Dynamic Arrays and Pointers

Dynamic Arrays and Recursion

- Arrays
  - Returning arrays
  - Pointers
  - Dynamic arrays
- Smart pointers
- Vectors

Array Problems

Array Declaration Review

- To declare an array specify the type, its name, and its size in []s
  - `int arr1[10]; //or`
  - `int arr2[] = {1,2,3,4,5,6,7,8};`
    - arr2 has 8 elements
- The size must be a literal or a constant
  - `int arr3[ARR_SIZE];`
- The size cannot be a variable

assuming ARR_SIZE is a constant

const int ARR_SIZE = 10;

Array Indexing Review

- To access an element of an array use the array name and an index

```
int arr[ARR_SIZE];
arr[0] = 1;
```

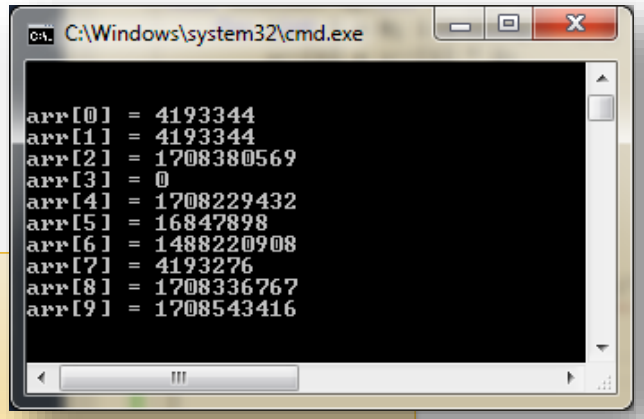
- To iterate through an array use a loop

```
for(int i = 0; i < ARR_SIZE; ++i)
{
   cout << "arr[i] = " << arr[i] << endl;
}
```

assuming ARR_SIZE is a constant and *arr* has been declared and initialized

Initializing Arrays

- If an array is not initialized it will contain garbage values
  - The bit pattern that happens to be stored in the array elements' memory locations




```
int arr[10];

for(int i = 0; i < 10; ++i){
        cout << "arr[" << i << "] = " << arr[i] << endl;
}
```

Array Bounds

- Be careful not to access an array using an index that is out of bounds
  - Less than zero or greater than array size - 1
- The *sizeof* function can be used to find the length of an array
  - But only for static arrays

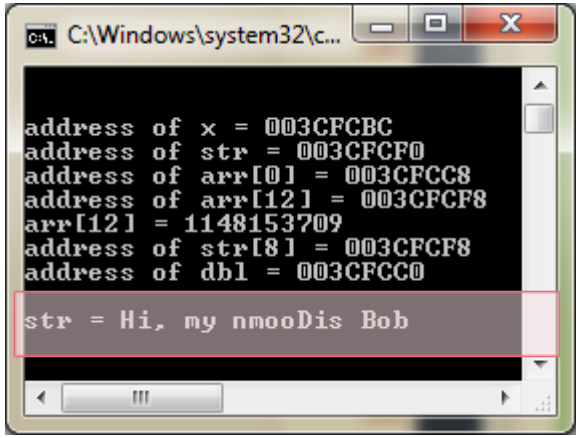
```
for(int i = 0; i < sizeof(arr) / sizeof(int); ++i){
        cout << "arr[" << i << "] = " << arr[i] << endl;
}
```

Using an Invalid Index

```
void arrayBoundsTest()
{
        int x;
        char str[20];
        int arr[10];
        double dbl;

        strcpy(str, "Hi, my name is Bob");
        arr[12] = 1148153709;

        cout << "address of x = " << &x << endl;
        cout << "address of str = " << (void*) str << endl;
        cout << "address of arr[0] = " << arr << endl;
        cout << "address of arr[12] = " << &arr[12] << endl;
        cout << "arr[12] = " << arr[12] << endl;
        cout << "address of str[8] = " << (void*) &str[8] << endl;
        cout << "address of dbl = " << &dbl << endl << endl;
        cout << "str = " << str << endl;
}
```



