CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

Administrative Issues

- Preparation
  - Basic if, while, for constructs
  - Arrays, linked-lists
  - Structs, classes
  - **Dynamic memory allocation and pointers**
  - **Recursion**
- Syllabus
  - http://bits.usc.edu/cs104
- Expectations
  - I'll give you my best, you give me yours…
    - Attendance, participation, asking questions, academic integrity, take an interest
  - Treat CS104 right!
  - Let's make this fun

More Helpful Links

- Remedial modules
  - http://ee.usc.edu/~redekopp/csmodules.html
- Class website
  - http://bits.usc.edu/cs104

An Opening Example

- Consider a paper phonebook
  - Stores names of people and their phone numbers
- What operations do we perform with this data
  - You: Lookup/search
  - Phone Company: Add, Remove
- How is the data stored and ordered and why?
  - Sorted by name to make lookup faster…
  - How fast? That's for you to figure out…
- What if it was sorted by phone number or just random? What is the worst case number of records you'd have to look at to find a particular persons phone number?

Opening Example (cont.)

- Would it ever be reasonable to have the phonebook in a random or unsorted order?
  - What if the phonebook was for the residence of a town with only a few residents
  - What if there was a phonebook for Mayflies (life expectancy of 1-24 hours)
    - Might want to optimize for additions and removals
    - Plus, a mayfly doesn't have fingers to dial their phones so why would they even be trying to search the phonebook
- Main Point: **The best way to organize data depends on how it will be used**.
  - Frequent search
  - Frequent addition/removals
  - Addition/removal patterns (many at once or one at a time)

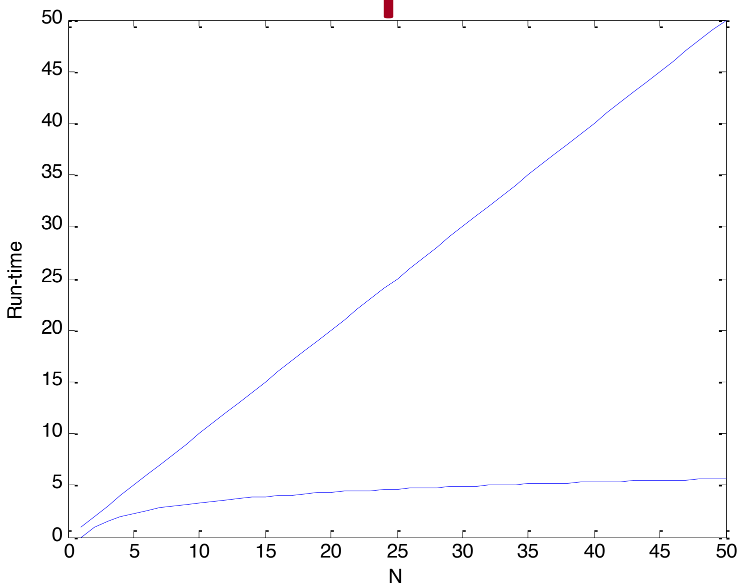
Why Data Structures Matter?

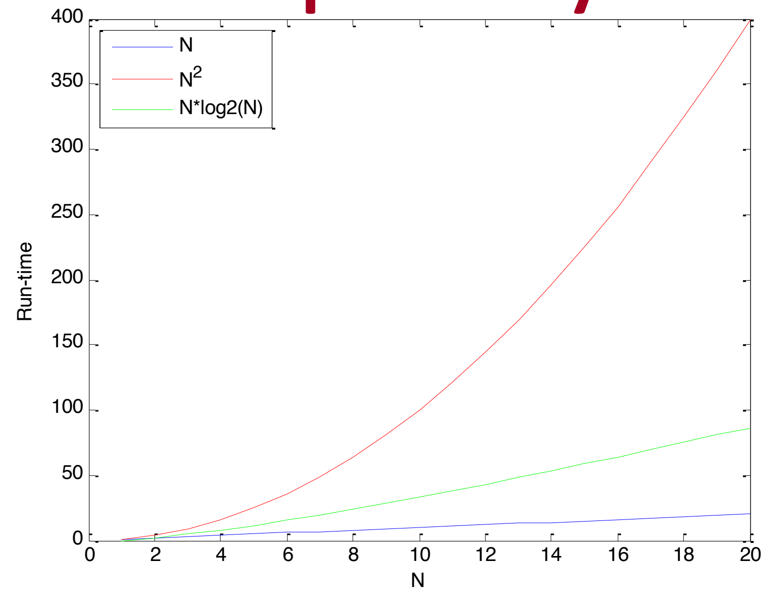
- Modern applications process vast amount of data
- Adding, removing, searching, and accessing are common operations
- Various data structures allow these operations to be completed with different time and storage requirements

Data Structure	Insert	Search	Get-Min
Unsorted List	O(1)	O(n)	O(n)
Balanced Binary Search Tree	O(lg n)	O(lg n)	O(lg n)
Heap	O(lg n)	O(n)	O(1)

Recall O(n) indicates that the actual run-time is bounded by some expression a*n for some $n > n_0$ (where a and n_0 are constants)

Importance of Complexity





N	O(1)	O($\log_2 n$)	O(n)	O($n*\log_2 n$)	O(n^2)	O(2^n)
2	1	1	2	2	4	4
20	1	4.3	20	86.4	400	1,048,576
200	1	7.6	200	1,528.8	40,000	1.60694E+60
2000	1	11.0	2000	21,931.6	4,000,000	#NUM!

Abstract Data Types

- Beginning programmers tend to focus on the code and less on the data and its organization
- More seasoned programmers focus first on
  - What data they have
  - How it should be organized
  - How it will be accessed
- An **abstract data type** describes what data is stored and what operations are to be performed
- A **data structure** is a specific way of storing the data implementing the operations
- Example **ADT**: List
  - Data: items of the same type in a particular order
  - Operations: insert, remove, get item at location, set item at location, find
- Example **data structures** implementing a List:
  - Linked list, array, etc.

Transition to Object-Oriented

- Object-oriented paradigm fits nicely with idea of ADTs
  - Just as ADTs focus on data and operations performed on it so objects combine data + functions
- Objects (C++ Classes) allows for more legible, modular, maintainable code units
- Suppose you and a friend are doing an electronic dictionary app. Your friend codes the dictionary internals and you code the user-interface.
  - You don't care how they implement it just that it supports the desired operations and is fast enough
  - **Abstraction**: Provides a simplified interface allowing you to reason about the higher level logic and not the low level dictionary ops.
  - **Encapsulation**: Shields inside from outside so that internals can be changed w/o affecting code using the object

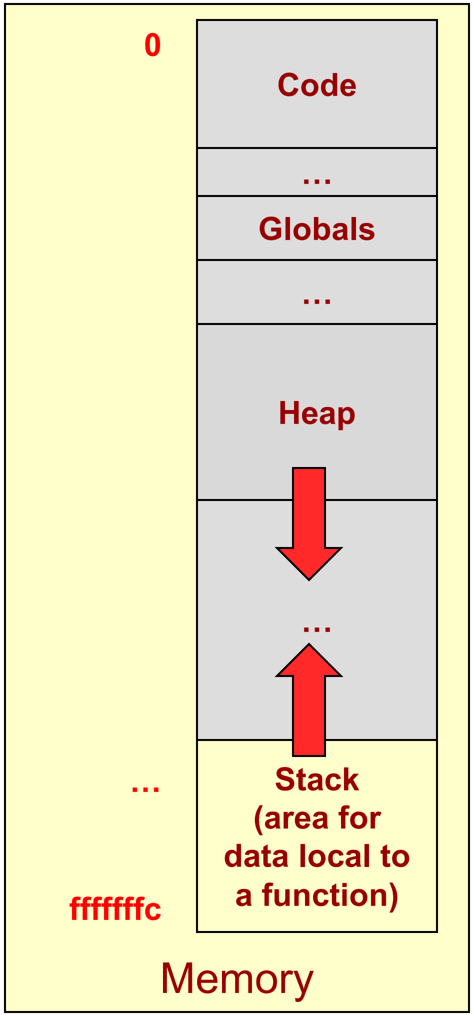
Course Goals

- Learn about good programming practice with object-oriented design
  - Learn good style and more advanced C++ topics such as templates, inheritance, polymorphism, etc.
- Learn basic and advanced techniques for implementing data structures and analyzing their efficiency
  - May require strong fundamentals including mathematical analysis
  - This is why we couple CS 104 and CS 170

