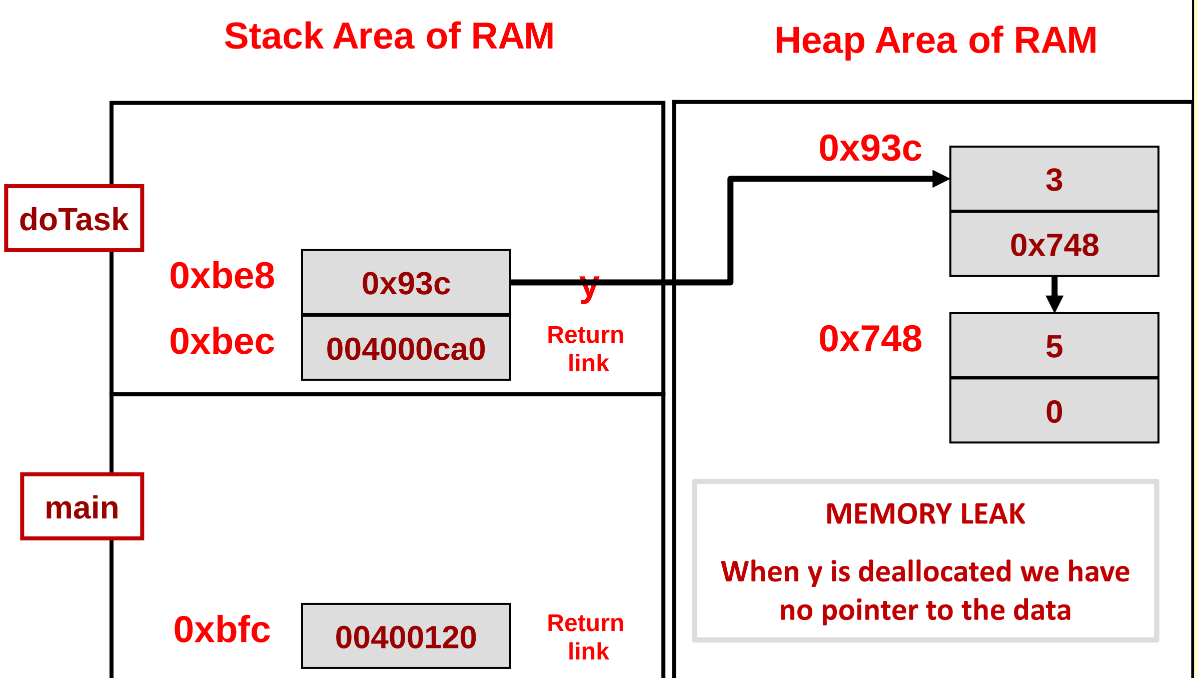
// Bad! Returns a pointer to a var.
// that will go out of scope
int* badmul1(int in1, int in2)
{
  int out = in1 * in2;
  return &out;
}

// 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;
};

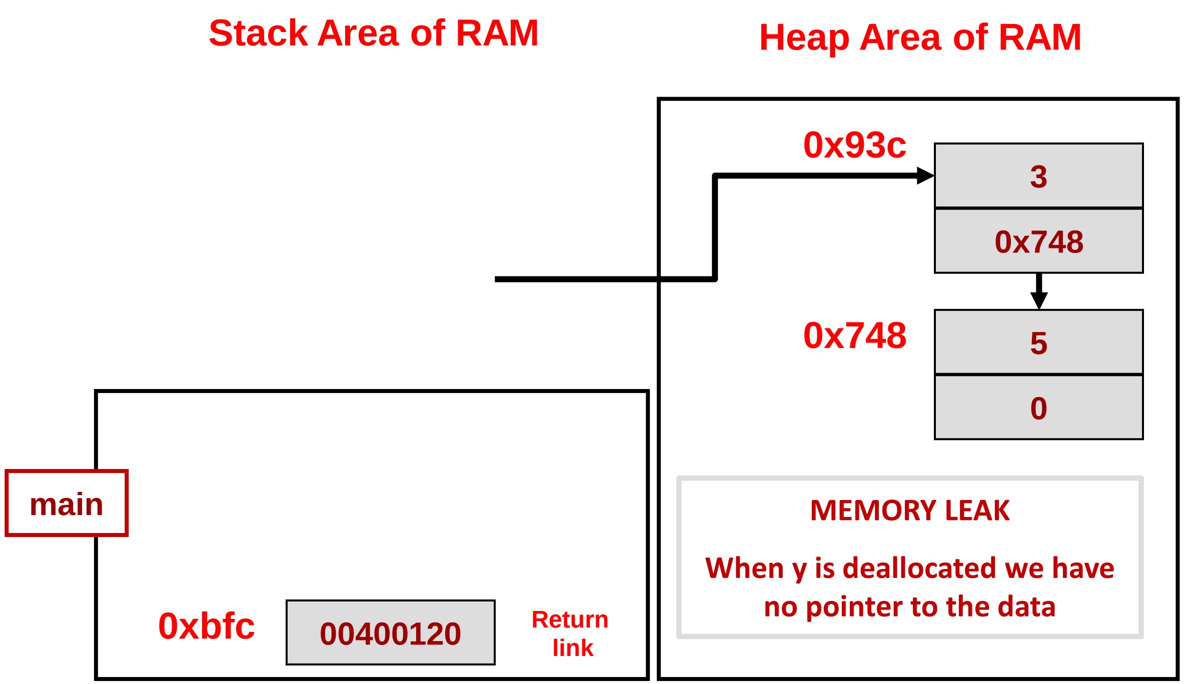
int main()
{
  doTask();
}

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

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

An Appropriate Destructor Will Help Solve This




```
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 */
}
```

If time allows

PRACTICE ACTIVITY 1

Object Assignment

- Assigning one struct or class object to another will cause an element by element copy of the source data destination struct or class

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

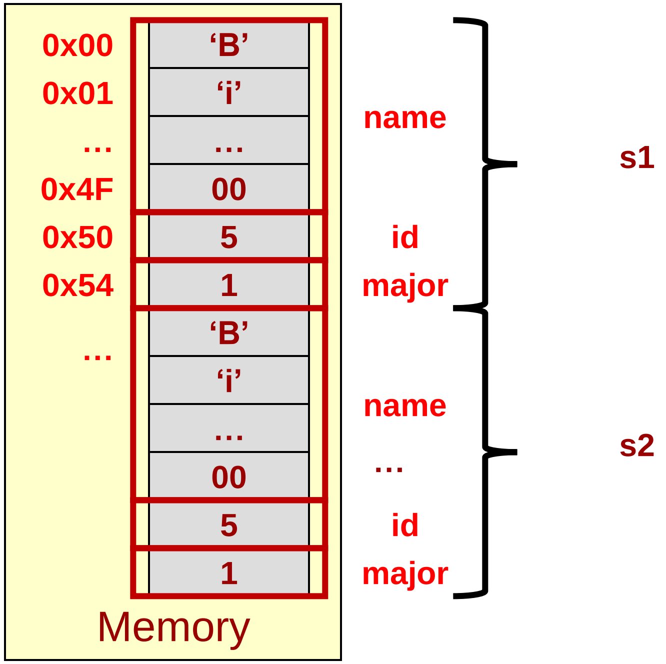
enum {CS, CECS };

struct student {
  char name[80];
  int id;
  int major;
};

int main(int argc, char *argv[])
{
  student s1;
  strncpy(s1.name,"Bill",80);
  s1.id = 5; s1.major = CS;

  student s2 = s1;

  return 0;
}
```



Memory Allocation Tips

- Take care when returning a pointer or reference that the object being referenced will persist beyond the end of a function
- Take care when assigning a returned referenced object to another variable…you are making a copy
- Try the examples yourself
  - $ wget http://ee.usc.edu/~redekopp/cs104/memref.cpp

Understanding Memory Allocation

There are no syntax errors. Which of these can correctly build an Item and then have main() safely access its data

```
class Item
{ public:
   Item(int w, string y);
};
Item buildItem()
{ Item x(4, "hi");
  return x;
}

int main()
{ Item i = buildItem();
  // access i's data.
}
```

ex1

```
class Item
{ public:
   Item(int w, string y);
};
Item& buildItem()
{ Item x(4, "hi");
  return x;
}

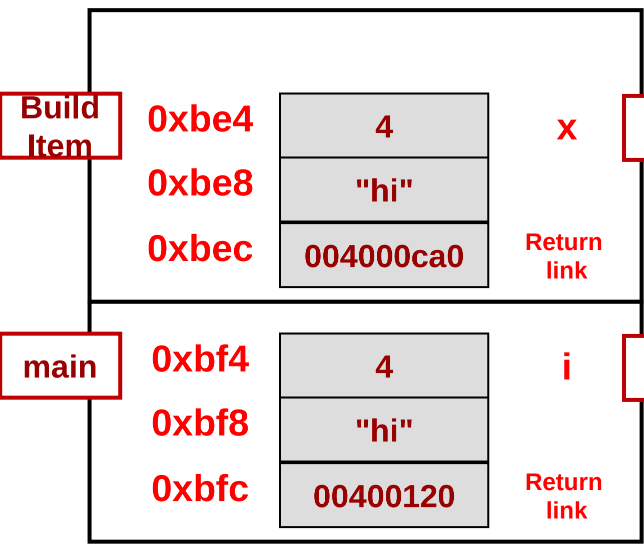
int main()
{ Item& i = buildItem();
  // access i's data
}
```

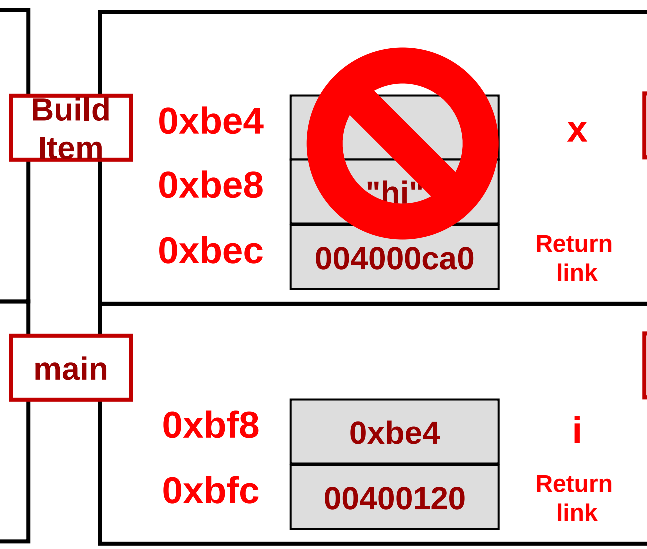
ex2

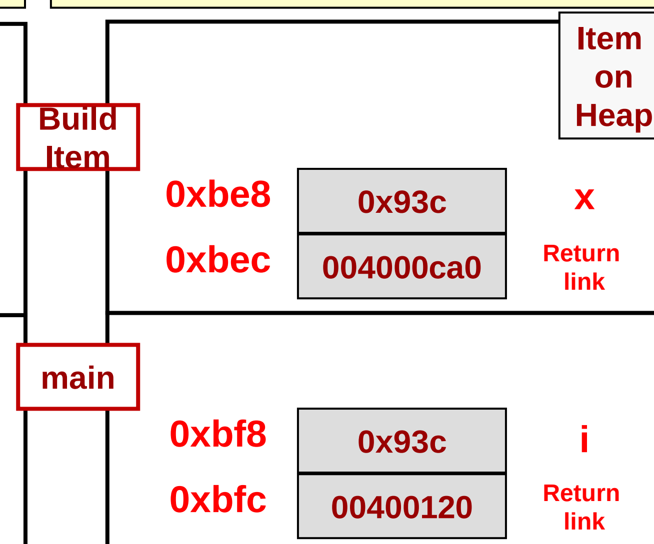
```
class Item
{ public:
   Item(int w, string y);
};
Item* buildItem()
{ Item* x = new Item(4,"hi");
  return x;
}