Arrays and Functions

- Arrays can be passed to functions
  - The parameter specifies an array
    - `int sumArray(int arr[], int size) { … }`
    - It is common to pass the size of the array
  - Array arguments are passed as normal
    - `int sum = sumArray(arr, ARR_SIZE);`
- Arrays can also be returned from functions
  - But *not* like this:
    - `int arr[] foo() { … }`

Arrays are Pointers

- If an array is passed to a function, changes made to it within the function will persist
  - Because an array variable contains the address of the first element of the array
    - Called a *pointer*
  - A static array variable is a *constant pointer* to the first element of the array
  - Array parameters give the address of the array

Returning Arrays

- Let's assume that we would like to write a function that returns an integer sequence
  - Like {1,2,3,4,5} or {11,12,13}
  - The return type would seem to be `int[]`
- But this implies that the function is returning a *constant* (array) pointer
  - Which presumably would be assigned to another *constant* pointer
  - Which is illegal – why?

Stack Problems

- There is one big issue with stack memory
  - Because memory is allocated in sequence it is not possible to change the *size* in bytes of a variable
- This isn't a problem that applies to single *base typed* variables (*int*, *double* etc.)
  - There is no need to change the number of bytes required for single variables
- But we might want to change an array's size
  - To store more things in it

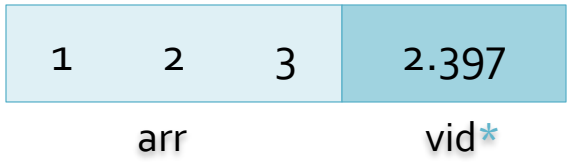
Changing an Array's Size

```
int main(){
        int arr[3];
        arr [0] = 1;
        arr [1] = 2;
        arr [2] = 3;
        double vid = 2.397;
        arr = sequence(5, 8);
```

```
int[] sequence(int start, int end){
        int arr[end - start + 1];
        int i;
        for(i=0; i <= end - start; i++){
            arr[i] = start + i;
        }
        return arr;
}
```

statements in red are illegal*

stack memory



*very important double ☺

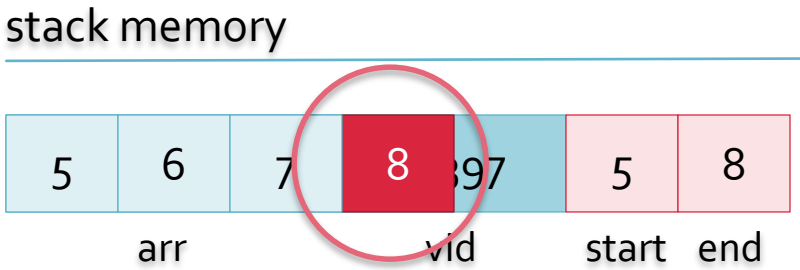
* the second line, a variable length array, may be legal using some compilers

Changing an Array's Size

```
int main(){
        int arr[3];
        arr [0] = 1;
        arr [1] = 2;
        arr [2] = 3;
        double vid = 2.397;
        arr = sequence(5, 8);
```

```
int[] sequence(int start, int end){
        int arr[end - start + 1];
        int i;
        for(i=0; i <= end - start; i++){
            arr[i] = start + i;
        }
        return arr;
}
```

statements in red are illegal



This is a problem, we've just corrupted the first 4 bytes of *vid*

... which is why you can't do this ...

Returning an Array

- To return an array you return a non-constant pointer of the appropriate type
  - e.g. `int* sequence(int start, int end)`
- The new array should be assigned space in *dynamic memory* in the function
  - Dynamic memory is a section of main memory that is separate from stack memory
  - We use pointers to access this area of main memory

Introducing Pointers

- A pointer is a special type of variable
  - That stores an *address* rather than a *value*
  - They are called pointers as they can be considered to *point to* a variable
- It is necessary to record the type of data that a pointer variable points to
  - So that the appropriate operations can be performed on the value it points to

Pointers

Pointers and Arrays

- The solution to returning an array is to return a pointer to an array in dynamic memory
- What exactly is a pointer?
  - And how do they operate?
- What is dynamic memory?
  - How is it used?
  - How does it compare to automatic memory
    - How is it allocated space?
    - When is that space released?