MEMORY ALLOCATION REVIEW
VARIABLES & SCOPE

A Program View of 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



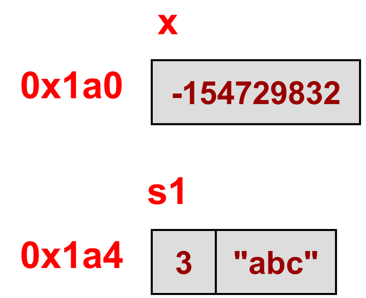
Variables and Static Allocation

- 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

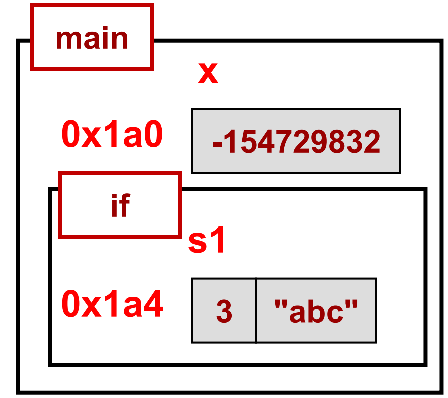
Code

```
int x;

string s1("abc");
```

Computer



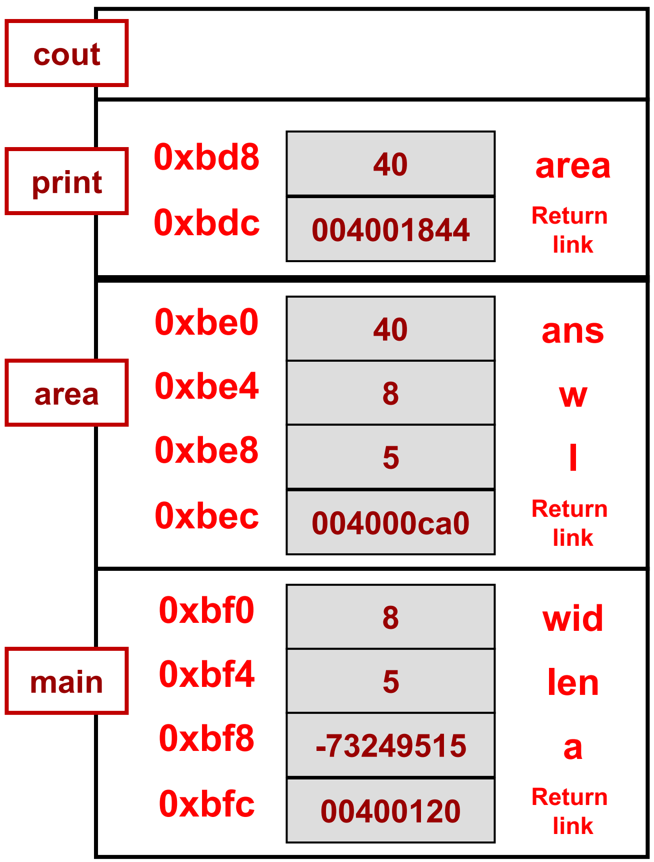

```
int main()
{
   int x;
   if( x ){
      string s1("abc");
   }
}
```



Automatic/Local Variables

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

Stack Area of RAM



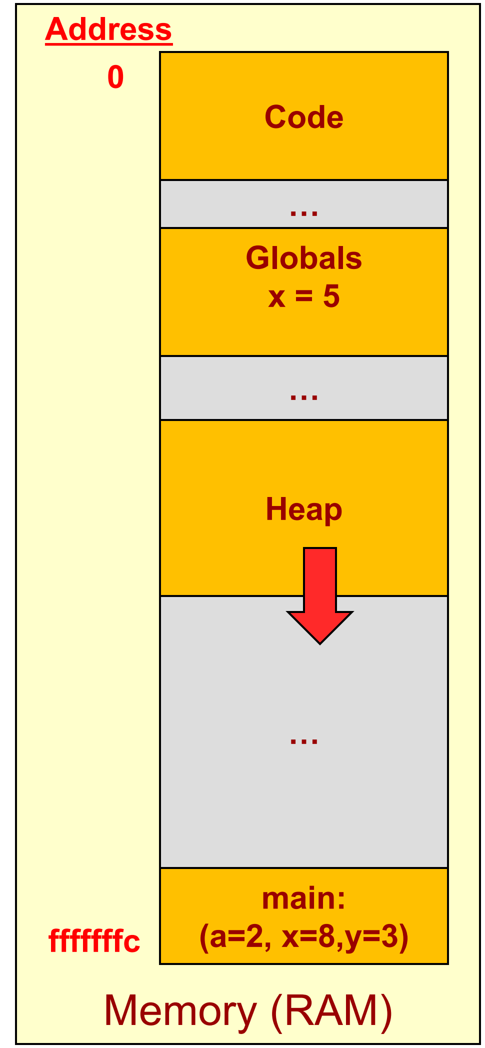

```
// Computes rectangle area,
//  prints it, & returns it
int area(int, int);
void print(int);

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;
}
```

USC Viterbi School of Engineering

Scope Example

- Globals live as long as the program is running
- Variables declared in a block { … } live as long as the block has not completed
  - { … } of a function
  - { … } of a loop, if statement, etc.
- When variables share the same name the closest declaration will be used by default

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

int x = 5;

int main()
{
  int a, x = 8, y = 3;
  cout << "x = " << x << endl;
  for(int i=0; i < 10; i++){
    int j = 1;
    j = 2*i + 1;
    a += j;
  }
  a = doit(y);
  cout << "a=" << a ;
  cout << "y=" << y << endl;
  cout << "glob. x" << ::x << endl;
}


int doit(int x)
{
   x--;
   return x;
}
```

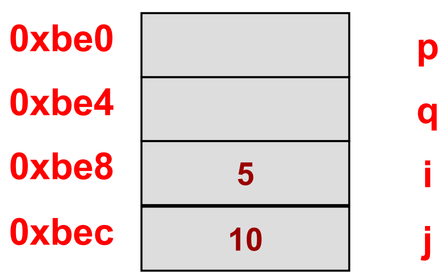


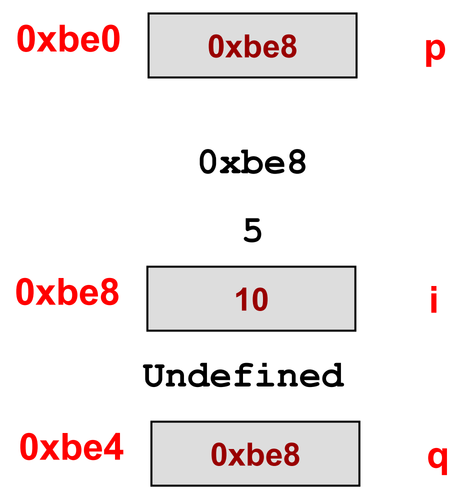
POINTERS & REFERENCES

Pointers in C/C++

- Generally speaking a "reference" can be a pointer or a C++ Reference
- 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
    - *ptr => object located at address given by ptr
    - *(&object) => object [i.e. * and & are inverse operators of each other]
- Example

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

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





Pointer Notes

- An uninitialized pointer is a pointer just waiting to cause a SEGFAULT
- **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

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…Take the type of pointer in the declaration and let each * in the expression 'cancel' one of the *'s in the declaration

Type	Expr	Yields
myptr = int*	*myptr	int
ourptr = int**	**ourptr	int
	*ourptr	int*

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

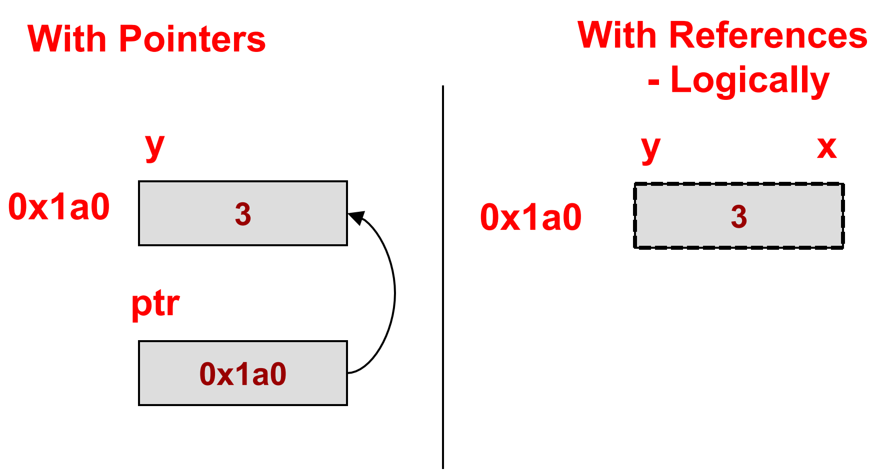
References in C/C++

- Reference type (type &)
- “Syntactic sugar” to make it so you don't have to use pointers
  - Probably really using/passing pointers behind the scenes
- 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

Using C++ References

- Can use it within the same function
- 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 &z;  // NO! must assign

 int &x = y;   // reference
               // declaration
  // we've not copied
  // y into x
  // we've created an alias
  // Now x can never reference
  //   any other int…only y!

  x++;    // y just got incr.
  cout << y << endl;
  return 0;
}
```