int main()
{ Item *i = buildItem();
  // access i's data
}
```

ex3







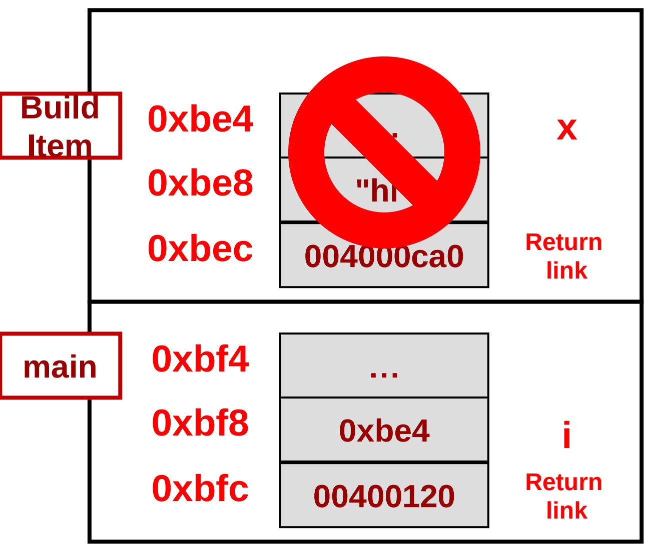
Understanding Memory Allocation

There are no syntax errors. Which of these can correctly build an Item and then have main() safely access its data

```
class Item
{ public:
  Item(int w, string y);
};
Item* buildItem()
{ Item x(4, "hi");
  return &x;
}

int main()
{ Item *i = buildItem();
  // access i's data
}
```

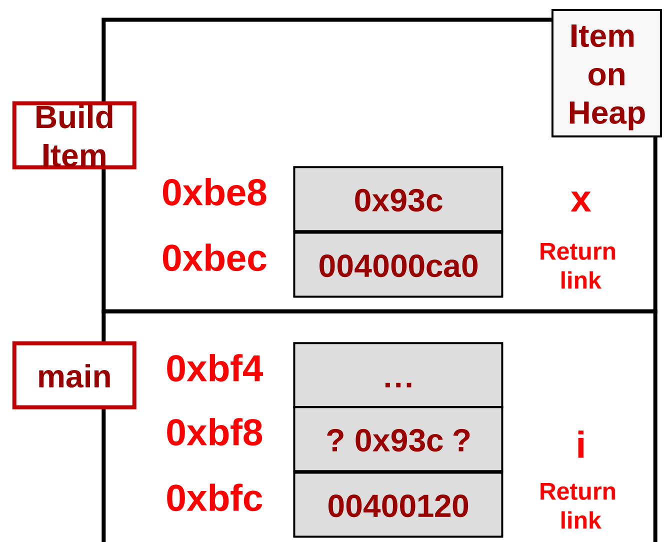
ex4




```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item& i = buildItem();
 // access i's data
}
```

ex5



Understanding Memory Allocation

```
class Item
{ public:
    Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item i = buildItem();
  // access i's data.
}
```

ex6

```
class Item
{ public:
    Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item *i = &(buildItem());
  // access i's data.
}
```

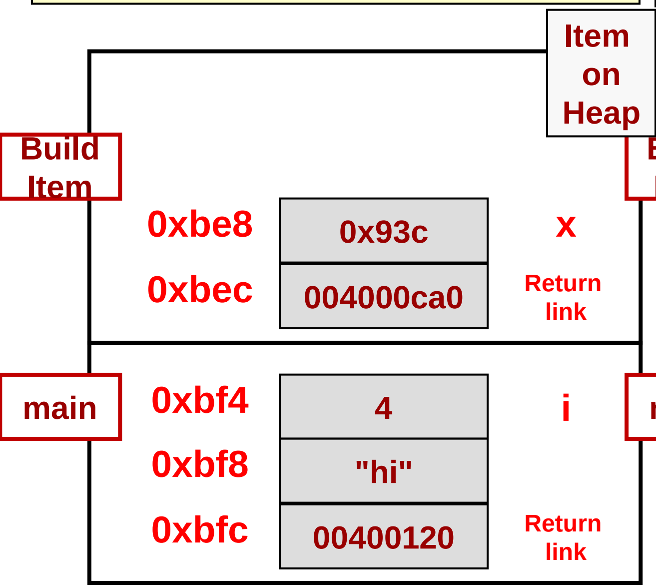
ex7

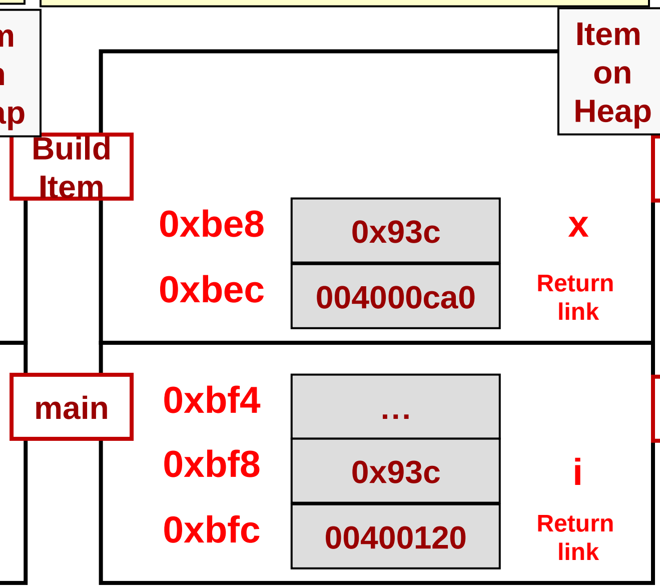
```
class Item
{ public:
    Item(int w, string y);

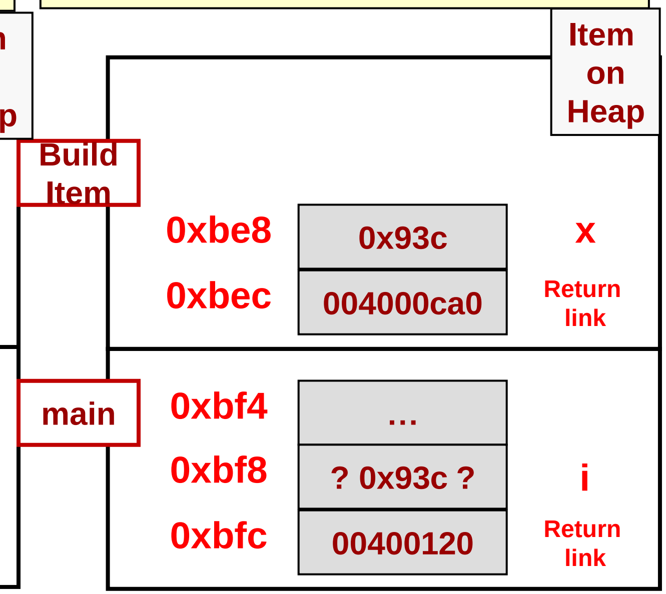
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item &i = buildItem();
 // access i's data
}
```

ex8





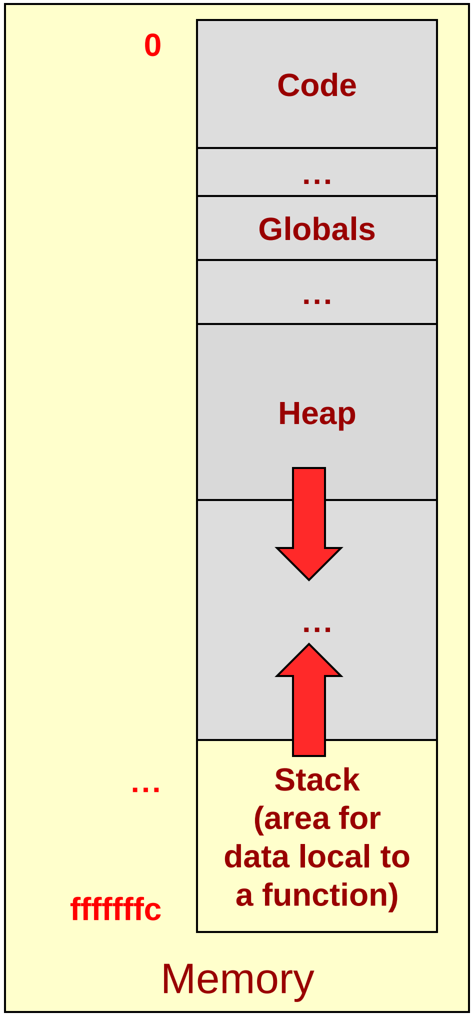


PRE-SUMMER 2021 BACKGROUND

VARIABLES & SCOPE

A Program View of RAM/Memory

- Code usually sits at low addresses
- Global variables somewhere after code
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.
- Heap: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
- Heap grows downward, stack grows upward…
  - In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error



Variables and Static Allocation

- Every variable/object in a computer has a:
  - Name (by which *programmer* references it)
  - Address (by which *computer* references it)
  - Value
- Let's draw these as boxes
- Every variable/object has **scope** (its lifetime and visibility to other code)
- Automatic/Local Scope
  - {…} of a function, loop, or if
  - Lives on the stack
  - Dies/Deallocated when the '}' is reached
- Let's draw these as nested container boxes

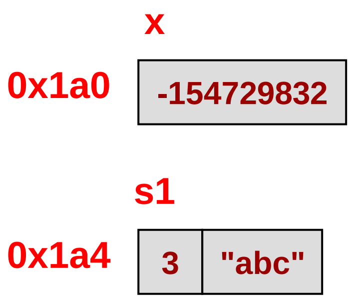
Code

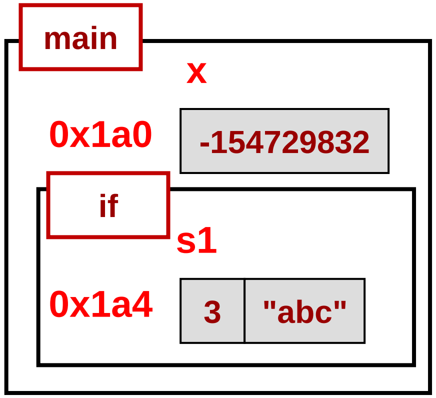
```
int x;