Pointers

- A pointer is a special type of variable
  - That stores an *address* rather than a *value*
  - They are called pointers as they can be considered to *point to* a variable
- It is necessary to record the type of data that a pointer variable points to
  - So that the appropriate operations can be performed on the value it points to

Pointers and Types

- Pointers store addresses
  - Addresses are always the same size on the same system
- So why do we have to say what type of data is going to be pointed to?
  - To reserve enough space for the data and
  - To ensure that the appropriate operations are used with the data

Declaring a Pointer

- Pointer variable are identified by an * that follows the type in the declaration
  - `int * p;`
- This declares a variable called *p* that will point to (or refer to) an integer
- Note that the type of a pointer is *not* the same as the type it points to
  - p is a *pointer to an int*, and *not* an int

Declaring a Pointer

- Previously I declared a pointer like this
  - `int * p;`
    - The spaces are not necessary
- You can do this
  - `int *p;`
- Or this
  - `int* p;`
- Or even this
  - `int*p;`

What does this declare? `int *p, x;`

But this is kind of ugly!

Pointers and Values

- The operation shown below is unsafe
  - `int x = 12;`
  - `int *p = x;`

This is not a good thing to do and will result in a compiler warning or error

- Remember that the type of *p* is an address of an *int*, and not an *int*
  - Addresses are actually whole numbers but assigning arbitrary numbers to them is a bad idea
  - Since a programmer is unlikely to know what is stored at a particular memory address

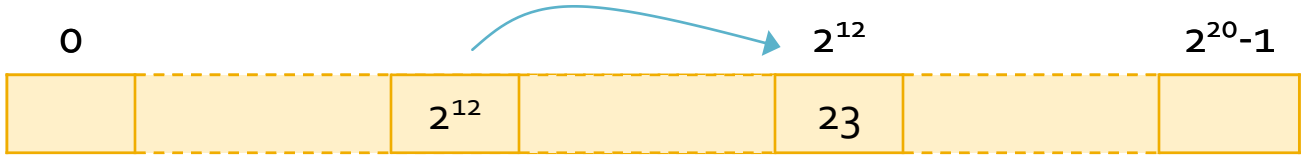
Address Operator

- Pointers can be assigned the address of an existing variable
  - Using the address operator, &
  - In this way it is possible to make a pointer refer to a variable
- In practice this is not something that happens often
  - But it is useful to illustrate pointer behaviour

Main Memory and Pointers


```
int x = 23; //located at byte 2^12
```

```
int* p = &x; //stores the address of x - byte 2^12
```



p stores the address of *x*

& is the *address of* operator

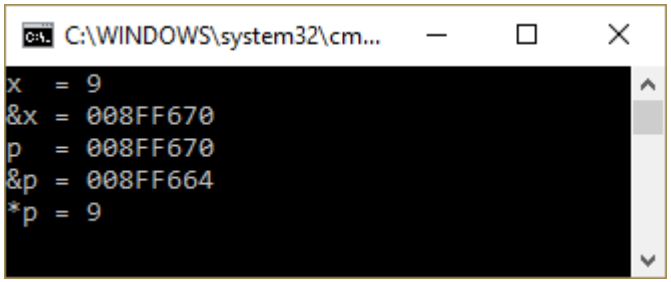
Using Pointers

- Pointers can be used to access variables
  - But only after they have been assigned the address of a variable
- To change the value of a variable a pointer points to the pointer has to be *dereferenced*
  - Using the * operator which can be thought of meaning *the variable pointed to*

Pointer Assignment

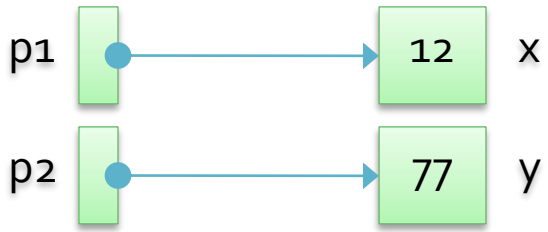
```
int x = 105;
int *p = &x; //assign p the address of x
*p = 9; //dereferences p, assigns 9 to x

cout << "x  = " << x << endl;
cout << "&x = " << &x << endl;
cout << "p  = " << p << endl;
cout << "&p = " << &p << endl;
cout << "*p = " << *p << endl;
```