Swap Two Variables

- 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

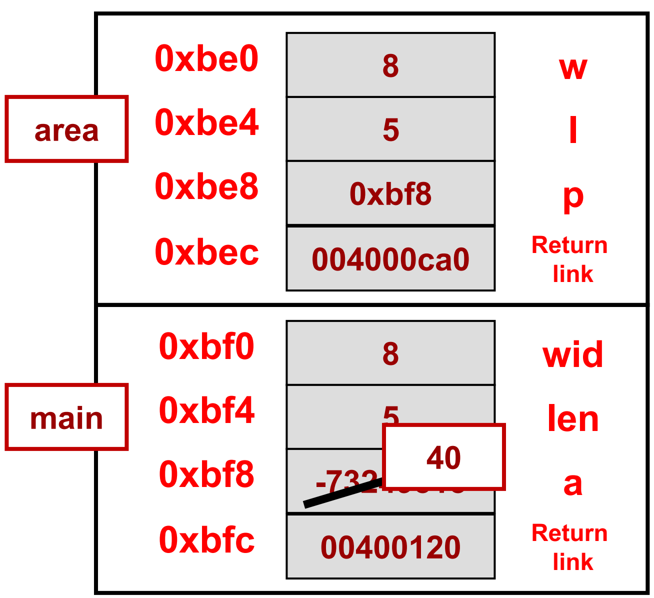
```
int main()
{
  int x=5,y=7;
  swapit(x,y);
 cout <<"x,y="<< x<<","<< y;
  cout << endl;
}

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

program output: x=7,y=5

Correct Usage of Pointers

- Can use a pointer to have a function modify the variable of another

Stack Area of RAM

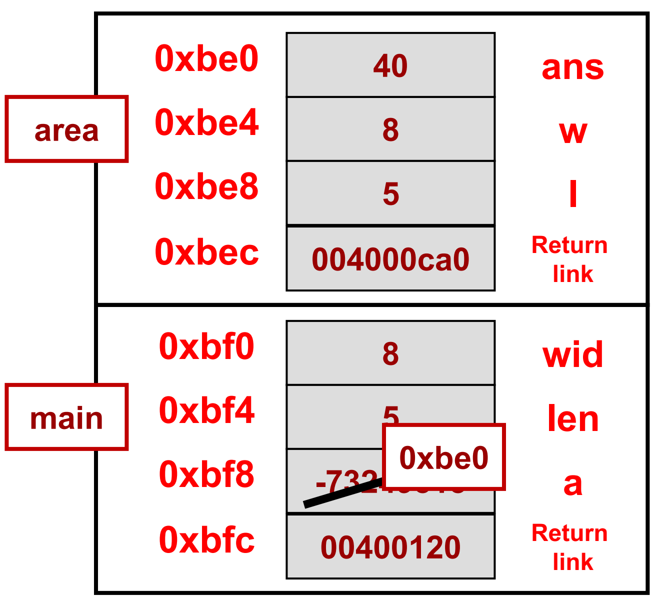



```
// Computes rectangle area,
//  prints it, & returns it
void area(int, int, int*);

int main()
{
  int wid = 8, len = 5, a;
  area(wid,len,&a);
}

void area(int w, int l, int* p)
{
  *p = w * l;
}
```

Misuse of Pointers

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

Stack Area of RAM

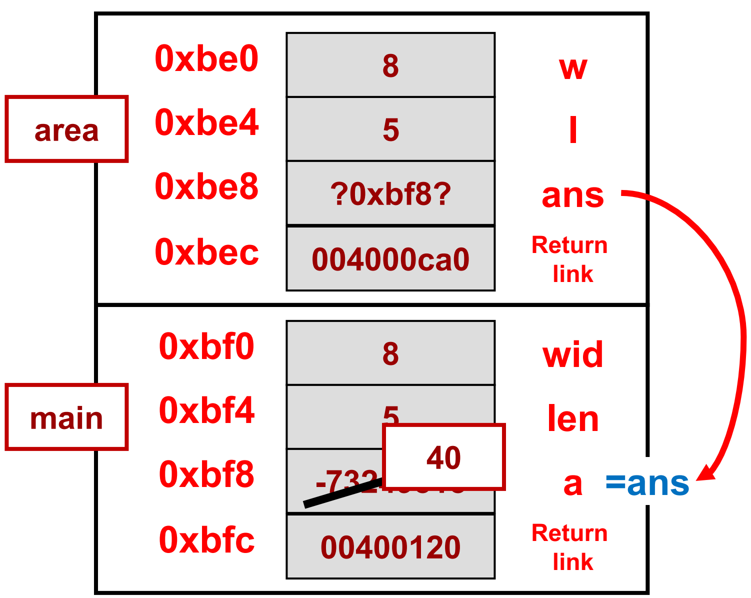



```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area(wid,len);
  cout << *a << endl;
}

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

  return &ans;
}
```

Use of C++ References

- We can pass using C++ reference
- The reference 'ans' 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 rectangle area,
//  prints it, & returns it
void area(int, int, int&);

int main()
{
  int wid = 8, len = 5, a;
  area(wid,len,a);
}

void area(int w, int l, int& ans)
{
  ans = w * l;
}
```

Pass-by-Value vs. -Reference

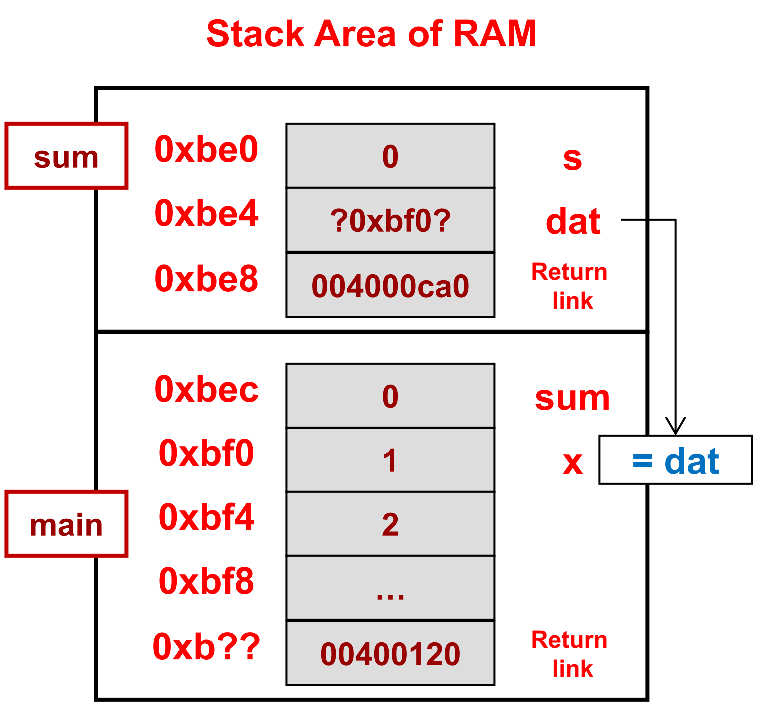
- Arguments are said to be:
  - Passed-by-value: A copy is made from one function and given to the other
  - Passed-by-reference: A reference (really the address) to the variable is passed to the other function

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




```
// Computes rectangle area,
//  prints it, & returns it
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++)
  {
     sum += dat[i];
  }
  return s;
}
```

Pointers vs. References

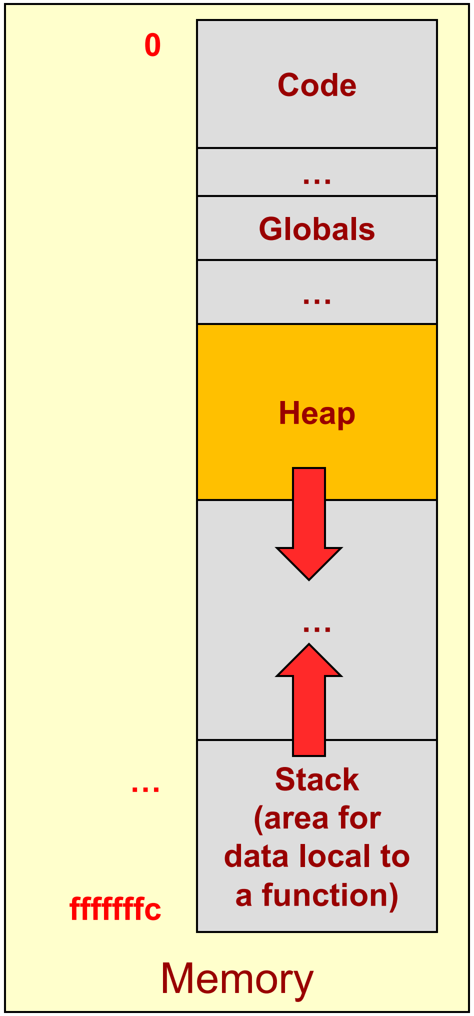
- How to tell references and pointers apart
  - Check if you see the '&' or '*' in a type declaration or expression

	Type	Expression
&	C++ Reference Var (int &val, vector<int> &vec)	Address-of (yields a pointer) &val => int *, &vec = vector<int>*
*	Pointer (int *valptr = &val, vector<int> *vecptr = &vec)	De-Reference (Value @ address) *valptr => val *vecptr => vec

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')
- Can be sized at run-time
  – int size;
    cin >> size;
    int *data = new int[size];