string s1("abc");
```

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

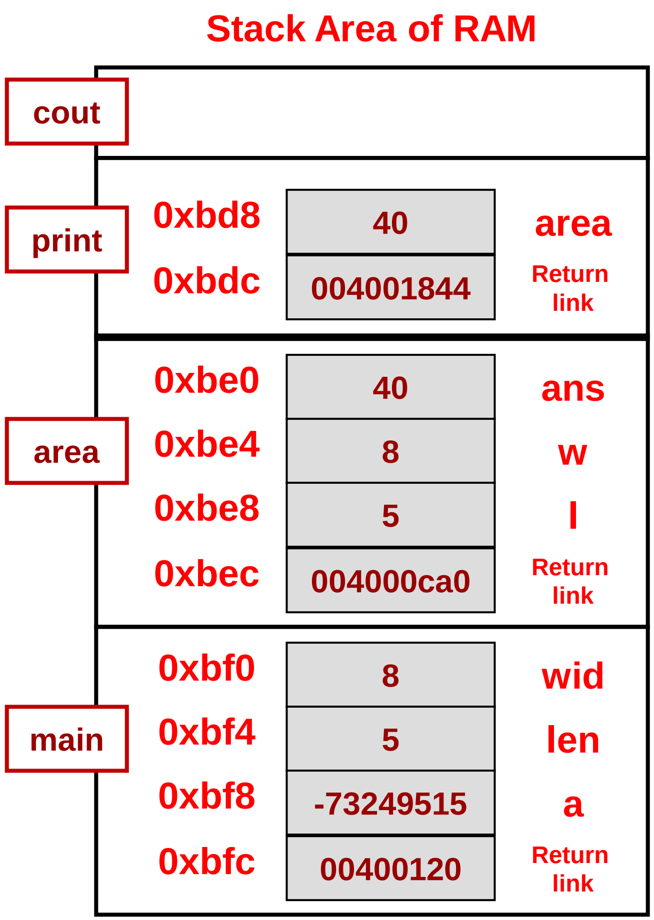
Computer





Automatic/Local Variables

- Variables declared inside {…} are allocated on the stack
- This includes functions




```
// 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

Kinds of References

Pointers

- A variable (like any other) which occupies memory and stores an address of another variable and can be updated (like any other variable) to store a new address to some other variable
- Declared with the `type*` syntax (e.g. `int*`, `char*`, `Item*`)

C++ Reference Variable

- A special variable that simply gives a second (or third, or fourth) name to an already-declared variable
- Declared with the `type&` syntax (e.g. `int&`, `string&`, `Item&`)
- Does not occupy any memory (just tells the compiler to allow another name to reference some other variable)

Important Note: When we use the general term "reference" as in "pass-by-reference" we can use EITHER **pointers** **OR** **C++ Reference Variables.**
Lets' take a look at each…

Review of Pointers in C/C++

- Pointer (type *)
  - Really just the 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)**
    - `*(&object)` => object [i.e. * and & are inverse operators of each other]
- Example: Indicate what each line prints or what variable is modified. Use **NA** for any invalid operation.

```
int* p, *q;
int i, j;

i = 5; j = 10;
p = &i;
cout << p << endl;
cout << *p << endl;
*p = j;
*q = *p;
q = p;
```

0xbe0		p
0xbe4		q
0xbe8	5	i
0xbec	10	j

Pointer Notes

- **NULL** (defined in <cstdlib>) or now **`nullptr`** (in C++11) are keywords for values you can assign to a pointer when it doesn't point to anything
  - NULL is effectively the value 0 so you can write:

```
int* p = NULL;
if( p )
{ /* will never get to this code */ }
```

  - To use **`nullptr`** compile with the C++11 version:

```
$ g++ -std=c++11 -g -o test test.cpp
```

- An uninitialized pointer is a pointer waiting to cause a SEGFAULT
- Beware of SEGFAULTS! What are they and what causes them?
- What tool can help find what is causing SEGFAULTS?

Check Yourself

- Consider these declarations:
  - `int k, x[3] = {5, 7, 9};`
  - `int *myptr = x;`
  - `int **ourptr = &myptr;`
- Indicate the formal type that each expression evaluates to (i.e. `int`, `int *`, `int **`)

To figure out the type of data a pointer expression will yield…

- **Each * in the expression cancels a * from the variable type.**
- **Each & in the expression adds a * to the variable type.**

Orig. Type	Expr	Yields
`myptr = int*`	`*myptr`	`int`
`ourptr = int**`	`**ourptr`	`int`
	`*ourptr`	`int*`
`k = int`	`&k`	`int*`
	`&myptr`	`int**`

Expression	Type
`&x[0]`	
`x`	
`myptr`	
`*myptr`	
`(*ourptr) + 1`	
`myptr + 2`	
`&ourptr`	

Using C++ References

- Reference type (`type &`) creates an alias (another name) the programmer/compiler can use for some other variable
  – Is **NOT** another variable; does **NOT** require memory
- "Syntactic sugar" (i.e. make programmer's life easy) to avoid using pointers
- 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

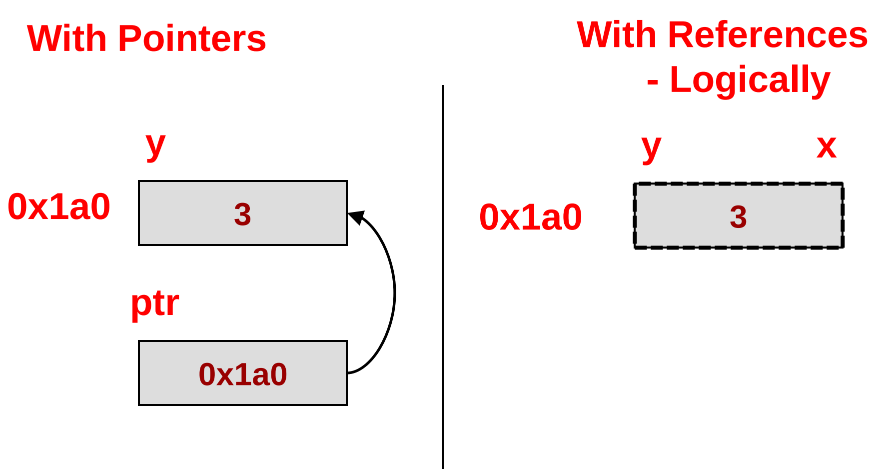

 int &x = y; // reference
             // declaration
  // We've not copied y into x.
  // Rather, we've created an alias.
  // What we do to x happens to y.
  // Now x can never reference
  //   any other int…only y!


  x++;    // y just got incr.

  cout << y << endl;

  int &z;      // NO! must assign

  int w = 5;
  x = w;  // doesn't make x
          // reference w...copies
          // w into y;
  return 0;
}
```



References in C/C++

- Declare a reference to an object as `type&` (e.g. `int&`)
- Must be initialized at declaration time (i.e. can't declare a reference variable if without indicating what object you want to reference)
  - Logically, C++ reference types DON'T consume memory…they are just an alias (another name) for the variable they reference
  - Physically, it *may* be implemented as a pointer to the referenced object but that is NOT your concern
- Cannot change what the reference variable refers to once initialized
- Most common usage is for parameter passing (see next slide)

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 (likely its really passing a pointer behind the scenes but now you don't have to dereference everything)

```
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;
}

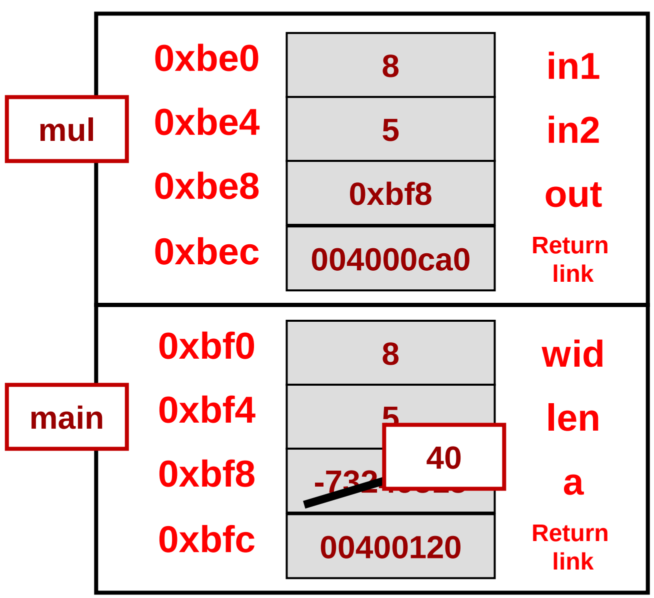
void swapit(int &x, int &y)
{
   int temp;
   temp = x;
   x = y;
   y = temp;
}
```

program output: x=7,y=5

Correct Usage of Pointers

- Commonly functions will take some inputs and produce some outputs
  - We'll use a simple 'multiply' function for now even though we can easily compute this without a function
  - We could use the return value from the function but let's practice with pointers
- Can use a pointer to have a function modify the variable of another

Stack Area of RAM




```
// Computes the product of in1 & in2
int mul1(int in1, int in2);
void mul2(int in1, int in2, int* out);


int main()
{
  int wid = 8, len = 5, a;
  mul2(wid,len,&a);
  cout << "Ans. is " << a << endl;
  return 0;
}

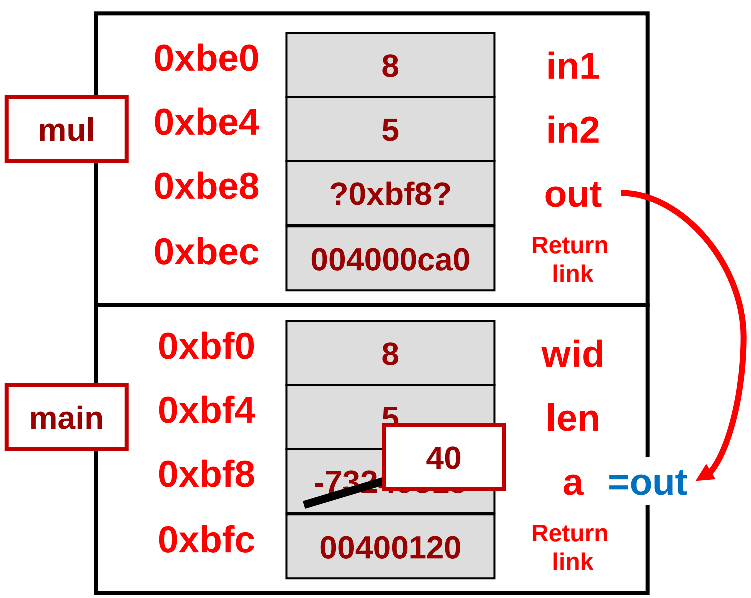
int mul1(int in1, int in2)
{
  return in1 * in2;
}

void mul2(int in1, int in2, int* out)
{
  *out = in1 * in2;
}
```

Now with C++ References

- We can pass using C++ reference
- The reference 'out' is just an alias for 'a' back in main
  - In memory, it might actually be a pointer, but you don't have to dereference (the kind of stuff you have to do with pointers)

Stack Area of RAM




```
// Computes the product of in1 & in2
void mul(int in1, int in2, int& out);