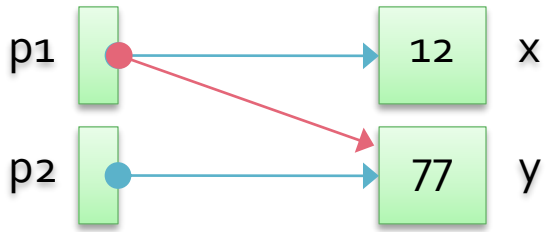
Pointers and Assignment

```
int x = 12;
int y = 77;
int *p1 = &x; //assign p1 the address of x
int *p2 = &y; //assign p2 the address of y
```



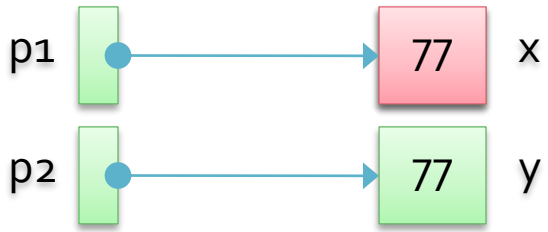
Pointers and Assignment

```
int x = 12;
int y = 77;
int *p1 = &x; //assign p1 the address of x
int *p2 = &y; //assign p2 the address of y
p1 = p2; //assigns the address in p2 to p1
```



Pointers and Assignment

```
int x = 12;
int y = 77;
int *p1 = &x; //assign p1 the address of x
int *p2 = &y; //assign p2 the address of y
*p1 = *p2;
```



Why Use Pointers?

- In practice we don't often use pointers like the preceding examples
- Pointers can be used to allow functions to change the value of their arguments
- They are also key to managing memory for objects that change size during run-time
  - Such objects are allocated space in another area of main memory – in *dynamic memory*

Dynamic Memory

Returning Data from a Function

- Many functions return data
  - This data is typically returned by value
  - Consider `int x = foo();`
    - foo presumably returns some integer value
    - Which is assigned to the memory space of variable x
- Array variables are pointers to arrays
  - Returning an array therefore entails returning an address
    - Which cannot be assigned to a constant pointer

Returning Data from a Function

- Consider a function that declares an array on the stack and returns its address
  - Recall that stack memory for a function is released once the function terminates
    - That is, it becomes available for re-use
  - Therefore the contents of such an array may be over-written with other data
    - As part of the normal processing of stack memory
- See class example …

Dynamic Memory Introduction

- The lifetime of variables allocated in automatic (stack) memory is fixed
  - To the duration of the variables function call
  - The size of such variables is dependent on type and is also fixed
- It is possible to fix the space and lifetime of a variable at run-time
  - Allowing the variable to exist for as long as it is needed and
  - To avoid wasting space where the space required for the variable may vary