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;
}
```

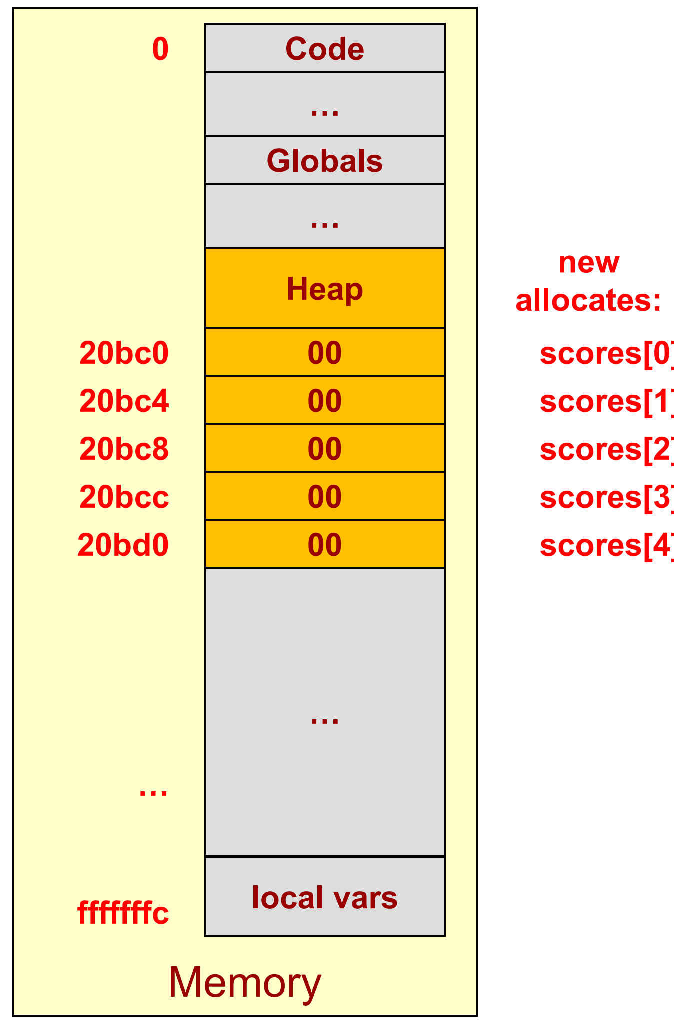
C++ `new` & `delete` operators

- 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 a 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

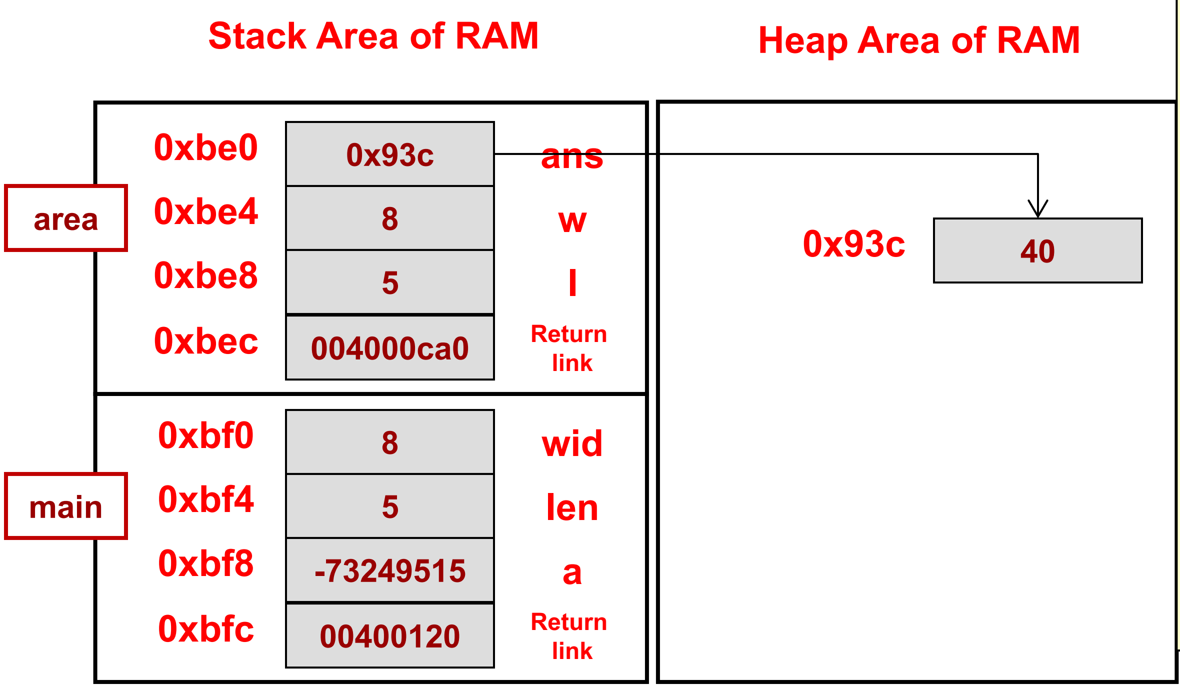
- __________ data = new int;
- __________ data = new char;
- __________ data = new char[100];
- __________ data = new char*[20];
- __________ data = new vector<string>;
- __________ data = new Student;

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

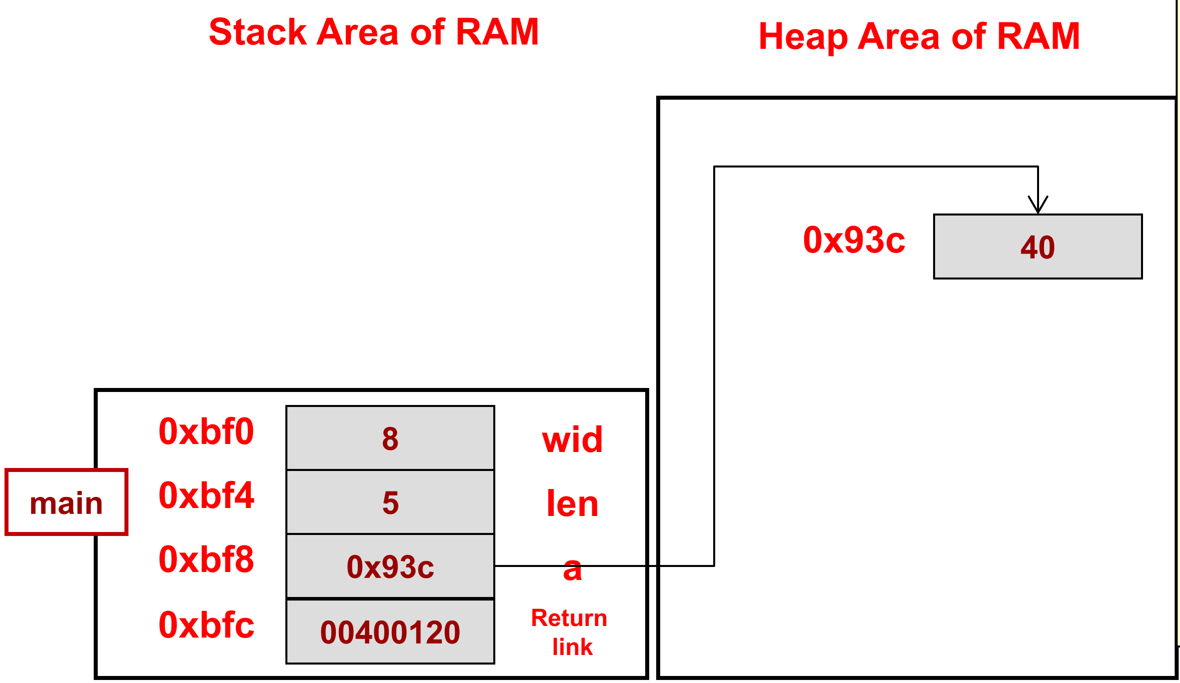
- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw these as boxes in the heap area




```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area(wid,len);
  cout << *a << endl;
  delete a;
}

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

Dynamic Allocation

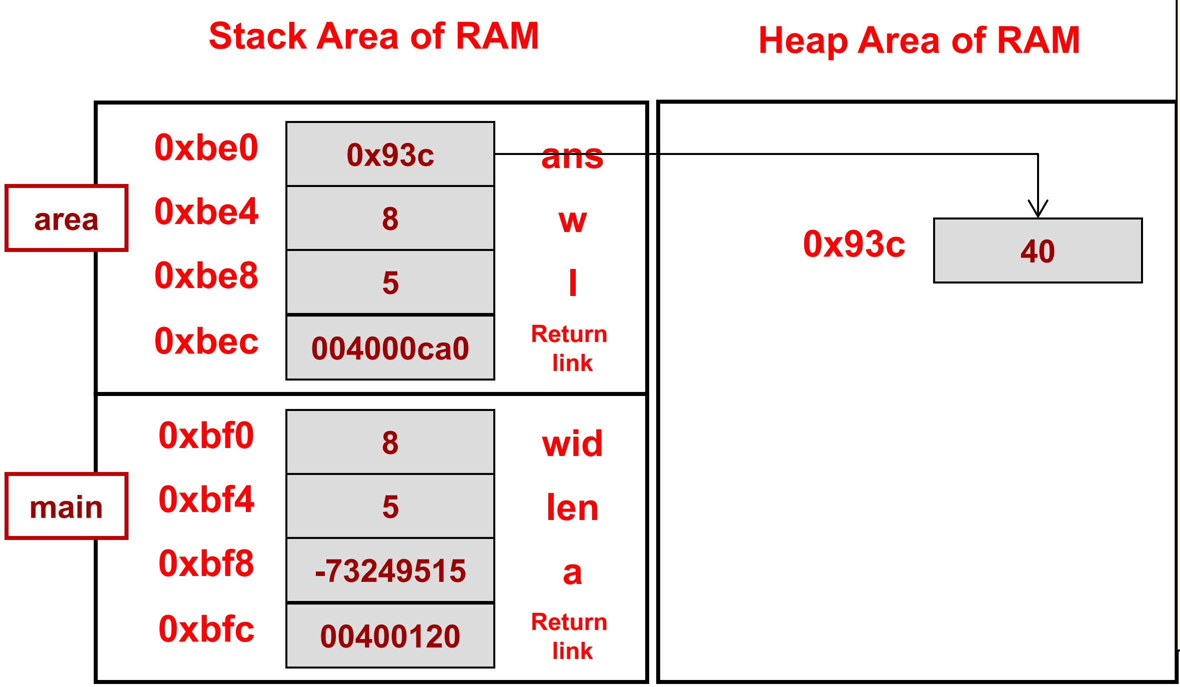
- Dynamic Allocation
  - Lives on the heap
    - Doesn't have a name, only pointer/address to it
  - Lives until you 'delete' it
    - Doesn't die at end of function (though pointer to it may)
- Let's draw these as boxes in the heap area




```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);

int main()
{
  int wid = 8, len = 5, *a;
  a = area(wid,len);
  cout << *a << endl;
  delete a;
}

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

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 these as boxes in the heap area

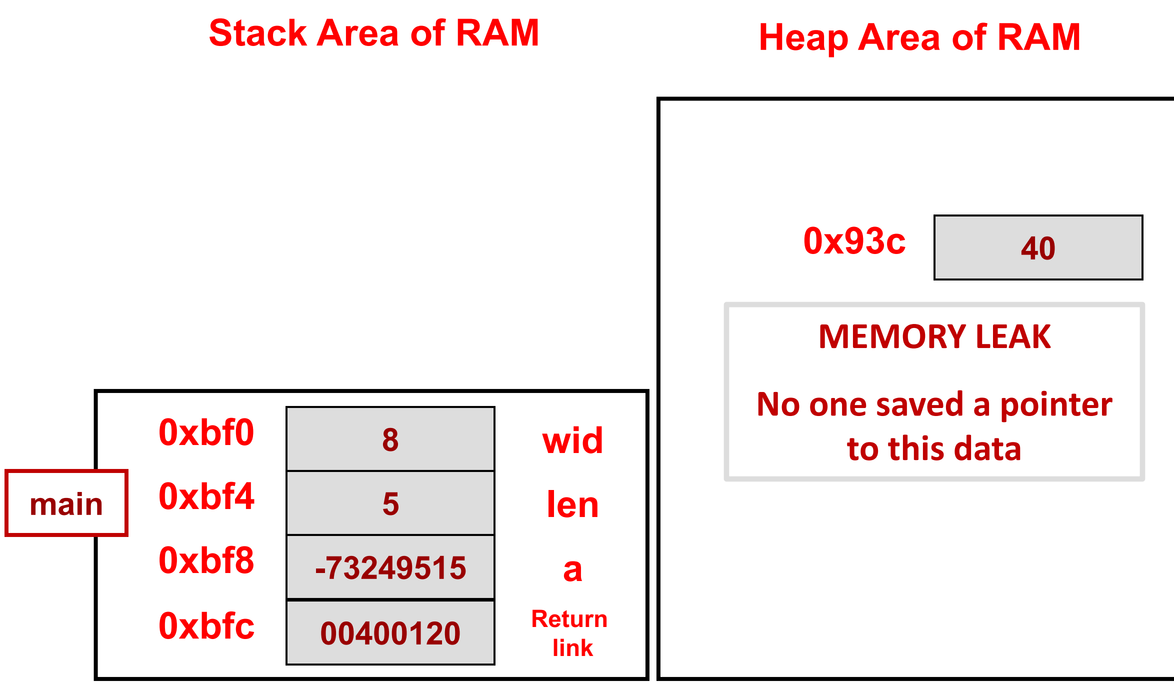



