CSCI 104
Memory Allocation
Mark Redekopp
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
Automatic/Local Variables

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

```c
// 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;
}
```
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

```cpp
#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 << " ;
    cout << "glob. x" << ::x << " ;
}

int doit(int x)
{
    x--;  // Adjusted to match the context
    return x;
}
```

Address

<table>
<thead>
<tr>
<th>Code</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Globals</td>
<td>x = 5</td>
</tr>
<tr>
<td>Heap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>main: (a=2, x=8, y=3)</td>
</tr>
</tbody>
</table>

Memory (RAM)
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;
q = p;
```

```
0xbe0 0xbe4 0xbe8 0xbec
p     q     i     j
```

```
0xbe0 0xbe8
p
```
```
0xbe8
5
```
```
0xbe8
10
```
```
Undefined
```
```
0xbe4
```
```
0xbe8
q
```
**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

<table>
<thead>
<tr>
<th>Type</th>
<th>Expr</th>
<th>Yields</th>
</tr>
</thead>
<tbody>
<tr>
<td>myptr = int*</td>
<td>*myptr</td>
<td>int</td>
</tr>
<tr>
<td>ourptr = int**</td>
<td>**ourptr</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>*ourptr</td>
<td>int*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>myptr</td>
<td></td>
</tr>
<tr>
<td>*myptr</td>
<td></td>
</tr>
<tr>
<td>(*ourptr) + 1</td>
<td></td>
</tr>
<tr>
<td>myptr + 2</td>
<td></td>
</tr>
<tr>
<td>ourptr</td>
<td></td>
</tr>
</tbody>
</table>
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.

```cpp
#include <iostream>

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.
    std::cout << y << endl;
    return 0;
}
```

With Pointers

- Logically

With References

y 3
x 3
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)

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

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

**program output:** x=5,y=7
Correct Usage of Pointers

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

```c
// 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

```c
// 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;
}
```

Stack Area of RAM

<table>
<thead>
<tr>
<th>0xbe0</th>
<th>0xbe4</th>
<th>0xbe8</th>
<th>0xbec</th>
<th>0xbf0</th>
<th>0xbf4</th>
<th>0xbf8</th>
<th>0xbfc</th>
</tr>
</thead>
<tbody>
<tr>
<td>ans</td>
<td>w</td>
<td>l</td>
<td>Return link</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>8</td>
<td>5</td>
<td>004000ca0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Return link

<table>
<thead>
<tr>
<th>0xbe0</th>
<th>0xbe4</th>
<th>0xbe8</th>
<th>0xbec</th>
<th>0xbf0</th>
<th>0xbf4</th>
<th>0xbf8</th>
<th>0xbfc</th>
</tr>
</thead>
<tbody>
<tr>
<td>wid</td>
<td>len</td>
<td>a</td>
<td>Return link</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>-732</td>
<td>0xbe0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00400120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

```cpp
// 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

<table>
<thead>
<tr>
<th>Pass-by-Value Benefits</th>
<th>Pass-by-Reference Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Protects the variable in the caller since a copy is made (any modification doesn’t affect the original)</td>
<td>+ Allows another function to modify the value of variable in the caller + Saves time vs. copying</td>
</tr>
</tbody>
</table>

• 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 \( \text{sum()} \) by reference
  - Notice that likely the computer passes the address to \( \text{sum()} \) but you should just think of \( \text{dat} \) as an alias for \( x \)

```cpp
// Computes rectangle area, // prints it, & returns it
int \text{sum}(\text{const vector<int>&});

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

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

Stack Area of RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbe0</td>
<td>0</td>
</tr>
<tr>
<td>0xbe4</td>
<td>0xbe0?</td>
</tr>
<tr>
<td>0xbe8</td>
<td>004000ca0</td>
</tr>
<tr>
<td>0xbec</td>
<td>0</td>
</tr>
<tr>
<td>0xbf0</td>
<td>1</td>
</tr>
<tr>
<td>0xbf4</td>
<td>2</td>
</tr>
<tr>
<td>0xbf8</td>
<td>...</td>
</tr>
<tr>
<td>0xb??</td>
<td>00400120</td>
</tr>
</tbody>
</table>