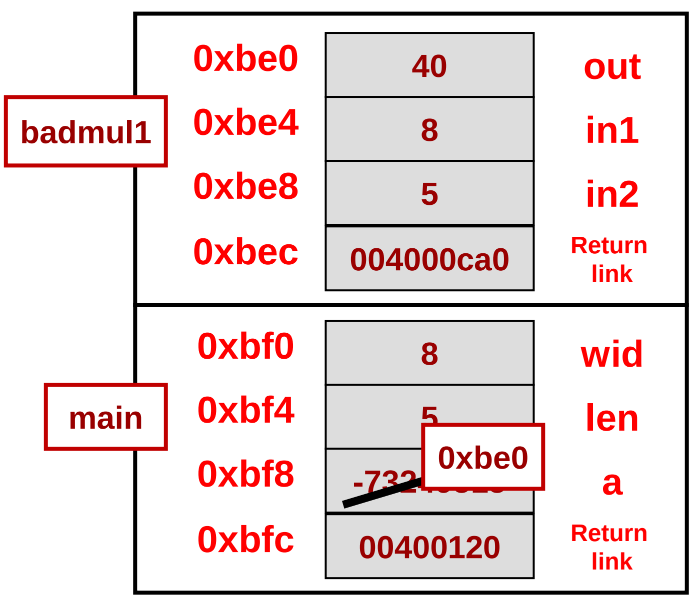
int main()
{
  int wid = 8, len = 5, a;
  mul(wid,len,a);
  cout << "Ans. is " << a << endl;
  return 0;
}

void mul(int in1, int in2, int& out)
{
  out = in1 * in2;
}
```

Misuse of Pointers/References

- Make sure you don't return a pointer or reference to a dead variable
- You might get lucky and find that old value still there, but likely you won't

Stack Area of RAM




```
// Computes the product of in1 & in2
int* badmul1(int in1, int in2);
int& badmul2(int in1, int in2);

int main()
{
  int wid = 8, len = 5;
  int *a = badmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  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;
}

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

Pass-by-Value vs. -Reference

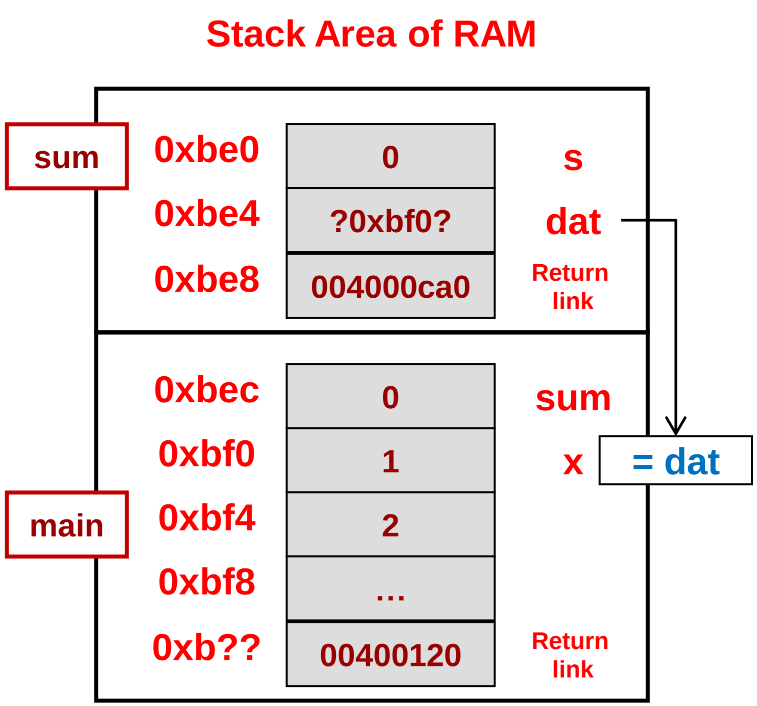
- Arguments are said to be:
  - Passed-by-value: A copy is made from one function and given to the other
  - Passed-by-reference (i.e. pointer or C++ reference): A reference (really the address) to the variable is passed to the other function

Pass-by-Value Benefits	Pass-by-Reference Benefits
+ Protects the variable in the caller since a copy is made (any modification doesn't affect the original)	+ Allows another function to modify the value of variable in the caller + Saves time vs. copying

- Care needs to be taken when choosing between the options

Pass by Reference

- Notice no copy of x need be made since we pass it to sum() by reference
  - Notice that likely the computer passes the address to sum() but you should just think of **dat** as an alias for **x**
  - The **const** keyword tells the compiler to double check that we don't modify the vector (giving the safety of pass-by-value but the performance of pass-by reference)




```
// 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;
}
```

Pointers vs. References Summary

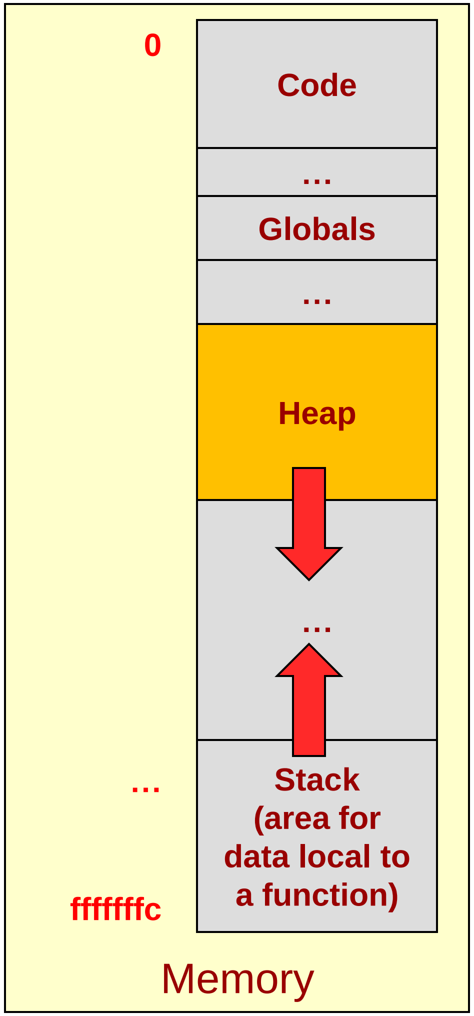
- How to tell references and pointers apart
  - Check if you see the '&' or '*' in a type declaration or expression

	With a Type	In an Expression
&	Indicates a C++ Reference Var (int &val, vector<int> &vec)	Address-of yields a pointer to the object Adds a * to the type of variable
*	Declares a pointer type variable (int *valptr = &val, vector<int> *vecptr = &vec)	De-Reference (Value @ address) Cancels a * from the type of variable

DYNAMIC ALLOCATION

Dynamic Memory & the Heap

- Code usually sits at low addresses
- Global variables somewhere after code
- System stack (memory for each function instance that is alive)
  - Local variables
  - Return link (where to return)
  - etc.
- Heap: Area of memory that can be allocated and de-allocated during program execution (i.e. dynamically at run-time) based on the needs of the program
- Heap grows downward, stack grows upward…
  - In rare cases of large memory usage, they could collide and cause your program to fail or generate an exception/error



Motivation

Automatic/Local Variables

- Deallocated (die) when they go out of scope
- As a general rule of thumb, they must be statically sized (size is a constant known at compile time)
  - `int data[100];`

Dynamic Allocation

- Persist until explicitly deallocated by the program (via 'delete')
  - Data lives indefinitely
- Can be sized at run-time
  - ```
    int size;
    cin >> size;
    int *data = new int[size];
    ```

(These are the 2 primary reasons to use dynamic allocation.)
    ```

C Dynamic Memory Allocation

- `void* malloc(int num_bytes)` function in stdlib.h
  - Allocates the number of bytes requested and returns a pointer to the block of memory
  - Use sizeof(*type*) macro rather than hardcoding 4 since the size of an int may change in the future or on another system
- `free(void * ptr)` function
  - Given the pointer to the (starting location of the) block of memory, free returns it to the system for re-use by subsequent malloc calls

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char *argv[])
{
  int num;

  cout << "How many students?" << endl;
  cin >> num;

  int *scores = (int*) malloc( num*sizeof(int) );
  // can now access scores[0] .. scores[num-1];

  free(scores);
  return 0;
}
```

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];`
  - can obviously use a variable to indicate array size
  - returns a pointer of the appropriate type
    - if you ask for a new int, you get an int * in return
    - if you ask for an new array (new int[10]), you get an int * in return
- 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;`

Dynamic Memory Allocation

```
int main(int argc, char *argv[])
{
  int num;

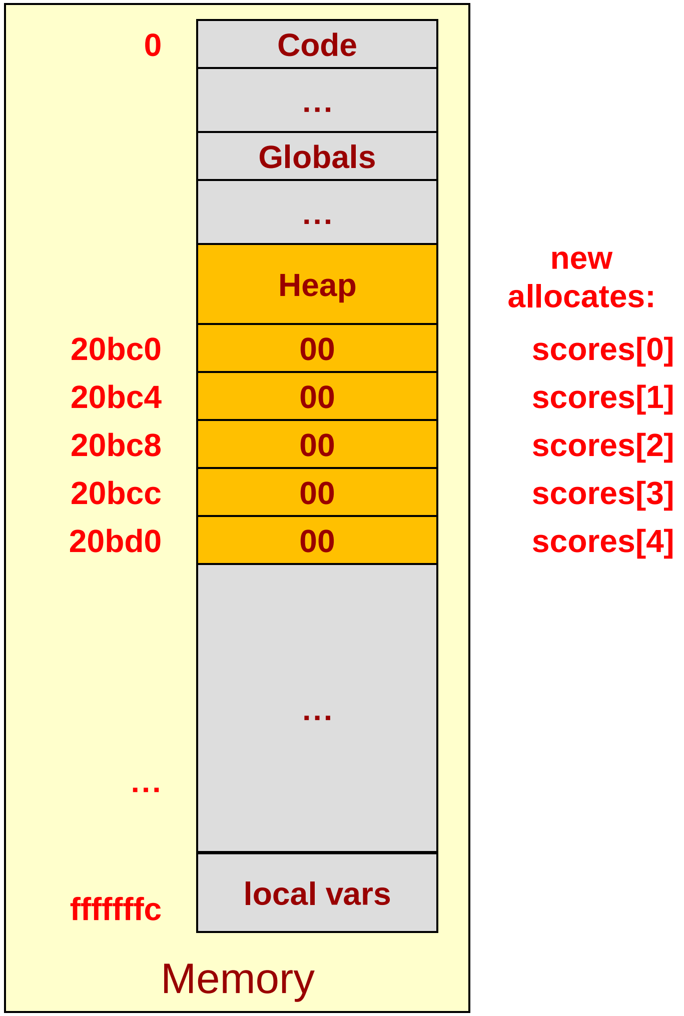
  cout << "How many students?" << endl;
  cin >> num;

  int *scores = new int[num];
  // can now access scores[0] .. scores[num-1];
  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int num;

  cout << "How many students?" << endl;
  cin >> num;

  int *scores = new int[num];
  // can now access scores[0] .. scores[num-1];
  delete [] scores
  return 0;
}
```