```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
  int wid = 8, len = 5, a;
  area(wid,len);
}


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

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 these as boxes in the heap area

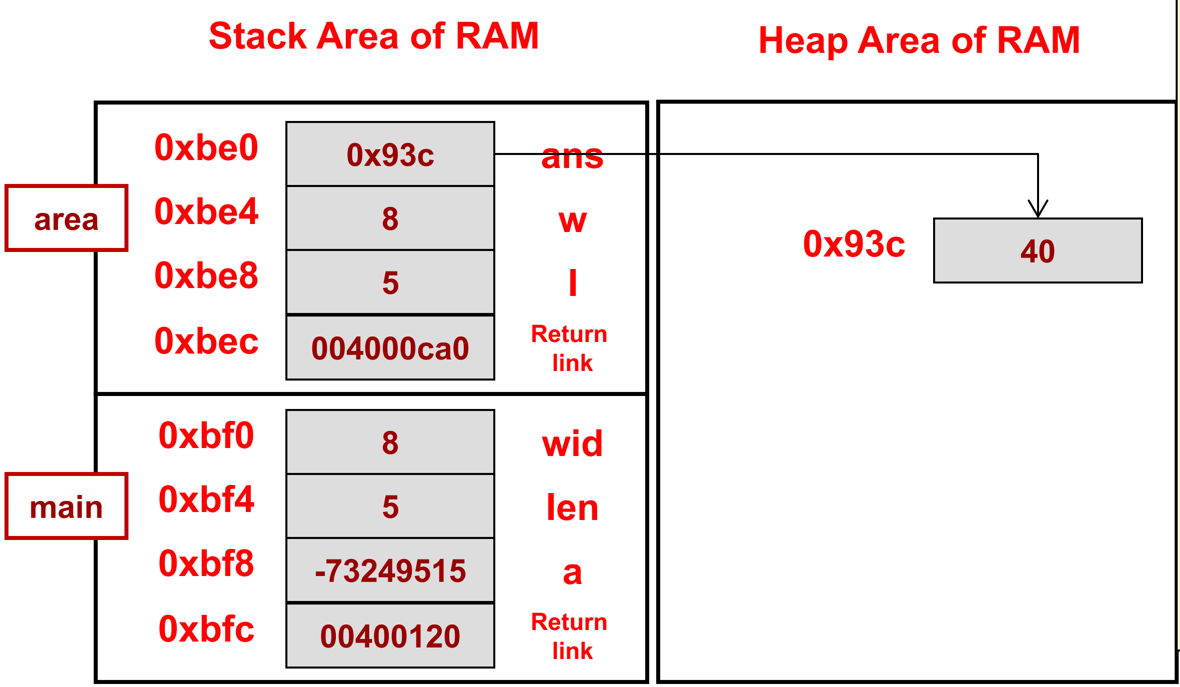
```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);
void print(int);

int main()
{
   int wid = 8, len = 5, a;
   area(wid,len);
}


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



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 these as boxes in the heap area

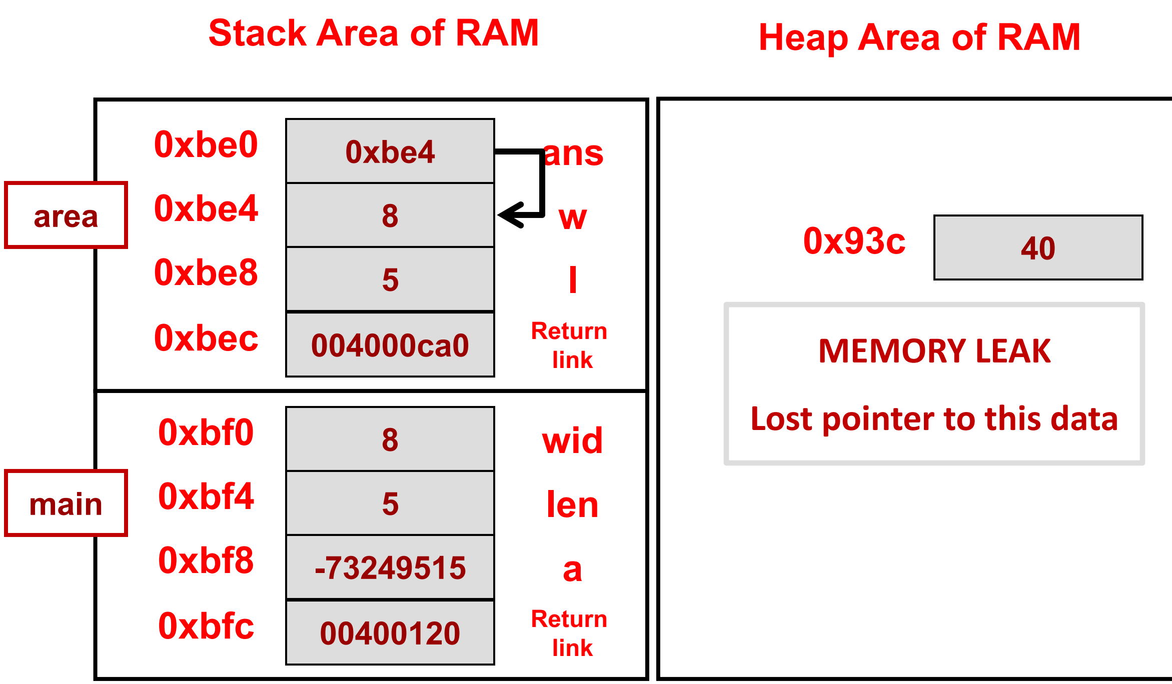
```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);