Return link
Pointers vs. References

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>C++ Reference Var (int &amp;val, vector&lt;int&gt; &amp;vec)</td>
</tr>
<tr>
<td></td>
<td>Address-of (yields a pointer) &amp;val =&gt; int <em>, &amp;vec = vector&lt;int&gt;</em></td>
</tr>
<tr>
<td>*</td>
<td>Pointer (int *valptr = &amp;val, vector&lt;int&gt; *vecptr = &amp;vec)</td>
</tr>
<tr>
<td></td>
<td>De-Reference (Value @ address) *valptr =&gt; val *vecptr =&gt; vec</td>
</tr>
</tbody>
</table>
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

```c
#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 *dpotr = 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 dpotr;
  - use **delete []** for pointers to arrays
    - delete [] myarray;
Dynamic Memory Allocation

```c
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;
}
```

```c
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

```cpp
// 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

```c
// 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;
}
```

Stack Area of RAM

Heap Area of RAM

MEMORY LEAK

No one saved a pointer to this data
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

```c
// 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

Stack Area of RAM

Heap 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;
  area(wid, len);
}

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

MEMORY LEAK
No one saved a pointer to this data
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

### Stack Area of RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbe0</td>
<td>0x93c</td>
</tr>
<tr>
<td>0xbe4</td>
<td>8</td>
</tr>
<tr>
<td>0xbe8</td>
<td>5</td>
</tr>
<tr>
<td>0xbec</td>
<td>004000ca0</td>
</tr>
</tbody>
</table>

### Heap Area of RAM

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xbf0</td>
<td>8</td>
</tr>
<tr>
<td>0xbf4</td>
<td>5</td>
</tr>
<tr>
<td>0xbf8</td>
<td>-73249515</td>
</tr>
<tr>
<td>0xbf0</td>
<td>00400120</td>
</tr>
</tbody>
</table>

```cpp
// 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;
    return ans;
}
```
Dynamic Allocation

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

```cpp
// Computes rectangle area, // prints it, & returns it
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

Stack Area of RAM

Heap Area of RAM

// Computes rectangle area, // prints it, & returns it
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

```cpp
#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?

- **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(4, "hi");
    return x;
  }
  int main()
  { Item& i = buildItem();
    // access i’s data
  }
  ``

- **ex3**
  ```
  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
Understanding Memory Allocation

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

### ex4

```cpp
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
}
```

### ex5

```cpp
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
}
```
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

Item on Heap

Build Item

0xbe8
0xbec

0x93c
0x4000ca0

x

Return link

main

0xbf4
0xbf8
0xbfc

4
"hi"
0x400120

Item on Heap

Build Item

0xbe8
0xbec

0x93c
0x4000ca0

x

Return link

main

0xbf4
0xbf8
0xbfc

0x93c
0x400120

Return link
STREAMS REVIEW
Kinds of Streams

- I/O streams
  - Keyboard (cin) and monitor (cout)
- File streams – Contents of file are the stream of data
  - #include <fstream> and #include <iostream>
  - ifstream and ofstream objects
- String streams
  - #include <sstream> and #include iostream
  - sstream objects
- Streams support appropriate << or >> operators as well as .fail(), .getline(), .get(), .eof() member functions
C++ Stream Input

- `cin`, `ifstreams`, and `stringstreams` can be used to accept data from the user
  - `int x;
  - cout << "Enter a number: ";
  - cin >> x;
- What if the user does not enter a valid number?
  - Check `cin.fail()` to see if the read worked
- What if the user enters multiple values?
  - `>>` reads up until the first piece of whitespace
  - `cin.getline()` can read a max number of chars until it hits a delimiter but only works for C-strings (character arrays)
    - `cin.getline(buf, 80)` // reads everything through a '\n' // stopping after 80 chars if no '\n'
    - `cin.getline(buf, 80, ';')` // reads everything through a ';' // stopping after 80 chars if no ';
  - The <string> header defines a `getline(...)` method that will read an entire line (including whitespace):
    - `string x;
    - getline(cin,x,';');` // reads everything through a ';'
When Does It Fail

- For files & string streams the stream doesn't fail until you read PAST the EOF