Fill in the Blanks

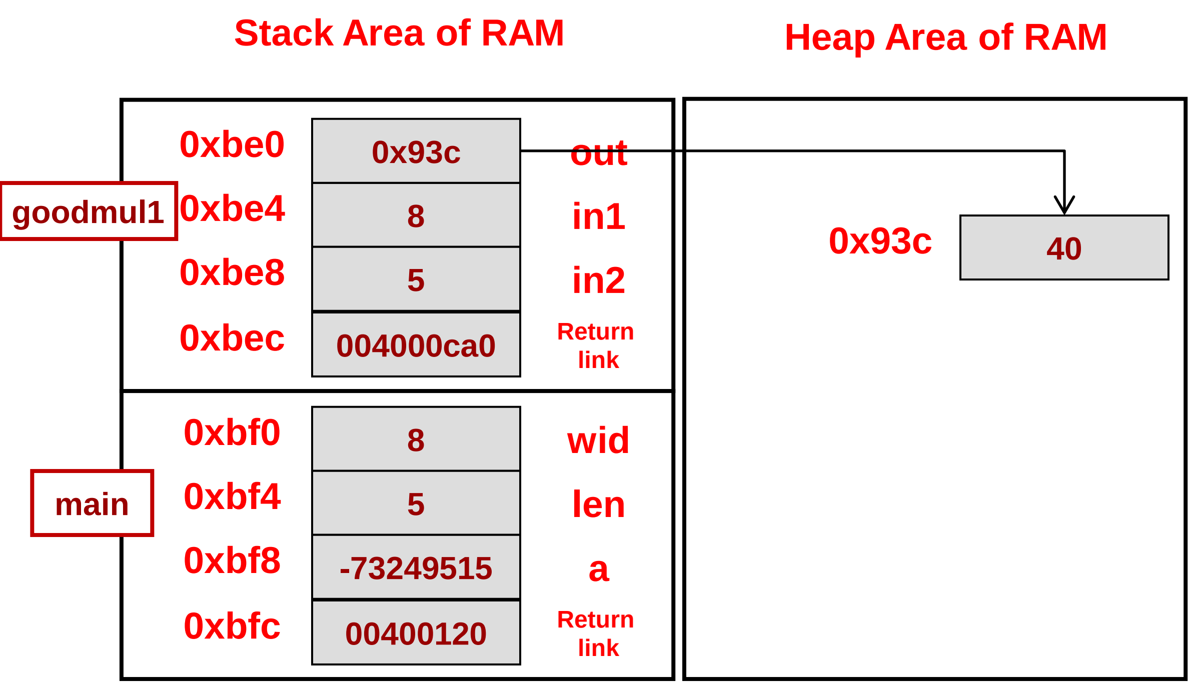
- _________ data = new int;
- _________ data = new char;
- _________ data = new char[100];
- _________ data = new char*[20];
- _________ data = new vector<string>;
- _________ data = new Student;

Fill in the Blanks

- __________ data = new int;
  - int*
- __________ data = new char;
  - char*
- __________ data = new char[100];
  - char*
- __________ data = new char*[20];
  - char**
- __________ data = new vector<string>;
  - vector<string>*
- __________ data = new Student;
  - Student*

Dynamic Allocation

- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw the operation of **goodmul1()**




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

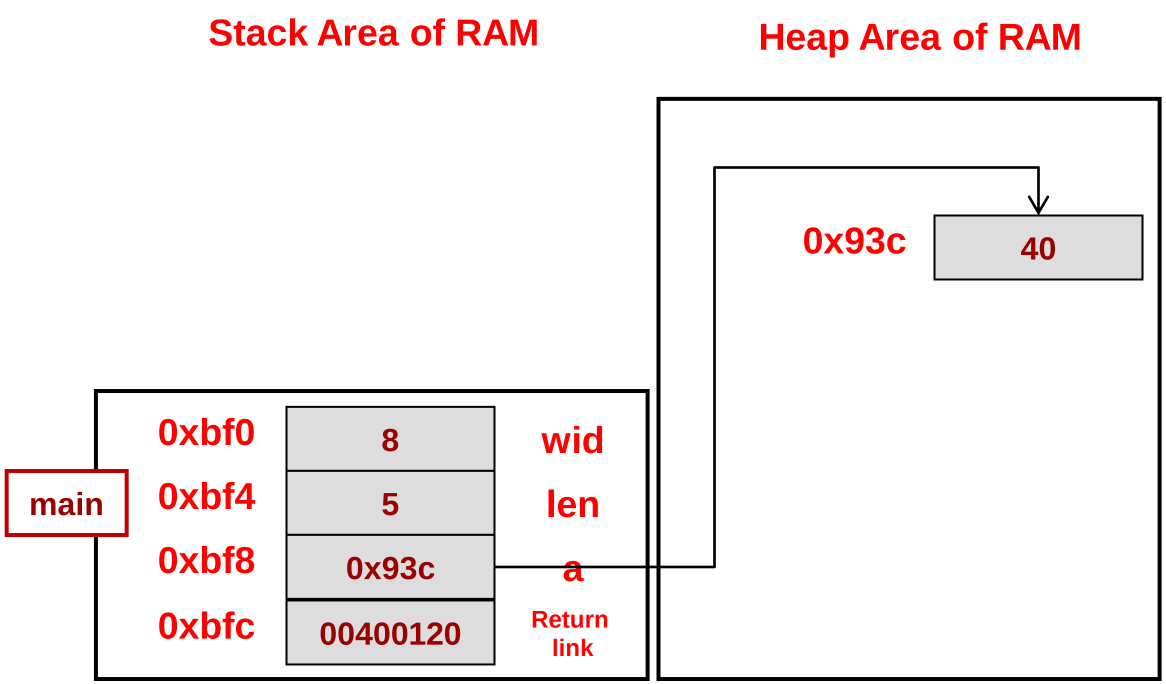
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  delete a;
  return 0;
}

// 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

- When goodmul1() exits, the out pointer goes out of scope
- Thus we need to return the pointer or save it somewhere so that there is a record of our allocation, otherwise we will have a leak




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

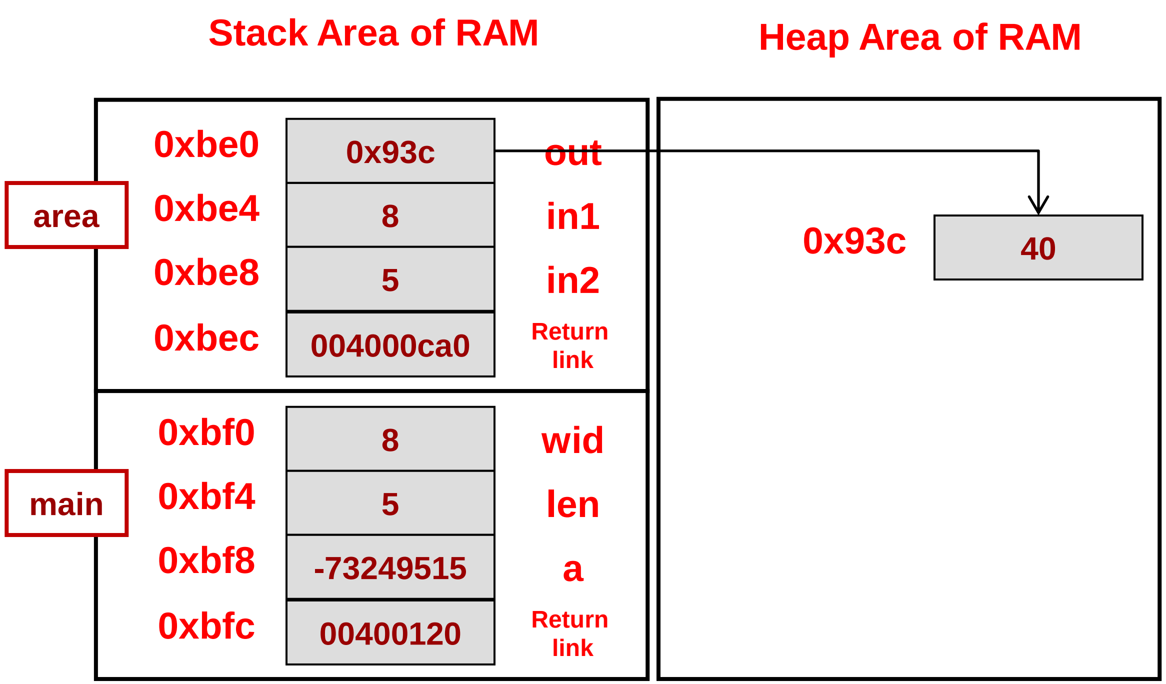
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  delete a;
  return 0;
}

// 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 – Q1

- What happens if we comment the 'delete a' line?




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

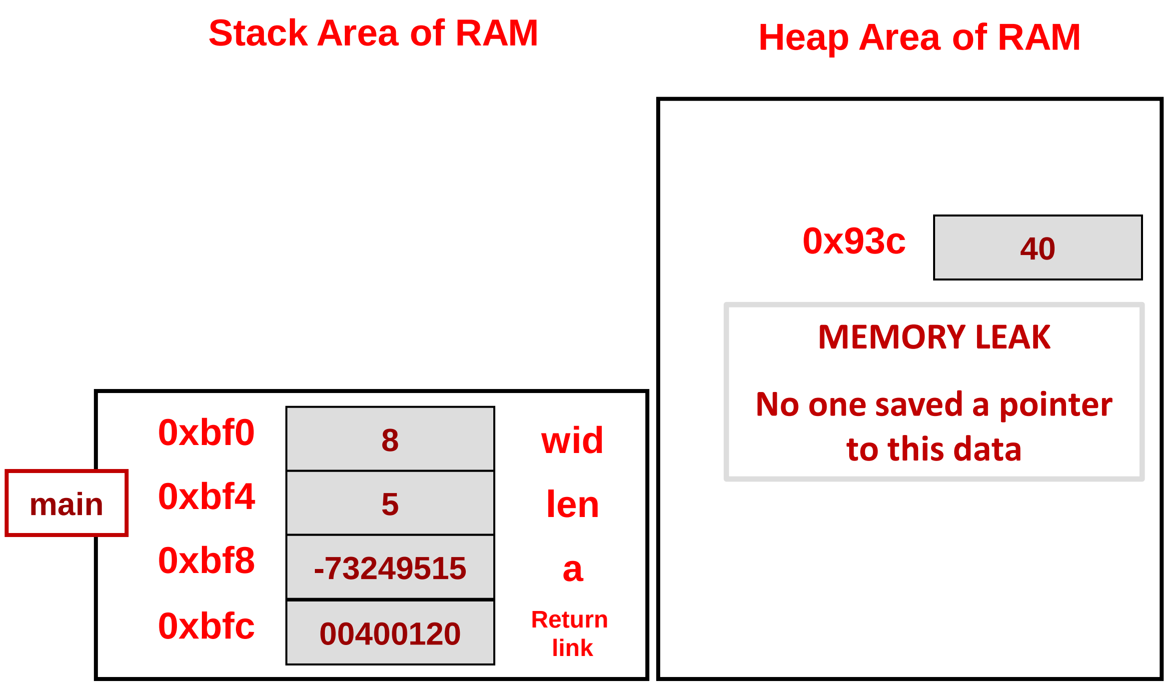
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
 // delete a;
  return 0;
}