int main()
{
  int wid = 8, len = 5, a;
  area(wid,len);
}

int* area(int w, int l)
{
  int* ans = new int;
  ans = &w;
  return ans;
}
```



Dynamic Allocation

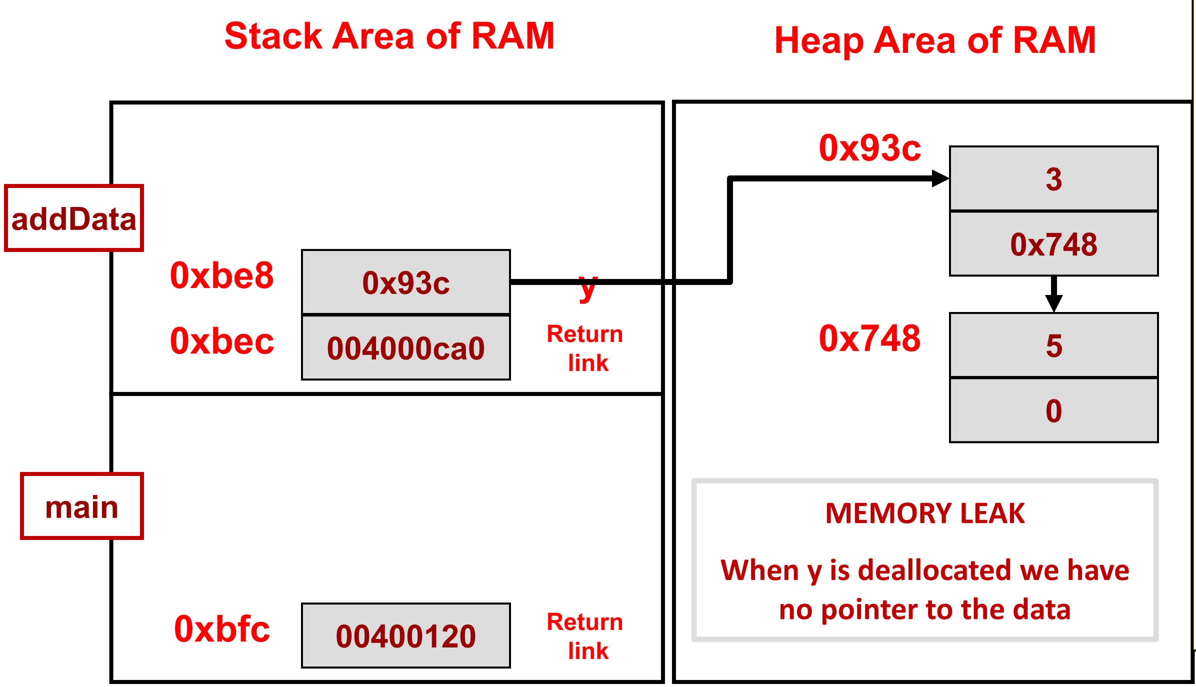
- Be sure you keep a pointer around somewhere otherwise you'll have a memory leak

```
// Computes rectangle area,
//  prints it, & returns it
int* area(int, int);

int main()
{
  int wid = 8, len = 5, a;
  area(wid,len);
}

int* area(int w, int l)
{
  int* ans = new int;
  ans = &w;
  return ans;
}
```



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

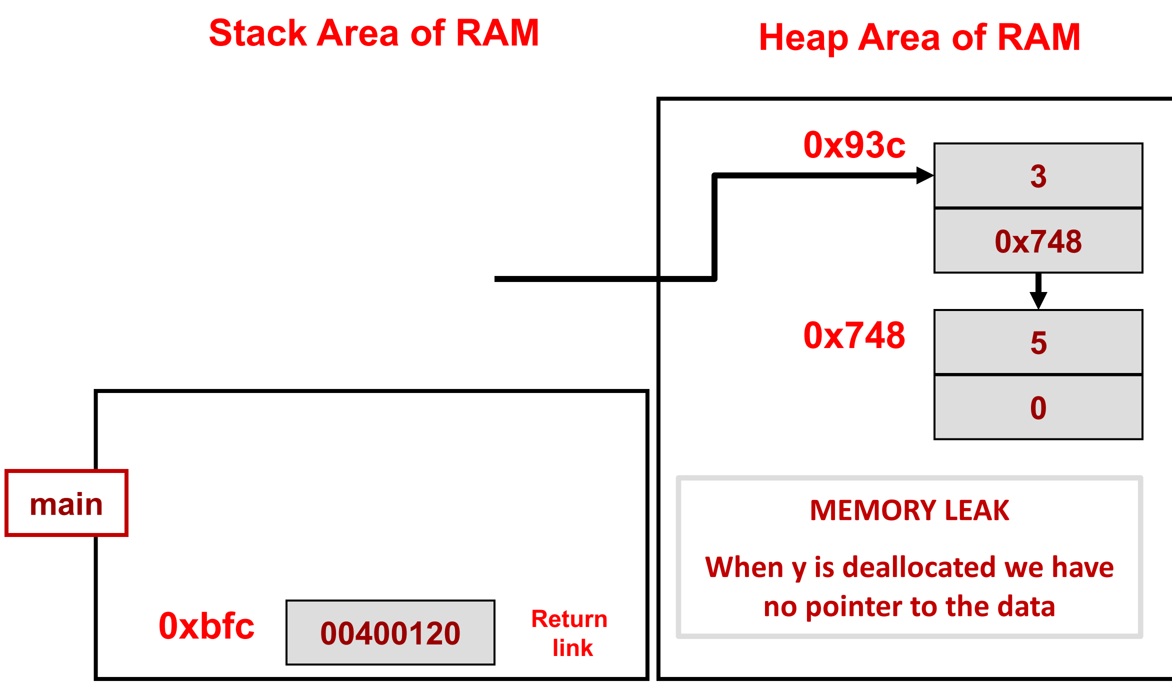



```
// Linked List example

struct Item {
   int val;
   Item* next;
};
class LinkedList {
  public:
    void push_back(int v);
  private:
    Item* head;
};
int main()
{
   addData();
}

void addData()
{
   LinkedList y;
   y.push_back(3);
   y.push_back(5);
}
```

Dynamic Allocation

- The LinkedList object is allocated as a static/local variable
  – But each element is allocated on the heap
- When x goes out of scope only the data members are deallocated
  – You may have a memory leak

An Appropriate Destructor Will Help Solve This



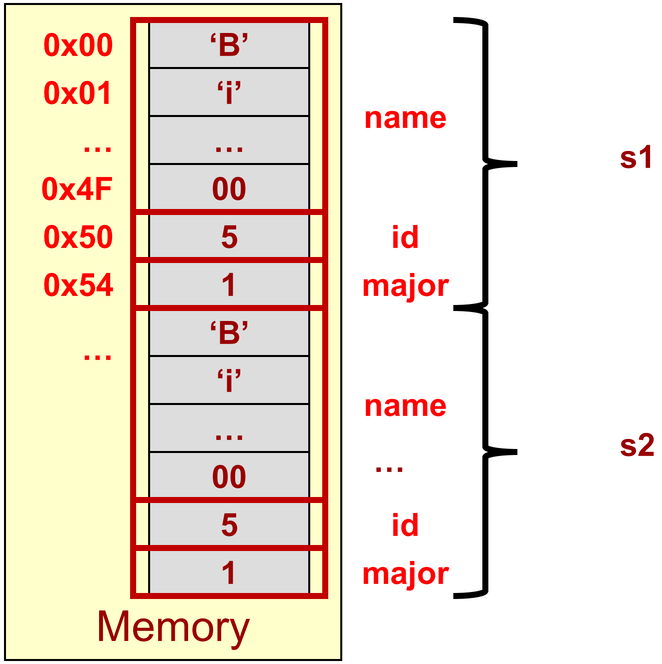

```
// LinkedList example

struct Item {
  int val;
  Item* next;
};
class LinkedList {
  public:
   void push_back(int v);
  private:
   Item* head;
};
int main()
{
  addData();
}

void addData()
{
  LinkedList y;
  y.push_back(3);
  y.push_back(5);
}
```

PRACTICE ACTIVITIES

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.

}
```

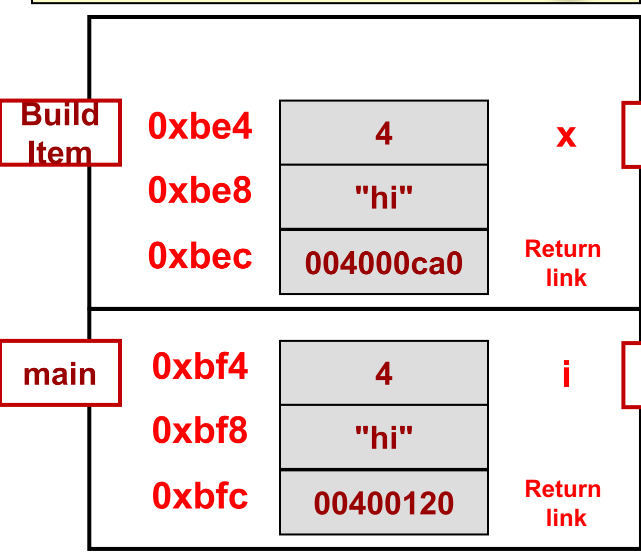
ex1

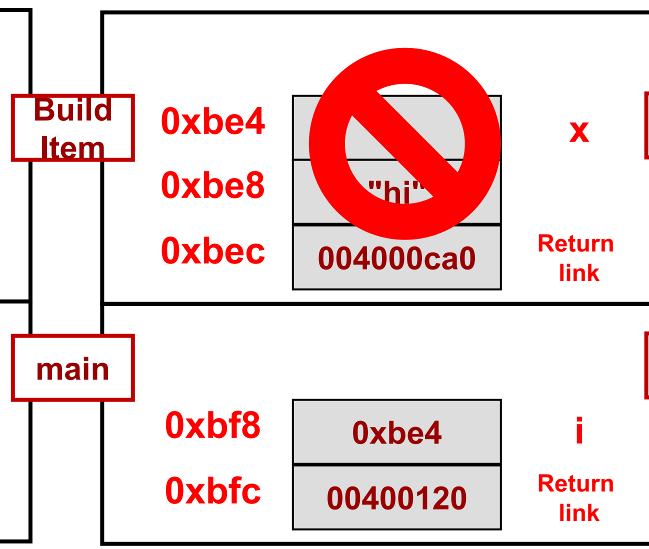
```
class Item
{ public:
  Item(int w, string y);
};
Item& buildItem()
{ Item x(4, "hi");
  return x;
}

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

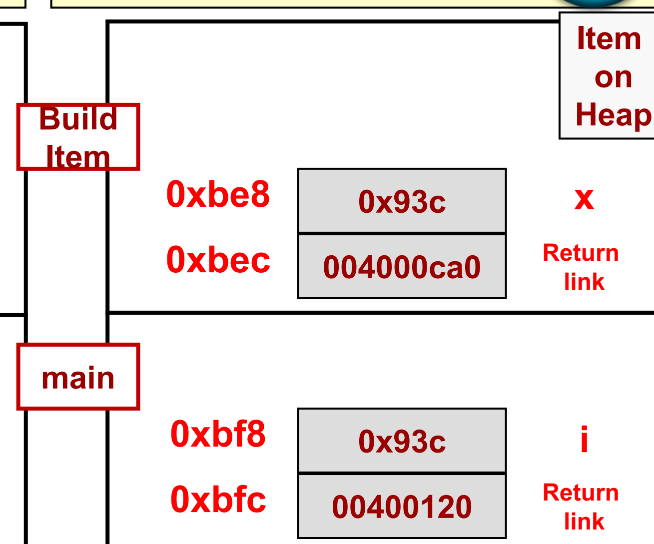
ex2