```cpp
char buf[40];
ifstream inf(argv[1]);
inf >> buf;
inf >> buf;
inf >> buf;
```

File text: `The end. \nEOF`

Getp

EOF BAD FAIL

buf

The \0 0 0 0

EOF BAD FAIL

buf

end. \0 0 0 0

EOF BAD FAIL

buf

end. \0 1 0 1
Which Option?

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

int main()
{
    vector<int> nums;
    ifstream ifile("data.txt");
    int x;
    while( !ifile.fail() ){
        ifile >> x;
        nums.push_back(x);
    }
    ...
}

Need to check for failure after you extract but before you store/use

A stream returns itself after extraction
A stream can be used as a bool (returns true if it hasn't failed)
Choices

Where is my data?
- Keyboard (use _____)
- File (use _____)
- String (use ______)

Do I know how many items to read?
- Yes, n items Use ______
- No, arbitrary Use ______
Choices

What type of data?

- Text
- Integers/Doubles

Is it delimited?

- Yes
- No
- Yes
Choices

Where is my data?

Keyboard (use iostream [cin])

File (use ifstream)

String (use stringstream)

Do I know how many items to read?

Yes, n items
Use for(i=0;i<n;i++)

No, arbitrary
Use while(cin >> temp) or while(getline(cin,temp))
Choices

What type of data?

Text
(getline or >>)
ggetline ALWAYS returns text

Ints/Doubles
(Use >> b/c it converts text to the given type)

Is it delimited?

Yes at newlines
Use getline()

No, stop on any whitespace...use >>

Yes at special chars (';' or ' ,')
Use getline with 3rd input parameter (delimeter parameter)
getline() and stringstreams

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use `>>`?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '\n' look the same to `>>` and it will skip over them)
- We can use `getline()` to get the whole line, then a `stringstream` with `>>` to parse out the pieces

```cpp
int num_lines = 0;
int total_words = 0;

ifstream myfile(argv[1]);

string myline;
while( getline(myfile, myline) ){
    stringstream ss(myline);
    string word;
    while( ss >> word ) {
        total_words++;
        num_lines++;
    }
}

double avg = (double) total_words / num_lines;

cout << "Avg. words per line: ";
cout << avg << endl;
```

The fox jumped over the log.
The bear ate some honey.
The CS student solved a hard problem.
Using Delimiters

- Imagine a file has a certain format where you know related data is on a single line of text but aren't sure how many data items will be on that line
- Can we use `>`?
  - No it doesn't differentiate between different whitespace (i.e. a ' ' and a '\n' look the same to `>>` and it will skip over them)
- We can use `getline()` to get the whole line, then a `stringstream` with `>>` to parse out the pieces

```cpp
vector<string> mywords;

ifstream myfile(argv[1]);

string myline;
getline(myfile, myline, '(');
// gets "garbage stuff"
// and throws away '(

generate(myfile, myline, ')');
// gets "words I care about"
// and throws away ')'

stringstream ss(myline);
string word;
while( ss >> word ) {
    mywords.push_back(word);
}
```

Text file:

```
garbage stuff (words I care about) junk
```

```
vector<string> mywords;

ifstream myfile(argv[1]);

string myline;
getline(myfile, myline, '(');
// gets "garbage stuff"
// and throws away '(

generate(myfile, myline, ')');
// gets "words I care about"
// and throws away ')'

stringstream ss(myline);
string word;
while( ss >> word ) {
    mywords.push_back(word);
}
```