// 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 – A1

- What happens if we comment the 'delete a' line?
  - **Memory LEAK!!**




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

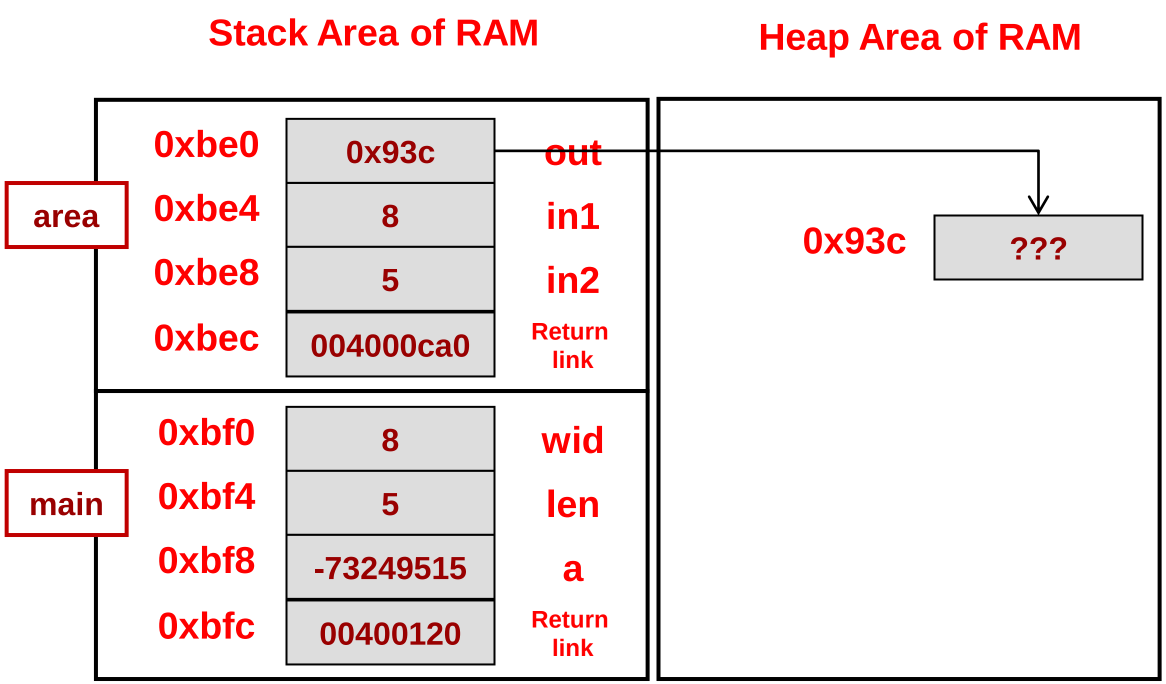
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
 // delete a;
  return 0;
}

// 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 – Q2

- What happens if we overwrite the only pointer to a dynamically allocated variable/object?




```
// Computes the product of in1 & 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;
}

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

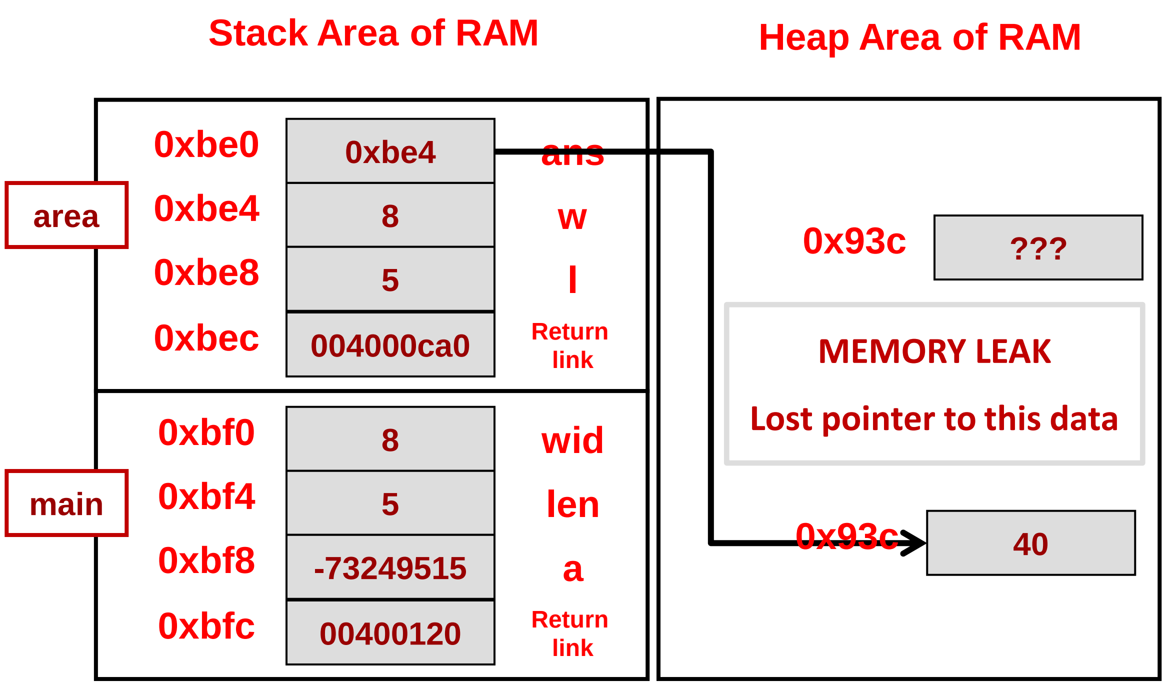
Dynamic Allocation – A2

- What happens if we overwrite the only pointer to a dynamically allocated variable/object?
  - A memory leak
- Be sure you keep a pointer around somewhere otherwise you'll have a memory leak!

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

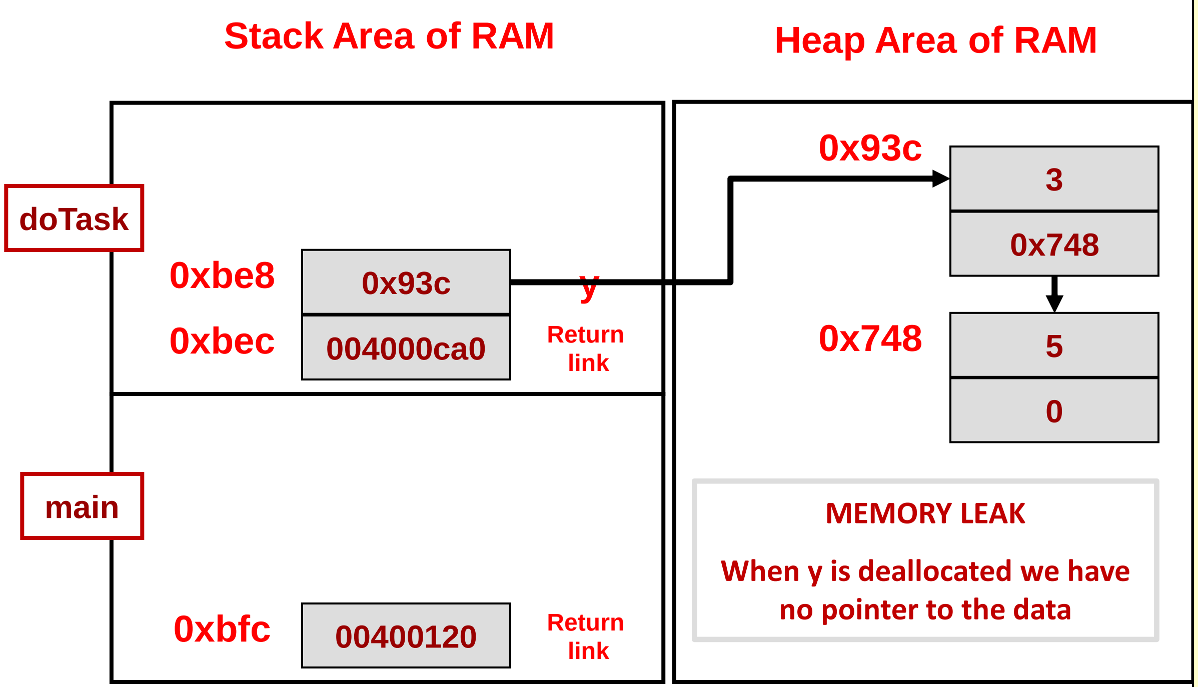
int main()
{
  int wid = 8, len = 5;
  int *a = goodmul1(wid,len);
  cout << "Ans. is " << *a << endl;
  delete a;
  return 0;
}

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




```
// Computes rectangle area,
//  prints it, & returns it
struct Item {
  int val;  Item* next;
};
class LinkedList {
  public:
   // create a new item
   // in the list
   void push_back(int v);
  private:
   Item* head;
};

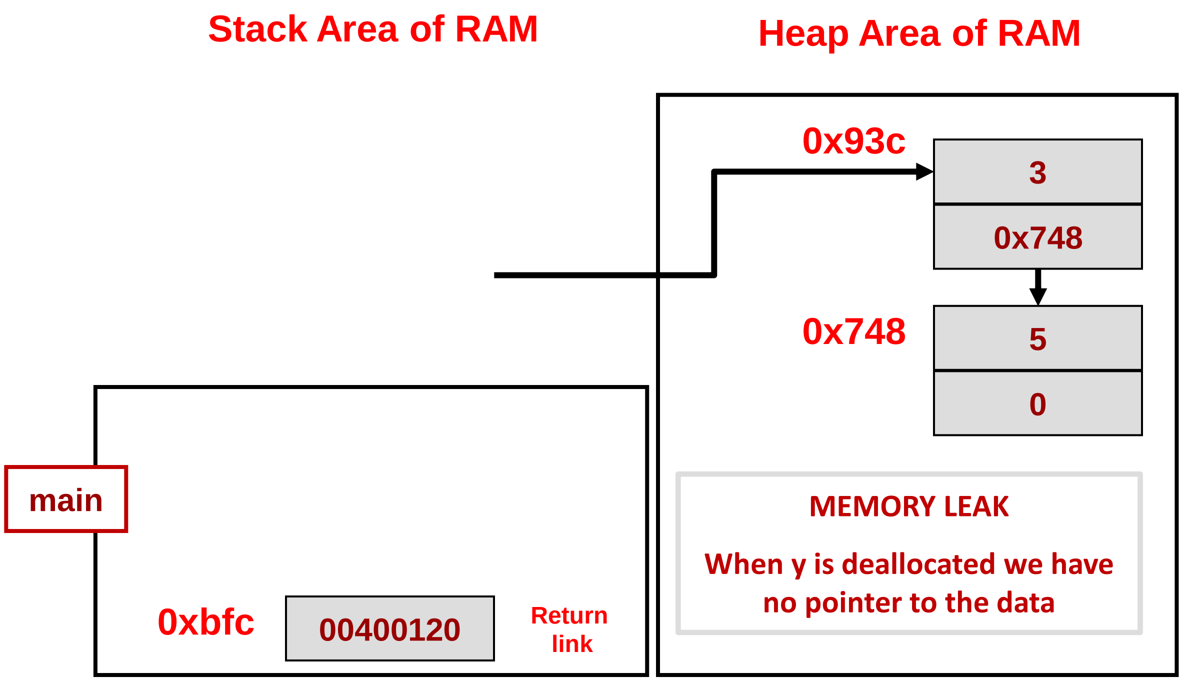
int main()
{
  doTask();
}

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

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

An Appropriate Destructor Will Help Solve This




```
// Computes rectangle area,
//  prints it, & returns it
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 */
}
```