```
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





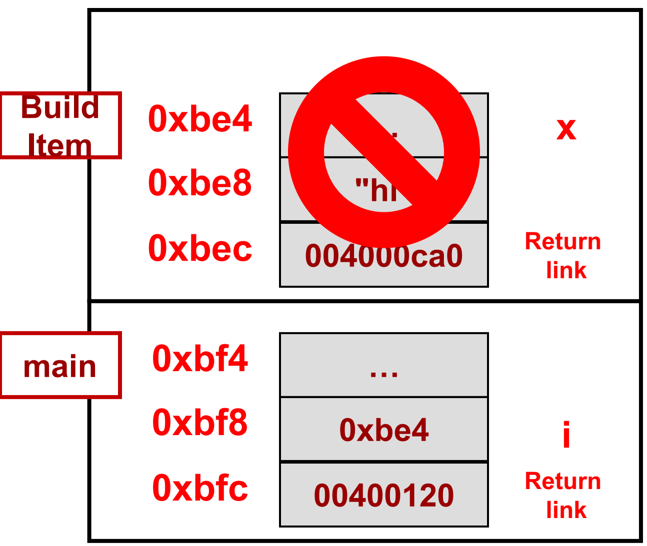


Understanding Memory Allocation

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

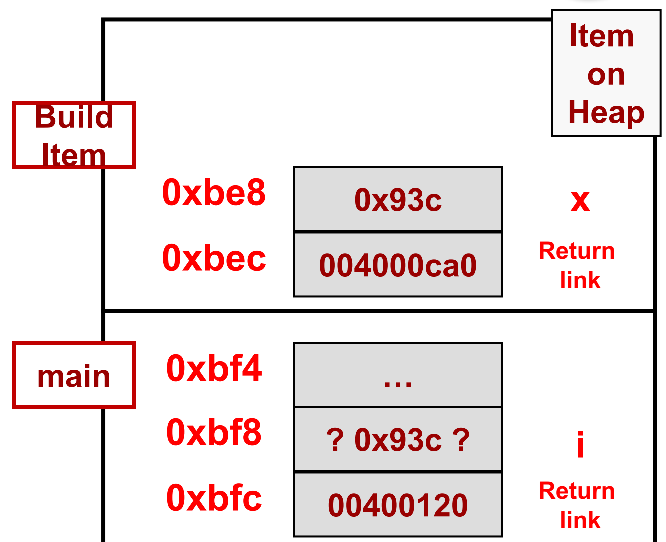
```
class Item
{ public:
  Item(int w, string y);
};
Item* buildItem()
{ Item x(4, "hi");
  return &x;
}

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

ex6:

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

ex7:

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

ex8:

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

You are responsible for this on your own since its covered in CS103

C++ LIBRARY REVIEW
(END LECTURE 1 SLIDES)

C++ Library

- String
- I/O Streams
- Vector

C Strings

- In C, strings are:
  - Character arrays (char mystring[80])
  - Terminated with a NULL character
  - Passed by reference/pointer (char *) to functions
  - Require care when making copies
    - Shallow (only copying the pointer) vs.
      Deep (copying the entire array of characters)
  - Processed using C String library (<cstring>)

String Function/Library (cstring)

In C, we have to pass the C-String as an argument for the function to operate on it

- int strlen(char *dest)
- int strcmp(char *str1, char *str2);
  - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise
- char *strcpy(char *dest, char *src);
  - strncpy(char *dest, char *src, int n);
  - Maximum of n characters copied
- char *strcat(char *dest, char *src);
  - strncat(char *dest, char *src, int n);
  - Maximum of n characters concatenated plus a NULL
- char *strchr(char *str, char c);
  - Finds first occurrence of character ‘c’ in str returning a pointer to that character or NULL if the character is not found

```
#include <cstring>
using namespace std;
int main() {
  char temp_buf[5];
  char str[] = "Too much";
  strcpy(temp_buf, str);
  strncpy(temp_buf, str, 4);
  temp_buf[4] = '\0'
  return 0;
}
```

C++ Strings

- So you don't like remembering all these details?
  - You can do it! Don't give up.
- C++ provides a 'string' class that **abstracts** all those worrisome details and **encapsulates** all the code to actually handle:
  - Memory allocation and sizing
  - Deep copy
  - etc.

String Examples

- Must:
  - #include <string>
  - using namespace std;
- Initializations / Assignment
  - Use initialization constructor
  - Use '=' operator
  - Can reassign and all memory allocation will be handled
- Redefines operators:
  - + (concatenate / append)
  - += (append)
  - ==, !=, >, <, <=, >= (comparison)
  - [] (access individual character)

http://www.cplusplus.com/reference/string/string/

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

int main(int argc, char *argv[]) {
  int len;
  string s1("CS is ");
  string s2 = "fun";

  s2 = "really fun";

  cout << s1 << " is " << s2 << endl;
  s2 = s2 + "!!!";
  cout << s2 << endl;
  string s3 = s1;
  if (s1 == s3){
    cout << s1 << " same as " << s3;
    cout << endl;
  }
  cout << "First letter is " << s1[0];
  cout << endl;
}
```

Output:

CS is really fun
really fun!!!
CS is same as CS is
First letter is C

More String Examples

- Size/Length of string
- Get C String (char *) equiv.
- Find a substring
  – Searches for occurrence of a substring
  – Returns either the index where the substring starts or string::npos
  – std::npos is a constant meaning ‘just beyond the end of the string’…it’s a way of saying ‘Not found’
- Get a substring
  – Pass it the start character and the number of characters to copy
  – Returns a new string
- Others: replace, rfind, etc.

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

int main(int argc, char *argv[]) {
  string s1("abc def");
  cout << "Len of s1: " << s1.size() << endl;

  char my_c_str[80];
  strcpy(my_c_str, s1.c_str() );
  cout << my_c_str << endl;

  if(s1.find("bc d") != string::npos)
    cout << "Found bc_d starting at pos=":
    cout << s1.find("bc_d") << endl;

  found = s1.find("def");
  if( found != string::npos){
    string s2 = s1.substr(found,3)
    cout << s2 << endl;
  }
}
```

Output:

Len of s1: 7
abc def
The string is: abc def
Found bc_d starting at pos=1
def

http://www.cplusplus.com/reference/string/string/

C++ Strings

- Why do we need the string class?
  - C style strings are character arrays (char[ ])
    - See previous discussion of why we don't like arrays
  - C style strings need a null terminator ('\0')

    "abcd" is actually a ***char[5] …*** Why?
  - Stuff like this won't compile:

    char my_string[7] = "abc" + "def";
- How can strings help?
  - Easier to use, less error prone
  - Has overloaded operators like +, =, [], etc.
  - Lots of built-in functionality (e.g. find, substr, etc.)

C++ Streams

- What is a “stream”?
  - A sequence of characters or bytes (of potentially infinite length) used for input and output.
- C++ has four major libraries we will use for streams:
  - <iostream>
  - <fstream>
  - <sstream>
  - <iomanip>
- Stream models some input and/or output device
  - fstream => a file on the hard drive;
  - cin => keyboard and cout => monitor
- C++ has two operators that are used with streams
  - Insertion Operator “<<”
  - Extraction Operator “>>”

C++ I/O Manipulators

- The <iomanip> header file has a number of "manipulators" to modify how I/O behaves
  - Alignment: internal, left, right, setw, setfill
  - Numeric: setprecision, fixed, scientific, showpoint
  - Other: endl, ends, flush, etc.
  - http://www.cplusplus.com/reference/iostream/manipulators/
- Use these inline with your cout/cerr/cin statements
  - double pi = 3.1415;
  - cout << setprecision(2) << fixed << pi << endl;

Understanding Extraction

- User enters value “512” at 1st prompt, enters “123” at 2nd prompt

int x=0;	X = 0	cin =
cout << “Enter X: “;	X = 0	cin = 5 1 2 \n
cin >> x;	X = 512	cin = \n
		cin.fail() is false
int y = 0;	Y = 0	cin = \n
cout << “Enter Y: “;	Y = 0	cin = \n 1 2 3 \n
cin >> y;	Y = 123	cin = \n
		cin.fail() is false

Understanding Extraction

- **User enters value "23 99" at 1st prompt, 2nd prompt skipped**

int x=0;	X = 0	cin =
cout << "Enter X: ";	X = 0	cin = `2` `3` ` ` `9` `9` `\n`
cin >> x;	X = 23	cin = ` ` `9` `9` `\n`
		cin.fail() is false
int y = 0;	Y = 0	cin = ` ` `9` `9` `\n`
cout << "Enter Y: ";	Y = 0	cin = ` ` `9` `9` `\n`
cin >> y;	Y = 99	cin = `\n`
		cin.fail() is false

Understanding Extraction

- User enters value "23abc" at 1st prompt, 2nd prompt fails

Code	Variable	cin
int x=0;	X = 0	cin =
cout << "Enter X: ";	X = 0	cin = 2 3 a b c \n
cin >> x;	X = 23	cin = a b c \n

cin.fail() is false

Code	Variable	cin
int y = 0;	Y = 0	cin = a b c \n
cout << "Enter Y: ";	Y = 0	cin = a b c \n
cin >> y;	Y = xxx	cin = a b c \n

cin.fail() is true

Understanding Extraction

- User enters value "23 99" at 1st prompt, everything read as string

string x;

cout << "Enter X: ";

getline(cin,x);

X = | |

cin =

X = | |

cin = | 2 | 3 | | 9 | 9 | \n | EOF |

X = | 23 99 |

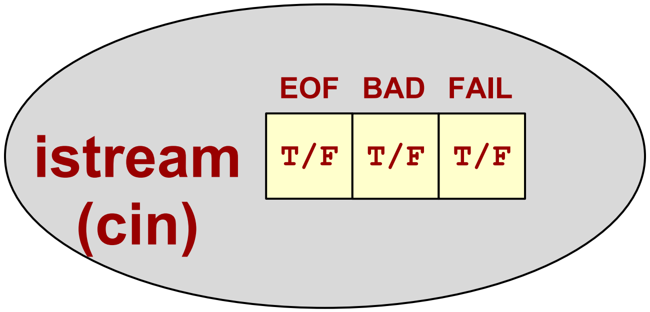
cin =

cin.fail() is
false

NOTE: \n character is discarded!

Understanding cin

- Things to remember
  - When a read operation on cin goes wrong, the fail flag is set
  - If the fail flag is set, all reads will automatically fail right away
  - This flag stays set until you clear it using the cin.clear() function
  - cin.good() returns true if ALL flags are false
- When you're done with a read operation on cin, you should wipe the input stream
  - Use the cin.ignore(...) method to wipe any remaining data off of cin
  - Example: cin.ignore(1000,'\n'); cin.clear();



Understanding Extraction

- User enters value "23abc" at 1st prompt, 2nd prompt fails