```
mywords
```

```
0 1 2 3
"words" "I" "care" "about"
```
Choosing an I/O Strategy

• Is my data delimited by particular characters?
  – Yes, stop on newlines: Use getline()
  – Yes, stop on other character: User getline() with optional 3rd character
  – No, Use >> to skip all whitespaces and convert to a different data type (int, double, etc.)

• If "yes" above, do I need to break data into smaller pieces (vs. just wanting one large string)
  – Yes, create a stringstream and extract using >>
  – No, just keep the string returned by getline()

• Is the number of items you need to read known as a constant or a variable read in earlier?
  – Yes, Use a loop and extract (>>) values placing them in array or vector
  – No, Loop while extraction doesn't fail placing them in vector

Remember: getline() always gives text/string. To convert to other types it is easiest to use >>
RECURSION
Recursion

• Problem in which the solution can be expressed in terms of itself (usually a smaller instance/input of the same problem) and a base/terminating case

• Input to the problem must be categorized as a:
  – Base case: Solution known beforehand or easily computable (no recursion needed)
  – Recursive case: Solution can be described using solutions to smaller problems of the same type
    • Keeping putting in terms of something smaller until we reach the base case

• Factorial: \( n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 \)
  – \( n! = n \times (n-1)! \)
  – Base case: \( n = 1 \)
  – Recursive case: \( n > 1 \Rightarrow n \times (n-1)! \)
Recursive Functions

• Recall the system stack essentially provides separate areas of memory for each ‘instance’ of a function.

• Thus each local variable and actual parameter of a function has its own value within that particular function instance’s memory space.

C Code:

```c
int fact(int n) {
    if(n == 1) {
        // base case
        return 1;
    } else {
        // recursive case
        return n * fact(n-1);
    }
}
```
Recursion & the Stack

• Must return back through the each call

```cpp
int fact(int n)
{
    if(n == 1){
        // base case
        return 1;
    }
    else {
        // recursive case
        return n * fact(n-1);
    }
}

int main()
{
    int val = 4;
    cout << fact(val) << endl;
}
```
Recursion

- Google is in on the joke too...
Recursive Functions

• Many loop/iteration based approaches can be defined recursively as well

C Code:

```c
int main()
{
    int data[4] = {8, 6, 7, 9};
    int size=4;
    int sum1 = isum_it(data, size);
    int sum2 = rsum_it(data, size);
}

int isum_it(int data[], int len)
{
    int sum = data[0];
    for(int i=1; i < len; i++){
        sum += data[i];
    }
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}
```
Recursive Call Timeline

int main()
{
    int data[4] = {8, 6, 7, 9};
    int size=4;
    int sum2 = rsum_it(data, size);
    ...
}

int rsum_it(int data[], int len)
{
    if(len == 1)
        return data[0];
    else
        int sum = rsum_it(data, len-1);
    return sum + data[len-1];
}

int sum = 8
return 8+data[1];
int sum = 14
return 14+data[2];
int sum = 21
return 21+data[3];

Each instance of rsum_it has its own len argument and sum variable

© 2015 by Mark Redekopp, All Rights Reserved  Every instance of a function has its own copy of local variables
System Stack & Recursion

- The system stack makes recursion possible by providing separate memory storage for the local variables of each running instance of the function.

```c
int main()
{
    int data[4] = {8, 6, 7, 9};
    int size = 4;
    int sum2 = rsum_it(data, size);
}

int rsum_it(int data[], int len)
{
    if (len == 1)
        return data[0];
    else
    {
        int sum =
            rsum_it(data, len - 1);
        return sum + data[len - 1];
    }
}  ```
HELPER FUNCTIONS
Exercise

• Write a recursive routine to find the maximum element of an array containing POSITIVE integers.

\[
\text{int } \text{data}[4] = \{8, 9, 7, 6\};
\]

• Primary signature:

\[
\text{int } \text{max}(\text{int}\ast \text{data}, \text{int } \text{len});
\]

• For recursion we usually need some parameter to tell use which item we are responsible for...thus the signature needs to change. We can make a helper function.