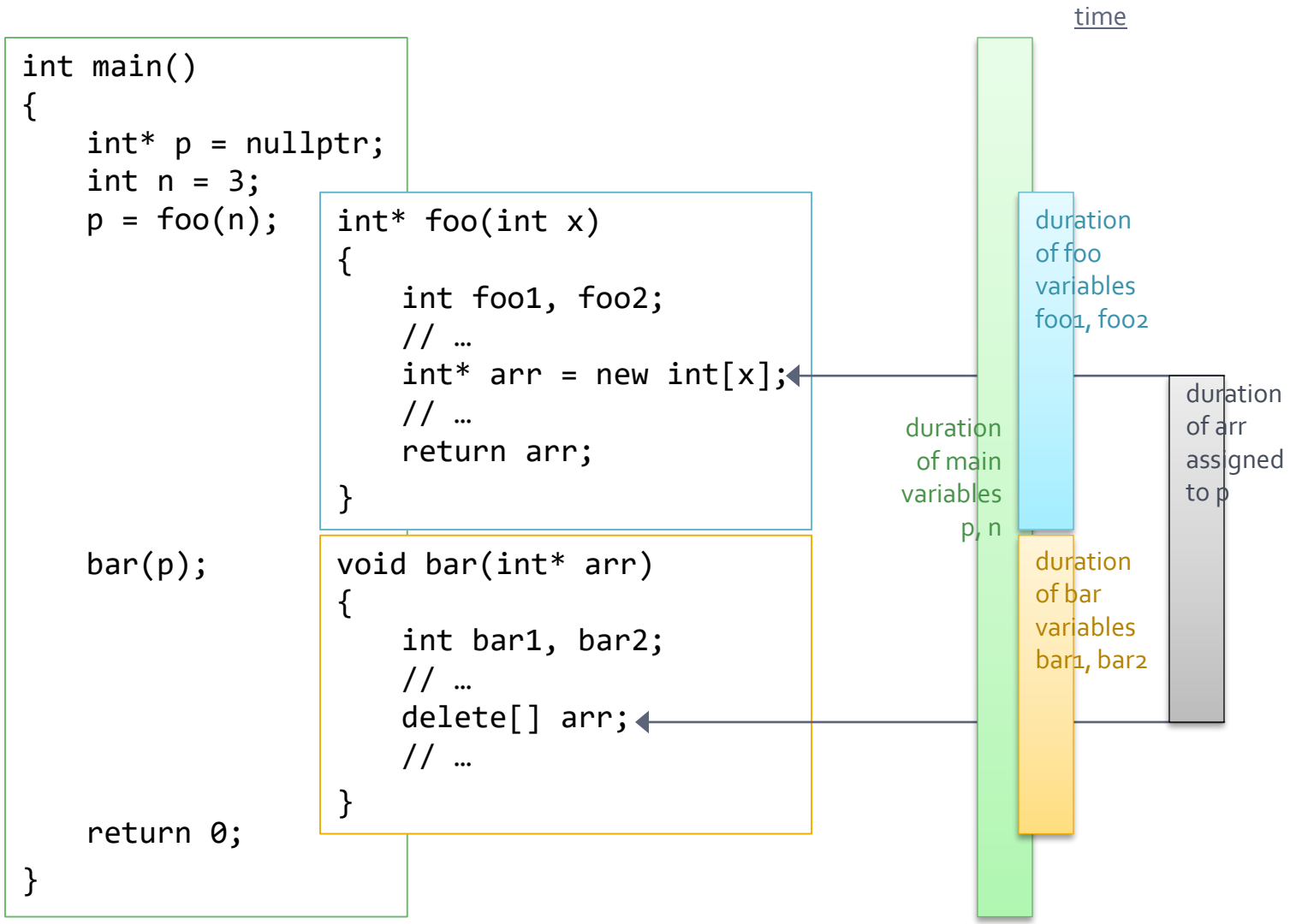
Dynamic Memory

- Dynamic memory is allocated from a different area of memory than the stack
  - This area of memory is referred to as the *free store* or the *heap*
  - Like stack memory the size is fixed, and is, of course, finite
- Dynamic memory can be allocated and de-allocated explicitly
  - Using *new* and *delete*

Allocating Dynamic Memory

- Dynamic memory can be allocated at runtime
  - Using *new* to request memory
    - *new* determines the number of bytes to be allocated by reference to the type of data being created
- Memory allocated with *new* remains allocated until it is released
  - It is not limited to the lifetime of the block in which it is allocated
  - Memory is released by calling *delete*

Variable Duration



More About Pointers

- *new* allocates dynamic memory and returns the address of its first byte
  - Which should be assigned to a pointer variable
- Dynamic memory is often used to create arrays whose size is only known at run time
  - For example create an array of ten integers
    - `int* arr = new int[10];`
    - The address returned by *new* is assigned to *arr*

Creating Arrays in Dynamic Memory

- Step 1 – declare a pointer of the desired type
  - `int*` if we are going to store integers
  - `float*` if we are going to store floats
  - `string*` if we are going to store strings
- Step 2 – assign the pointer the address of the desired amount of space using *new*
  - `int* arr = new int[100];`
    - An instruction to make space for 100 integers in dynamic memory and store the address in *arr*

Allocating Space

- The keyword *new* assigns space in dynamic memory
- We can assign new arrays to existing pointer variables
  - `int* arr = new int[1000];`
  - `arr = new int[100000];`
  - This second line causes a problem since we have not de-allocated the originally assigned memory
    - The space should be de-allocated using *delete*