```
int y = 0;

cout << "Enter Y: ";

cin >> y;

cin.ignore(100, '\n');
// doing a cin >> here will
// still have the fail bit set
cin.clear();
// now safe to do cin >>
```

Y = | 0 | cin = | a | b | c | \n | EOF |

Y = | 0 | cin = | a | b | c | \n | EOF |

Y = | xxx | cin = | a | b | c | \n | EOF |

cin.fail() is **true**

cin = | EOF |

EOF	BAD	FAIL
0	0	1

cin = | EOF |

EOF	BAD	FAIL
0	0	0

C++ File I/O

- Use <fstream> library for reading/writing files
  - Use the open( ) method to get access to a file

    ofstream out; //ofstream is for writing, ifstream is for reading

    out.open(“my_filename.txt”) //must be a C style string!
- Write to a file exactly as you would the console!
  - out << “This line gets written to the file” << endl;
- Make sure to close the file when you're done
  - out.close();
- Use fail( ) to check if the file opened properly
  - out.open(“my_filename.txt”)
  - if(out.fail()) cerr << “Could not open the output file!”;

Validating User Input

- Reading user input is easy, validating it is hard
- What are some ways to track whether or not the user has entered valid input?
  - Use the fail( ) function on cin and re-prompt the user for input
  - Use a stringstream for data conversions and check the fail( ) method on the stringstream
  - Read data in as a string and use the cctype header to validate each character (http://www.cplusplus.com/reference/clibrary/cctype/)
  - for(int i=0; i < str.size(); i++)

```
if( ! isdigit(str[i]) )
    cerr << "str is not a number!" << endl
```

C++ String Stream

- If streams are just sequences of characters, aren't strings themselves like a stream?
  - The <sstream> library lets you treat C++ string objects like they were streams
- Why would you want to treat a string as a stream?
  - Buffer up output for later display
  - Parse out the pieces of a string
  - Data type conversions
    - This is where you'll use stringstream the most!
- Very useful in conjunction with string's getline(...)

C++ String Stream

- Convert numbers into strings (i.e. 12345 => "12345")

```
#include<sstream>
using namespace std;
int main()
{
 stringstream ss;
 int number = 12345;
 ss << number;

 string strNumber;
 ss >> strNumber;

 return 0;
}
```

sstream_test1.cpp

C++ String Stream

- Convert string into numbers [same as atoi()]

```
#include<sstream>

using namespace std;

int main()
{
 stringstream ss;

 string numStr = "12345";

 ss << numStr;


 int num;

 ss >> num;

 return 0;
}
```

sstream_test2.cpp

C++ String Stream

- Beware of re-using the same stringstream object for multiple conversions. It can be weird.
  – Make sure you clear it out between uses and re-init with an empty string
- Or just make a new stringstream each time

```
stringstream ss;

//do something with ss

ss.clear();
ss.str("");
// now you can reuse ss
// or just declare another stream
stringstream ss2;
```

C++ Arrays

- What are arrays good for?
  - Keeping collections of many pieces of the same data type (e.g. I want to store 100 integers)
  - int n[100];
- Each value is called out explicitly by its index
  - Indexes start at 0:
- Read an array value:
  - cout << "5th value = " << n[4] << endl;
- Write an array value
  - n[2] = 255;

C++ Arrays

- Unfortunately C++ arrays can be tricky...
  - Arrays need a contiguous block of memory
  - Arrays are difficult/costly to resize
  - Arrays don't know their own size
  - You must pass the size around with the array
  - Arrays don't do bounds checking
  - Potential for buffer overflow security holes
    - e.g. Twilight Hack: http://wiibrew.org/wiki/Twilight_Hack
  - Arrays are not automatically initialized
  - Arrays can't be directly returned from a function
  - You have to decay them to pointers

C++ Vectors

- Why do we need the vector class?
  - Arrays are a fixed size. Resizing is a pain.
  - Arrays don't know their size (no bounds checking)
  - This compiles:
    - int stuff[5];
    - cout << stuff[-1] << “ and “ << stuff[100];
- How can vectors help?
  - Automatic resizing to fit data
  - Sanity checking on bounds
  - They do everything arrays can do, but more safely
    - Sometimes at the cost of performance
  - See http://www.cplusplus.com/reference/stl/

Vector Class

- Container class (what it contains is up to you via a template)
- Mimics an array where we have an indexed set of homogenous objects
- Resizes automatically

1 my_vec

0	1	2	3	4
50	51	52	53	54

2 my_vec

0	1	2	3	4	5	6
30	51	52	53	54	10	8

3 my_vec

0	1	2	3	4	5	
30	51	52	53	54	10	

4 my_vec

0	1	2	3	4	5	
30	43	51	53	54	10	

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> my_vec(5); // init. size of 5
  for(unsigned int i=0; i < 5; i++){          1
    my_vec[i] = i+50;
  }
  my_vec.push_back(10); my_vec.push_back(8);
  my_vec[0] = 30;                             2
  unsigned int i;
  for(i=0; i < my_vec.size(); i++){
    cout << my_vec[i] << " ";
  }
  cout << endl;

  int x = my_vec.back(); // gets back val.    3
  x += my_vec.front(); // gets front val.
  // x is now 38;
  cout << "x is " << x << endl;
  my_vec.pop_back();

  my_vec.erase(my_vec.begin() + 2);           4
  my_vec.insert(my_vec.begin() + 1, 43);
  return 0;
}
```

Vector Class

- constructor
  - Can pass an initial number of items or leave blank
- operator[ ]
  - Allows array style indexed access (e.g. myvec[1] + myvec[2])
- push_back(T new_val)
  - Adds a **copy** of new_val to the end of the array allocating more memory if necessary
- size(), empty()
  - Size returns the current number of items stored as an unsigned int
  - Empty returns True if no items in the vector
- pop_back()
  - Removes the item at the back of the vector (does not return it)
- front(), back()
  - Return item at front or back
- erase(*iterator*)
  - Removes item at specified index (use begin() + index)
- insert(*iterator*, T new_val)
  - Adds new_val at specified index (use begin() + index)

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> my_vec(5); // 5= init. size
  for(unsigned int i=0; i < 5; i++){
    my_vec[i] = i+50;
  }
  my_vec.push_back(10); my_vec.push_back(8);
  my_vec[0] = 30;
  for(int i=0; i < my_vec.size(); i++){
    cout << my_vec[i] << " ";
  }
  cout << endl;

  int x = my_vec.back(); // gets back val.
  x += my_vec.front(); // gets front val.
  // x is now 38;
  cout << "x is " << x << endl;
  my_vec.pop_back();

  my_vec.erase(my_vec.begin() + 2);
  my_vec.insert(my_vec.begin() + 1, 43);
  return 0;
}
```

Vector Suggestions

- If you don't provide an initial size to the vector, you must add items using push_back()
- When iterating over the items with a for loop, used an 'unsigned int'
- When adding an item, a copy will be made to add to the vector
- [] or at() return a reference to an element, not a copy of the element
- Usually pass-by-reference if an argument to avoid the wasted time of making a copy

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
  vector<int> my_vec;
  for(int i=0; i < 5; i++){
    // my_vec[i] = i+50; // doesn't work
    my_vec.push_back(i+50);
  }
  for(unsigned int i=0;
      i < my_vec.size();
      i++)
  { cout << my_vec[i] << " ";  }
  cout << endl;

  my_vec[1] = 5; my_vec.at(2) = 6;

  do_something(myvec);

  return 0;
}

void do_something(vector<int> &v)
{
  // process v;

}
```