• The client uses the original:

\[
\text{int } \text{max}(\text{int}\ast \text{data}, \text{int } \text{len});
\]

• But it just calls:

\[
\text{int } \text{max}(\text{int}\ast \text{data}, \text{int } \text{len}, \text{int } \text{curr});
\]
Exercise – Helper Function

• Head recursion

int data[4] = {8, 9, 7, 6};

int max(int* data, int len)
{
    return max(data, len, 0);
}

int max(int* data, int len, int curr)
{
    if(curr == len) return 0;
    else {
        int prevmax = max(data, len, curr+1);
        if(data[curr] > prevmax)
            return data[curr];
        else
            return prevmax;
    }
}

• Tail recursion

// The client only wants this
int max(int* data, int len);

// But to do the job we need this
void max(int* data, int len, int curr, int& mx);

int max(int* data, int len)
{
    int mymax = 0;
    max(data, len, 0, mymax);
    return mymax;
}

void max(int* data, int len, int curr, int& mx)
{
    if(curr == len) return;
    else {
        if(data[curr] > mx)
            mx = data[curr];
        else
            max(data, len, curr+1, mx);
    }
Exercise

• We can also formulate things w/o the helper function in this case...

```c
int data[4] = {8, 6, 9, 7};
```

```c
int max(int* data, int len) {
    if(len == 1) return data[0];
    else {
        int prevmax = max(data, len-1);
        if(data[len-1] > prevmax)
            return data[len-1];
        else
            return prevmax;
    }
}
```
GENERATING ALL COMBINATIONS
Recursion's Power

• The power of recursion often comes when each function instance makes *multiple* recursive calls

• As you will see this often leads to exponential number of "combinations" being generated/explored in an easy fashion
## Binary Combinations

- If you are given the value, $n$, and a string with $n$ characters could you generate all the combinations of $n$-bit binary?
- Do so recursively!

### Exercise: `bin_combo_str`

<table>
<thead>
<tr>
<th>1-bit Bin.</th>
<th>2-bit Bin.</th>
<th>3-bit Bin.</th>
<th>4-bit Bin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
<td>001</td>
<td>0001</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>010</td>
<td>0010</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>011</td>
<td>0011</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>0100</td>
</tr>
<tr>
<td></td>
<td></td>
<td>101</td>
<td>0101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>110</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>111</td>
<td>0111</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1010</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1011</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1100</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1110</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1111</td>
</tr>
</tbody>
</table>
Recursion and DFS

- Recursion forms a kind of Depth-First Search

```cpp
// user interface
void binCombos(int len)
{
    binCombos("", len);
}

// helper-function
void binCombos(string prefix, int len)
{
    if(prefix.length() == len)
        cout << prefix << endl;
    else {
        // recurse
        binCombos(prefix+"0", len);
        // recurse
        binCombos(prefix+"1", len);
    }
}
```
Recursion and DFS (w/ C-Strings)

- Recursion forms a kind of Depth-First Search

```c
void binCombos(char* data, int curr, int len)
{
    if(curr == len )
        data[curr] = '\0';
    else {
        // set to 0
        data[curr] = '0';
        // recurse
        binCombos(data, curr+1, len);
        // set to 1
        data[curr] = '1';
        // recurse
        binCombos(data, curr+1, len);
    }
}
```
Generating All Combinations

• Recursion offers a simple way to generate all combinations of \( N \) items from a set of options, \( S \)
- Example: Generate all 2-digit decimal numbers (\( N=2, S=\{0,1,...,9\} \))

```c
void TwoDigCombos(char data[3], int curr)
{
    if(curr == 2)
        cout << data;
    else {
        for(int i=0; i < 10; i++){
            // set to i
            data[curr] = '0'+i;
            // recurse
            TwoDigCombos(data, curr+1);
        }
    }
}
```
Recursion and Combinations