Arrays in Dynamic Memory

- Arrays assigned to pointers may be used just like regular (static) arrays
  - The elements can be accessed using an index
    - Without dereferencing the pointer
  - Recall that static array variables are also pointers
    - That are constant
- Unlike static arrays, dynamic arrays space must be explicitly de-allocated
  - Using *delete*

Sequence – Corrected

Here is a function that correctly returns a sequence of values to a pointer

returns a pointer to an array

```
int* sequence(int start, int end)
{
    int size = end - start + 1;
    int* arr = new int[size];
    for(int i = 0; i < size; ++i){
        arr[i] = start + i;
    }
    return arr;
}
```

allocates space in dynamic memory for the array

the function would be called like this

```
int* p;
int size = 11;
p = sequence(10, 20);
cout << p[3] << endl;
// prints 13
```

The elements of the array that *p* points to are indexed just like a regular array

Re-using Array Variables

```
int* arr = new int[10];
// ... do stuff with arr
// ... and make a new, larger array
arr = new int[1000];
```

allocates 40 bytes in the heap

allocates another 4000 bytes in the heap, it does not re-use the 40 bytes that were previously allocated

the original 40 bytes cannot be accessed, since the program no longer records the address

however, they are also unavailable for reuse, thereby causing a *memory leak*

Memory Leaks

- A memory leak occurs when dynamic memory is allocated but is not de-allocated
  - When it is no longer required
- Dynamic memory that is not de-allocated is unavailable until the program terminates
  - Or even longer in some older operating systems
- Large memory leaks may affect the performance of applications

Freeing Dynamic Memory

- Any dynamic memory that is allocated by a program should be de-allocated
  - By calling *delete*
    - *delete* takes a single argument, a pointer to the memory that is to be de-allocated
    - If the pointer refers to an array use *delete[]*
- Every use of *new* should be matched by a call to *delete*
  - When the allocated dynamic memory is no longer required

Why?

- Other languages simplify the use of automatic and dynamic memory
  - So that programmers are not responsible for deleting unneeded dynamic memory
    - Known as *automated garbage collection*
  - Often achieved by removing the choice of where to store data
- While memory allocation in C++ is more complex it allows for greater flexibility
  - Which is useful for low level programming
- Modern C++ provides smart pointer syntax that removes the need to use *new* and *delete*

Vectors and the STL

Dynamic Arrays

- A dynamic array is an array that increases size as necessary
  - Usually by doubling the number the size of the array when it is full
  - The underlying array is stored in dynamic memory
- Dynamic arrays would normally be created as a class in C++
  - We do not cover classes in CMPT 130

Dynamic Array Insertion

- The process for inserting values into a dynamic array is like this
  - If the array is full:
    - Assign the address of the array to a temporary pointer
    - Assign a new array of twice the size to the array pointer
    - Copy the contents of the original array to the new array
    - Delete the original array
  - Insert the new value into the next free element
  - And increment the count of the values

Arrays

- Arrays are somewhat fiddly to use
  - We have to distinguish between arrays on the stack and arrays in dynamic memory
    - Static arrays can't change size
    - We have to be responsible for the allocating and de-allocating memory to arrays in dynamic memory
  - We need to record their size separately
    - And pass this value to any functions that process the array
- These issues do not exist because arrays are somehow bad
  - It is because they are very low level structures
  - That are used to create more sophisticated containers

The STL

- C++ has a set of classes and functions that implement common algorithms and containers
  - Called the Standard Template Library
    - Or STL for short
- These algorithms and classes are *templates* that can be used with data of different types
  - Templates are beyond the scope of CMPT 130
    - But are covered in CMPT 135
  - We will look at one function and one container from the STL

Vectors

- The C++ Standard Template Library (STL) provides a number of container template classes
  - A container class is used to store data
    - Different containers organize data in different ways
    - To optimize certain operations
- The vector class implements a dynamic array
  - The underlying array increases size as required
    - A programmer using a vector is not responsible for dealing with the allocation and deallocation of dynamic memory
- *#include* the <*vector*> library to use vectors