If time allows

PRACTICE ACTIVITY

Object Assignment

- Assigning one struct or class object to another will cause an element by element copy of the source data destination struct or class

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

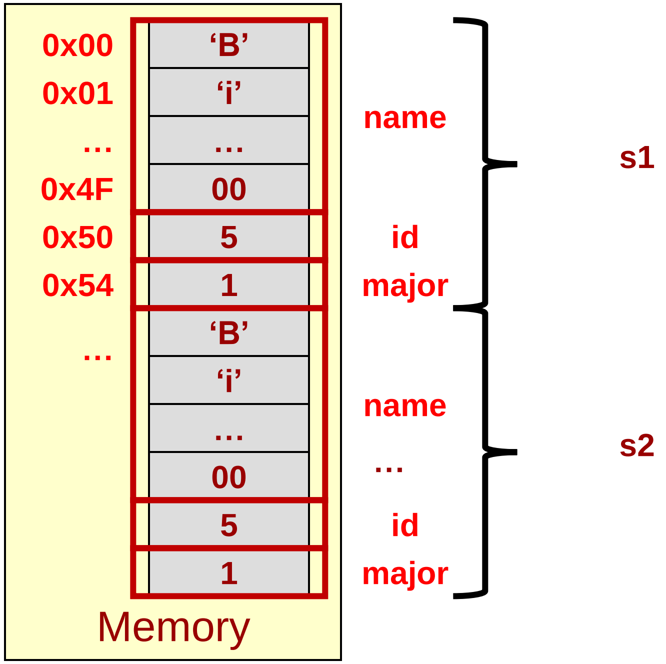
enum {CS, CECS };

struct student {
  char name[80];
  int id;
  int major;
};

int main(int argc, char *argv[])
{
  student s1;
  strncpy(s1.name,"Bill",80);
  s1.id = 5; s1.major = CS;

  student s2 = s1;

  return 0;
}
```



Memory Allocation Tips

- Take care when returning a pointer or reference that the object being referenced will persist beyond the end of a function
- Take care when assigning a returned referenced object to another variable…you are making a copy
- Try the examples yourself
  - $ wget http://ee.usc.edu/~redekopp/cs104/memref.cpp

Understanding Memory Allocation

There are no syntax errors. Which of these can correctly build an Item and then have main() safely access its data

```
class Item
{ public:
  Item(int w, string y);
};
Item buildItem()
{ Item x(4, "hi");
  return x;
}

int main()
{ Item i = buildItem();
  // access i's data.

}
```

ex1

```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item x(4, "hi");
  return x;
}

int main()
{ Item& i = buildItem();
  // access i's data
}
```

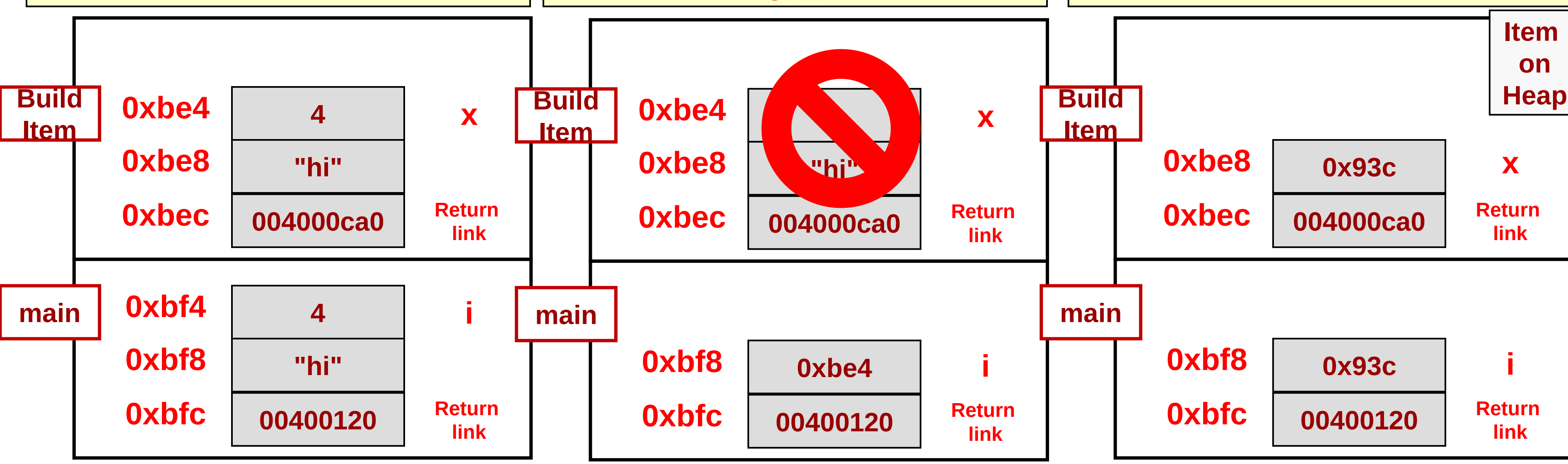
ex2

```
class Item
{ public:
  Item(int w, string y);

};
Item* buildItem()
{ Item* x = new Item(4,"hi");
  return x;
}

int main()
{ Item *i = buildItem();
  // access i's data
}
```

ex3



USC Viterbi School of Engineering

Understanding Memory Allocation

There are no syntax errors. Which of these can correctly build an Item and then have main() safely access its data

```
class Item
{ public:
  Item(int w, string y);
};
Item* buildItem()
{ Item x(4, "hi");
  return &x;
}

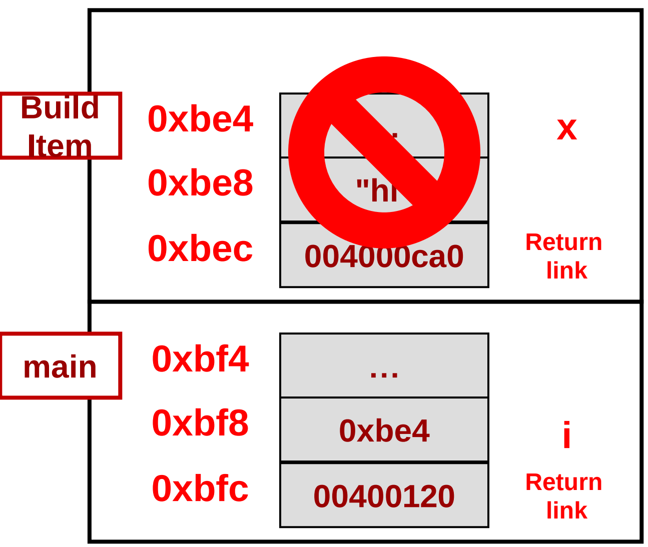
int main()
{ Item *i = buildItem();
  // access i's data
}
```

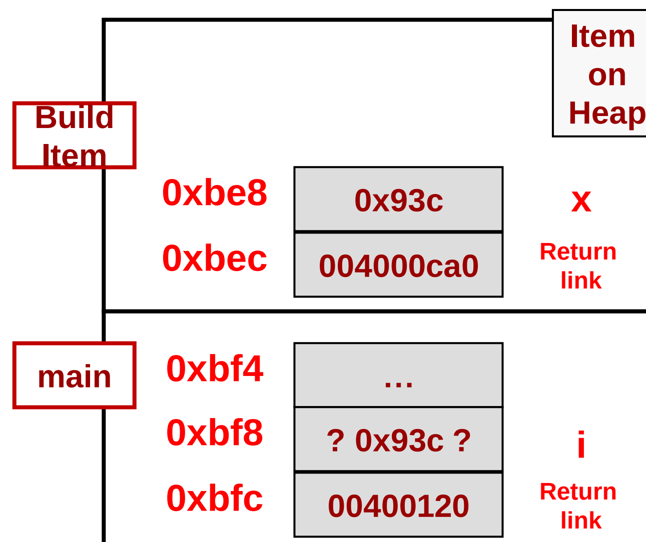
ex4

```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item& i = buildItem();
 // access i's data
}
```

ex5



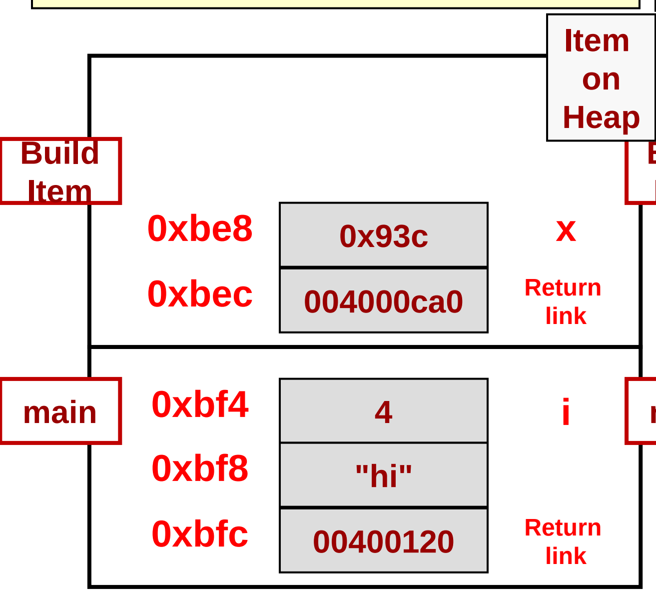


Understanding Memory Allocation

```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item i = buildItem();
  // access i's data.
}
```

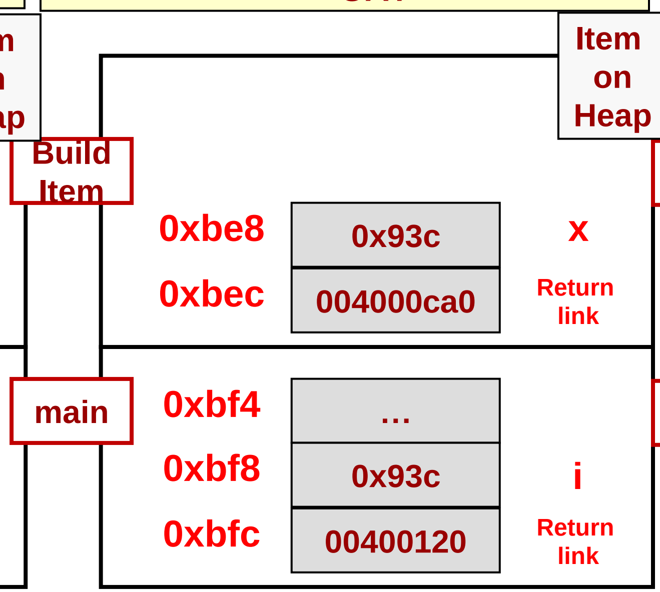
ex6