• Recursion provides an elegant way of generating all $n$-length combinations of a set of values, $S$.  
  – Ex. Generate all length-$n$ combinations of the letters in the set $S$={'U','S','C'} (i.e. for $n=2$: UU, US, UC, SU, SS, SC, CU, CS, CC)

• General approach:
  – Need some kind of array/vector/string to store partial answer as it is being built
  – Each recursive call is only responsible for one of the $n$ "places" (say location, $i$)
  – The function will iteratively (loop) try each option in $S$ by setting location $i$ to the current option, then recurse to handle all remaining locations ($i+1$ to $n$)
    • Remember you are responsible for only one location
  – Upon return, try another option value and recurse again
  – Base case can stop when all $n$ locations are set (i.e. recurse off the end)
  – Recursive case returns after trying all options
Exercises

• bin_combos_str
• Zero_sum
• Prime_products_print
• Prime_products
• basen_combos
• all_letter_combos
Another Exercise

• Generate all string combinations of length n from a given list (vector) of characters

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

void all_combos(vector<char>& letters, int n) {
}

int main() {
    vector<char> letters;
    letters.push_back('U');
    letters.push_back('S');
    letters.push_back('C');
    all_combos(letters, 2);
    all_combos(letters, 4);
    return 0;
}
```
RECURSIVE DEFINITIONS
Recursive Definitions

• N = Non-Negative Integers and is defined as:
  – The number 0 [Base]
  – n + 1 where n is some non-negative integer [Recursive]

• String
  – Empty string, ε
  – String concatenated with a character (e.g. 'a'-'z')

• Palindrome (string that reads the same forward as backwards)
  – Example: dad, peep, level
  – Defined as:
    • Empty string [Base]
    • Single character [Base]
    • xPx where x is a character and P is a Palindrome [Recursive]

• Recursive definitions are often used in defining grammars for languages and parsers (i.e. your compiler)
C++ Grammar

• Languages have rules governing their syntax and meaning
• These rules are referred to as its grammar
• Programming languages also have grammars that code must meet to be compiled
  – Compilers use this grammar to check for syntax and other compile-time errors
  – Grammars often expressed as “productions/rules”

• ANSI C Grammar Reference:
  – http://www.lysator.liu.se/c/ANSI-C-grammar-y.html#declaration
## Simple Paragraph Grammar

<table>
<thead>
<tr>
<th>Substitution</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>subject</td>
<td>&quot;I&quot;</td>
</tr>
<tr>
<td>verb</td>
<td>&quot;run&quot;</td>
</tr>
<tr>
<td>sentence</td>
<td>subject verb '.'</td>
</tr>
<tr>
<td>sentence_list</td>
<td>sentence</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>paragraph</td>
<td>[TAB = \t] sentence_list [Newline = \n]</td>
</tr>
</tbody>
</table>

**Example:**

```
I run. You walk. We exercise.
subject verb. subject verb.
subject verb.
```

```
sentence sentence sentence
sentence_list sentence sentence
sentence_list sentence
sentence_list paragraph
```

**Example:**

```
I eat You sleep
Subject verb subject verb
```

**Error**
# C++ Grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>constant</td>
</tr>
<tr>
<td></td>
<td>variable_id</td>
</tr>
<tr>
<td></td>
<td>function_call</td>
</tr>
<tr>
<td></td>
<td>assign_statement</td>
</tr>
<tr>
<td></td>
<td>'(' expr ')'</td>
</tr>
<tr>
<td></td>
<td>expr binary_op expr</td>
</tr>
<tr>
<td></td>
<td>unary_op expr</td>
</tr>
<tr>
<td>assign_statement</td>
<td>variable_id ‘=‘ expr</td>
</tr>
<tr>
<td>expr_statement</td>
<td>‘;’</td>
</tr>
<tr>
<td></td>
<td>expr ‘;’</td>
</tr>
</tbody>
</table>

**Example:**