Vector Basics

- Vectors are used in a very similar way to arrays
  - Elements can be accessed using indexes
- In addition to being used like arrays, vectors have some useful methods
  - The size() method records the size of the vector
  - There are a variety of methods for insertion and removal of values

Creating a Vector

- Vectors are implemented in the *vector* library
  - `#include <vector>`
- Vectors can contain data of any type
  - The type must be given in <>s when the vector is declared
    - `vector <string> name;`
    - `vector <int> scores;`
    - `vector <Student> class;`
- The <>s contain *template arguments*
  - In this case, the name of the type to be stored

Vectors and Size

- Vectors may be given a size when declared
- The vector elements are given default values
  - 0 for numbers
  - " " for strings (the empty string)
- You can provide your own default value
  - vector <double> v(10000, 2.1);

size

value

- You cannot refer to nonexistent elements
  - v[12001] = 23.67;

error because there is no such element

Is v[10000] = 23.67; an error?

Yes!

Vectors and Types

- Vectors can contain data of any type
  - The type must be given in <>s when the vector is declared

```
vector <string> name(3);
name[0] = "Bob";
name[1] = "Susan";
name[2] = "Kelly";
```

- The <>s contain *template arguments*
  - In this case, the name of a type

Using a Vector

```
// Create an empty vector of doubles
vector <double> v;
// Insert some values
v.push_back(1.2);
v.push_back(3.7);
v.push_back(7.4);
// check how big the vector is
cout << "v's size is " << v.size();
// print the contents of the vector
for(int i=0; i < v.size(); +i){
  cout << v[i] << endl;
}
```

Vectors are template classes that require a template argument when declared

The template argument is in <>s and appears between the type name (vector) and variable name (v)

prints 3

Vectors can be accessed using an index just like a regular array

Inserting Elements

- Sometimes we don't know how big a vector should be
  - e.g. because the size is dependent on user input
- Vectors can be created empty
  - By not specifying any size
- New elements can be created
  - Using the vector method *push_back*()
  - Note that *push_back* can be used regardless of the starting size of the vector

The push_back() Method

```
// Create an empty vector of doubles
vector <double> v;

// And insert some values
v.push_back(1.2);
v.push_back(3.7);
v.push_back(7.4);

// Now let's check how big the vector is
cout << "v's size is " << v.size();
```

push_back and *size* are methods or member functions, so must be preceded by the name of the object and a dot (*v.*)

prints 3

Member Functions

- Vectors have *member functions*
  - Functions that belong to an object
  - Also known as methods
  - Many other objects have methods as well
- To use a member function
  - Specify the object that the function belongs to
  - By typing it's name and a dot .
    - The member operator
  - And then call the member function

Vectors and Loops

- It is easy to use a for loop to traverse through all of the elements of a vector
- For example, print all of *v*'s values

```
for(int i=0; i < v.size(); i++){
   cout << v[i] << endl;
}
```

- This is where the new C++ for loop is useful

```
for (int x : v) {
    cout << x << endl;
}
```

STL Sort

- In reality programmers don't write sort functions every time they want to sort something
  - They use a library sorting function
  - The STL has a sort function in the algorithm library
    - #include <algorithm>
- The sort function can be used to sort pretty much any container
  - Including vectors and standard arrays
  - It needs to know the start and end points
    - And has an optional argument for how values should be compared

Using Sort

- Suppose that we have a vector, *v* that we want to sort
  - `sort(v.begin(), v.end());`
  - `.begin()` and `.end()` are *iterators* that point to the start and end of the vector
    - An iterator is a generalization of a pointer
- Now suppose that we want to sort an array of size *n*
  - `sort(arr, arr+n);`
  - The function takes two pointers, one to the start, and one to one past the end of the array

Smart Pointers

C++ 11 Pointers

- The C++ 11 standard introduced new pointer types
  - Known as smart pointers
- These pointers are responsible for the memory management of the data they point to
  - That is, it is not necessary to call *new* or *delete*
- There are three types of smart pointer
  - unique_ptr
  - shared_ptr
  - weak_ptr

Discussion of these pointers is out of the scope of this course, see cplusplus or MSDN for reference

Or take CMPT 135 …