```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item *i = &(buildItem());
  // access i's data.
}
```

ex7

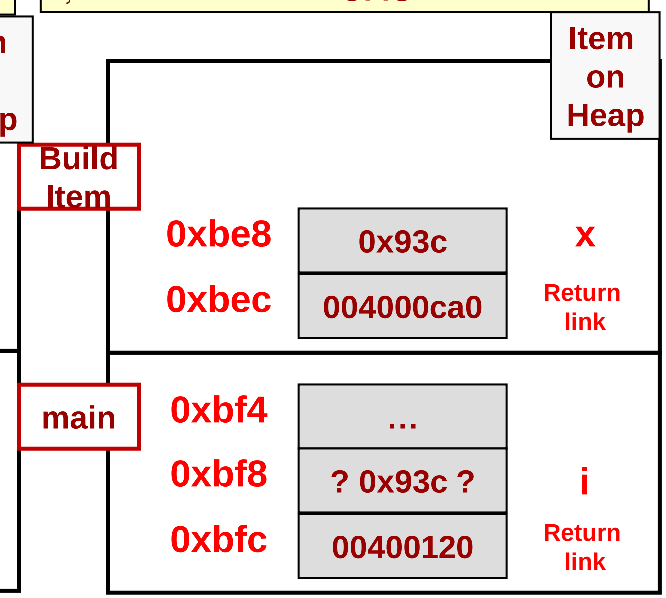



```
class Item
{ public:
  Item(int w, string y);

};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item &i = buildItem();
 // access i's data
}
```

ex8



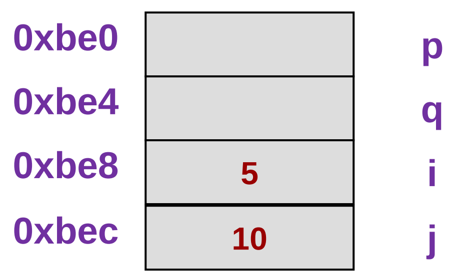
SOLUTIONS

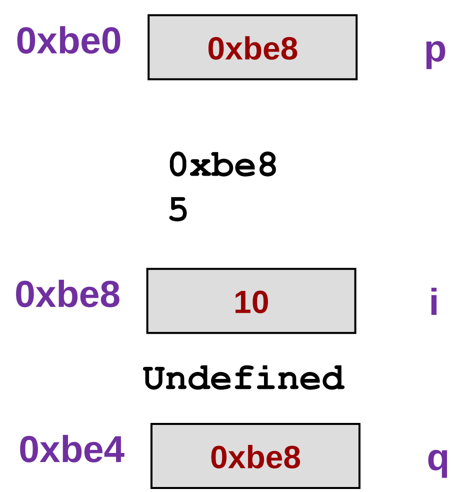
Review of Pointers in C/C++

- Pointer (type *)
  - Really just the 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)**
    - `*(&object)` => object [i.e. * and & are inverse operators of each other]
- Example: Indicate what each line prints or what variable is modified. Use **NA** for any invalid operation.

```
int* p, *q;
int i, j;

i = 5; j = 10;
p = &i;
cout << p << endl;
cout << *p << endl;
*p = j;
*q = *p;
q = p;
```





Check Yourself

- Consider these declarations:
  - `int k, x[3] = {5, 7, 9};`
  - `int *myptr = x;`
  - `int **ourptr = &myptr;`
- Indicate the formal type that each expression evaluates to (i.e. `int`, `int *`, `int **`)

To figure out the type of data a pointer expression will yield…
- **Each * in the expression cancels a * from the variable type.**
- **Each & in the expression adds a * to the variable type.**

Orig. Type	Expr	Yields
myptr = int*	*myptr	int
ourptr = int**	**ourptr	int
	*ourptr	int*
k = int	&k	int*
	&myptr	int**

Expression	Type
&x[0]	int*
x	int*
myptr	int*
*myptr	int
(*ourptr) + 1	int*
myptr + 2	int*
&ourptr	int***

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 (likely its really passing a pointer behind the scenes but now you don't have to dereference everything)

```
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

Understanding Memory Allocation

There are no syntax errors. Which of these can correctly build an Item and then have main() safely access its data

```
class Item
{ public:
   Item(int w, string y);
};
Item buildItem()
{ Item x(4, "hi");
   return x;
}

int main()
{ Item i = buildItem();
   // access i's data.
}
```

ex1

```
class Item
{ public:
   Item(int w, string y);
};
Item& buildItem()
{ Item x(4, "hi");
   return x;
}

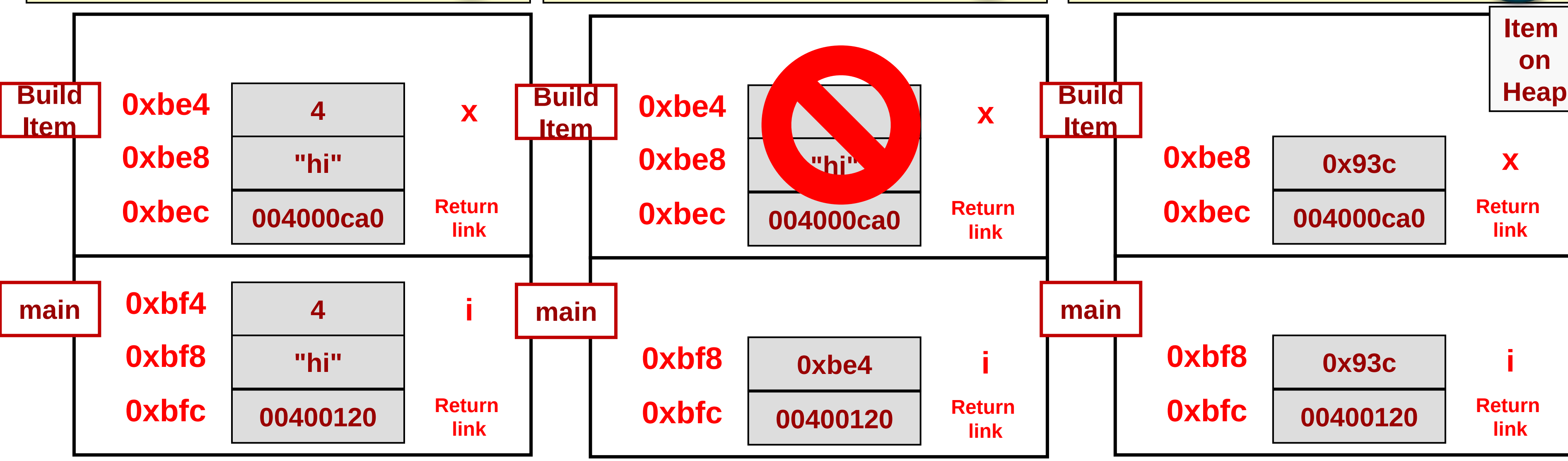
int main()
{ Item& i = buildItem();
   // access i's data
}
```

ex2

```
class Item
{ public:
   Item(int w, string y);
};
Item* buildItem()
{ Item* x = new Item(4,"hi");
   return x;
}

int main()
{ Item *i = buildItem();
   // access i's data
}
```

ex3



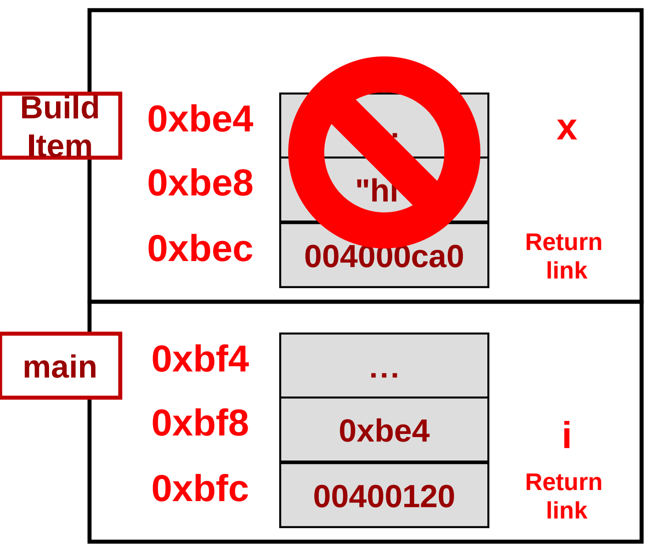
Understanding Memory Allocation

There are no syntax errors. Which of these can correctly build an Item and then have main() safely access its data

```
class Item
{ public:
  Item(int w, string y);
};
Item* buildItem()
{ Item x(4, "hi");
  return &x;
}

int main()
{ Item *i = buildItem();
  // access i's data
}
```

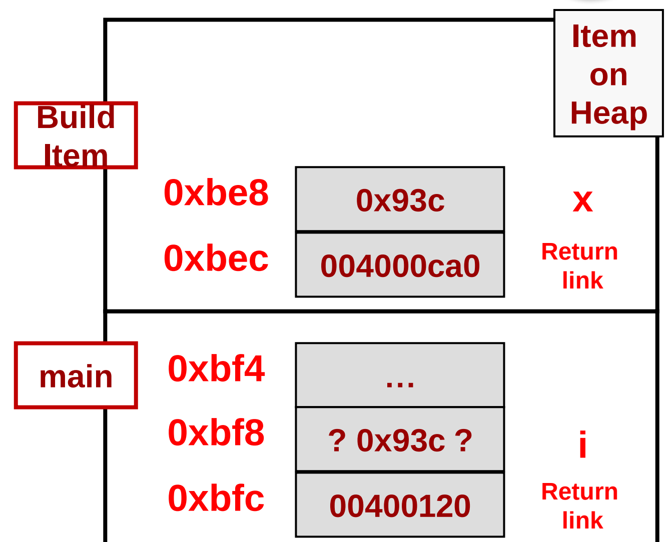
ex4




```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
  return *x;
}

int main()
{ Item& i = buildItem();
 // access i's data
}
```

ex5



Understanding Memory Allocation

ex6 (✓ ✗)

```
class Item
{ public:
   Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
   return *x;
}

int main()
{ Item i = buildItem();
   // access i's data.
}
```

Item on Heap

Build Item	0xbe8	0x93c	x
	0xbec	004000ca0	Return link
main	0xbf4	4	i
	0xbf8	"hi"	
	0xbfc	00400120	Return link

ex7 (✓)

```
class Item
{ public:
   Item(int w, string y);
};
Item& buildItem()
{ Item* x = new Item(4,"hi");
   return *x;
}

int main()
{ Item *i = &(buildItem());
   // access i's data.
```

Item on Heap

Build Item	0xbe8	0x93c	x
	0xbec	004000ca0	Return link
main	0xbf4	…	
	0xbf8	0x93c	i
	0xbfc	00400120	Return link

ex8 (✓)

```
class Item
{ public:
   Item(int w, string y);

};
Item& buildItem()
{ Item* x = new Item(4,"hi");
   return *x;
}

int main()
{ Item &i = buildItem();
   // access i's data
}
```

Item on Heap

Build Item	0xbe8	0x93c	x
	0xbec	004000ca0	Return link
main	0xbf4	…	
	0xbf8	? 0x93c ?	i
	0xbfc	00400120	Return link