```
5 * (9 + max);
expr * ( expr + expr );
expr * ( expr );
expr * expr;
expr;
expr_statement
```

**Example:**

```
x + 9 = 5;
expr + expr = expr;
expr = expr;
```

NO SUBSTITUTION
Compile Error!
# C++ Grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>statement</td>
<td>expr_statement</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>compound_statement</td>
<td>'{' statement_list '}'</td>
</tr>
<tr>
<td>statement_list</td>
<td>statement</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Example:

```cpp
while(x > 0) { doit(); x = x-2; }
while(expr) { expr; assign_statement; }
while(expr) { expr; expr; }
while(expr) { expr_statement expr_statement }
while(expr) { statement statement }
while(expr) { statement_list statement }
while(expr) { statement_list }
while(expr) compound_statement
while(expr) statement
statement
```

### Example:

```cpp
while(x > 0) { x--; x = x + 5; }
while(expr)
statement
statement
```
Follow slides are for your own review

END LECTURE
MORE EXAMPLES
Towers of Hanoi Problem

• Problem Statements: Move n discs from source pole to destination pole (with help of a 3rd alternate pole)
  – Cannot place a larger disc on top of a smaller disc
  – Can only move one disc at a time

```
Start (n=3)  Goal (n=3)
```

© 2015 by Mark Redekopp, All Rights Reserved
Observation 1

• Observation 1: Disc 1 (smallest) can always be moved
• Solve the n=2 case:

Start

Move 1 from src to alt

Move 2 from src to dst

Move 1 from alt to dst
Observation 2

- Observation 2: If there is only one disc on the src pole and the dest pole can receive it the problem is trivial.
Recursive solution

• But to move n-1 discs from src to alt is really a smaller version of the same problem with
  – n => n-1
  – src=>src
  – alt =>dst
  – dst=>alt

• Towers(n,src,dst,alt)
  – Base Case: n==1  // Observation 1: Disc 1 always movable
    • Move disc 1 from src to dst
  – Recursive Case:  // Observation 2: Move of n-1 discs to alt & back
    • Towers(n-1,src,alt,dst)
    • Move disc n from src to dst
    • Towers(n-1,alt,dst,src)
Exercise

• Implement the Towers of Hanoi code
  – $ wget http://ee.usc.edu/~redekopp/cs104/hanoi.cpp
  – Just print out "move disc=x from y to z" rather than trying to "move" data values
    • Move disc 1 from a to b
    • Move disc 2 from a to c
    • Move disc 1 from b to c
    • Move disc 3 from a to b
    • Move disc 1 from c to a
    • Move disc 2 from c to b
    • Move disc 1 from a to b
Recursive Box Diagram

Towers Function Prototype

Towers(disc, src, dst, alt)

Towers(3, a, b, c)
  - Move D=3 a to b
    - Towers(2, c, b, a)
      - Move D=2 c to b
        - Towers(1, a, b, c)
          - Move D=1 a to b
    - Towers(2, a, c, b)
      - Move D=2 a to c
        - Towers(1, b, c, a)
          - Move D=1 b to c
    - Towers(1, a, b, c)
      - Move D=1 a to b
      - Move D=1 c to a
      - Move D=1 c to a
      - Move D=1 a to b

© 2015 by Mark Redekopp, All Rights Reserved
Combinatorics Examples

• Given n things, how can you choose k of them?
  – Written as C(n,k)

• How do we solve the problem?
  – Pick one person and single them out
    • Groups that contain Joe ⇒ _______________
    • Groups that don't contain Joe ⇒ _______________
  – Total number of solutions: _______________
  – What are base cases?

Joe

© 2015 by Mark Redekopp, All Rights Reserved
Combinatorics Examples

• Given n things, how can you choose k of them?
  – Written as C(n,k)

• How do we solve the problem?
  – Pick one person and single them out
    • Groups that contain Joe => C(n-1, k-1)
    • Groups that don't contain Joe => C(n-1, k)
  – Total number of solutions: C(n-1,k-1) + C(n-1,k)
  – What are base cases?
Combinatorics Examples

• You're going to Disneyland and you're trying to pick 4 people from your dorm to go with you
• Given n things, how can you choose k of them?
  – Written as C(n,k)
  – Analytical solution: \( C(n,k) = \frac{n!}{k! \cdot (n-k)!} \)
• How do we solve the problem?

[Diagram of people]
Recursive Solution

- Sometimes recursion can yield an incredibly simple solution to a very complex problem
- Need some base cases
  - $C(n,0) = 1$
  - $C(n,n) = 1$

```c
int C(int n, int k)
{
    if(k == 0 || k == n)
        return 1;
    else
        return C(n-1,k-1) + C(n-1,k);
}
```
You are responsible for this on your own since its covered in CS103

C++ LIBRARY REVIEW
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)

- **int strlen(char *dest)**
  - Return 0 if equal, >0 if first non-equal char in str1 is alphanumerically larger, <0 otherwise

- **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

---

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

```cpp
#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)

```cpp
#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
```

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

```cpp
#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
```

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.

• 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 1\textsuperscript{st} prompt, enters “123” at 2\textsuperscript{nd} prompt

```
int x=0;

cout << “Enter X: “;
cin >> x;

X = 0

int y = 0;

cin >> y;

Y = 0

X = 0

X = 512

Y = 0

Y = 123

\text{cin.fail()} is false
```

© 2015 by Mark Redekopp, All Rights Reserved
Understanding Extraction

User enters value “23 99” at 1\textsuperscript{st} prompt, 2\textsuperscript{nd} prompt skipped

```cpp
int x=0;

std::cout << “Enter X: “;

std::cin >> x;

int y = 0;

std::cout << “Enter Y: “;

std::cin >> y;
```

<table>
<thead>
<tr>
<th>X</th>
<th>cin fail() is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 3 9 9 \n</td>
</tr>
<tr>
<td>23</td>
<td>9 9 \n</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th>cin fail() is false</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9 9 \n</td>
</tr>
<tr>
<td>99</td>
<td>\n</td>
</tr>
</tbody>
</table>
Understanding Extraction

User enters value “23abc” at 1st prompt, 2nd prompt fails

```cpp
int x = 0;

cout << "Enter X: ";

int y = 0;

cout << "Enter Y: ";
```

- **X = 0**  
  - **cin =** (value not shown)
  - cin.fail() is **false**

- **X = 23**  
  - **cin =** (value not shown)
  - cin.fail() is **true**

- **Y = 0**  
  - **cin =** (value not shown)

- **Y = xxx**  
  - **cin =** (value not shown)
  - cin.fail() is **true**
Understanding Extraction

User enters value “23 99” at 1st prompt, everything read as string

```cpp
string x;
cout << “Enter X: “;
getline(cin,x);
```

<table>
<thead>
<tr>
<th>X</th>
<th>cin</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>23 99</code></td>
<td><code>EOF</code></td>
</tr>
</tbody>
</table>

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();

istream (cin)
Understanding Extraction

User enters value “23abc” at 1st prompt, 2nd prompt fails

```cpp
int y = 0;
cout << “Enter Y: “;
cin >> y;
cin.ignore(100, ‘\n’);
// doing a cin >> here will
// still have the fail bit set

// now safe to do cin >>
cin.clear();
```
C++ File I/O

• Use <fstream> library for reading/writing files
  – Use the open( ) method to get access to a file
    ```cpp
    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(...)

© 2015 by Mark Redekopp, All Rights Reserved
C++ String Stream

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

```cpp
#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()]

```cpp
#include<iostream>
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

```cpp
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:

• Write an array value
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

```cpp
#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++)
    {
        my_vec[i] = i+50;
    }
    my_vec.push_back(10); my_vec.push_back(8);
    my_vec[0] = 30;
    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.
    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 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)

```cpp
#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

```cpp
#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;
}
```