Software Design and Analysis for Engineers

by
Dr. Lesley Shannon
Email: lshannon@ensc.sfu.ca
Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 10
Date: November 2, 2015

What we're learning in this Slide Set:

- Test Results
- Templates

Textbook Chapters:

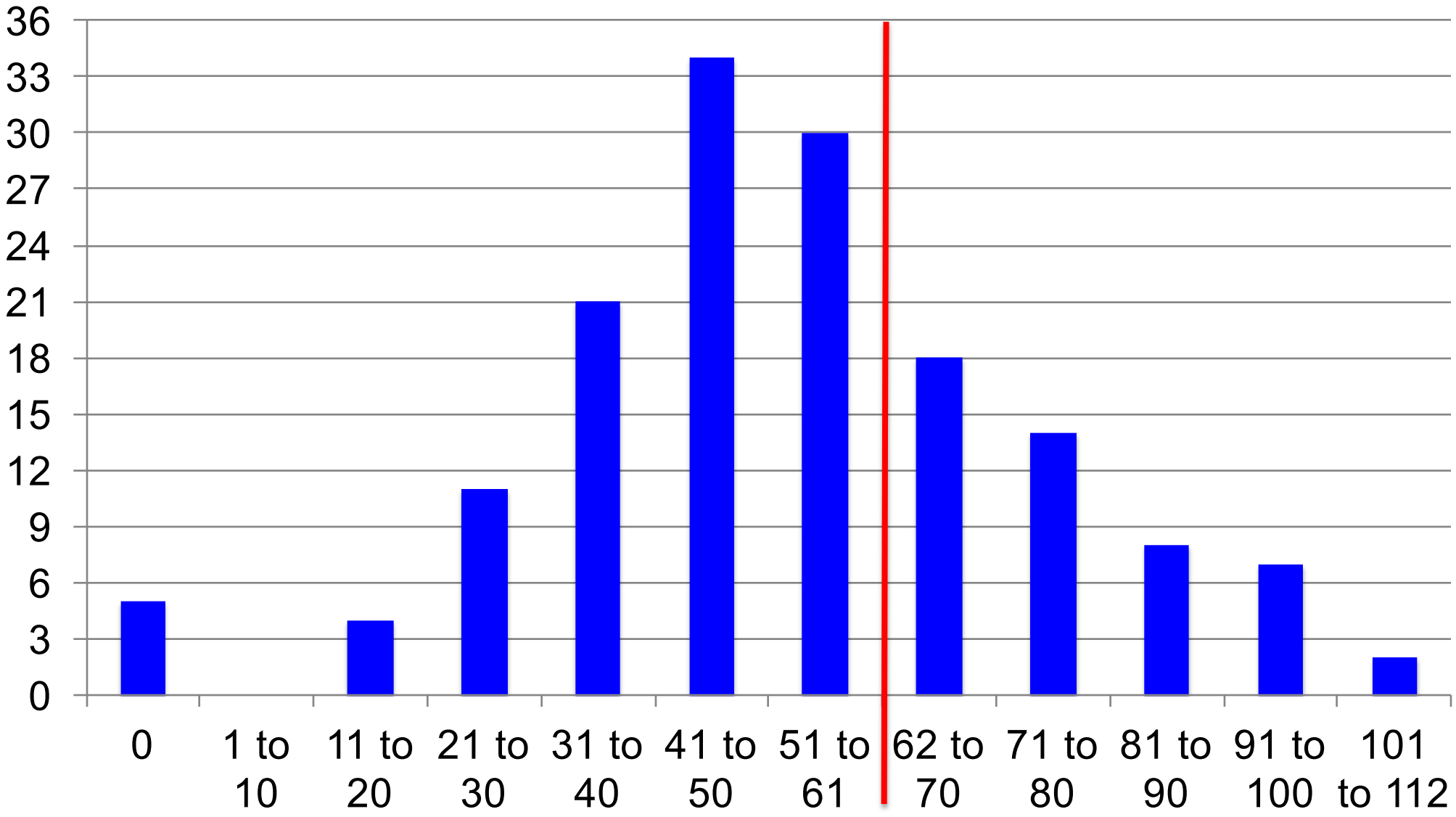
Relevant to this slide set:

- Chapter 17
- Reminder from Chapter 8.3

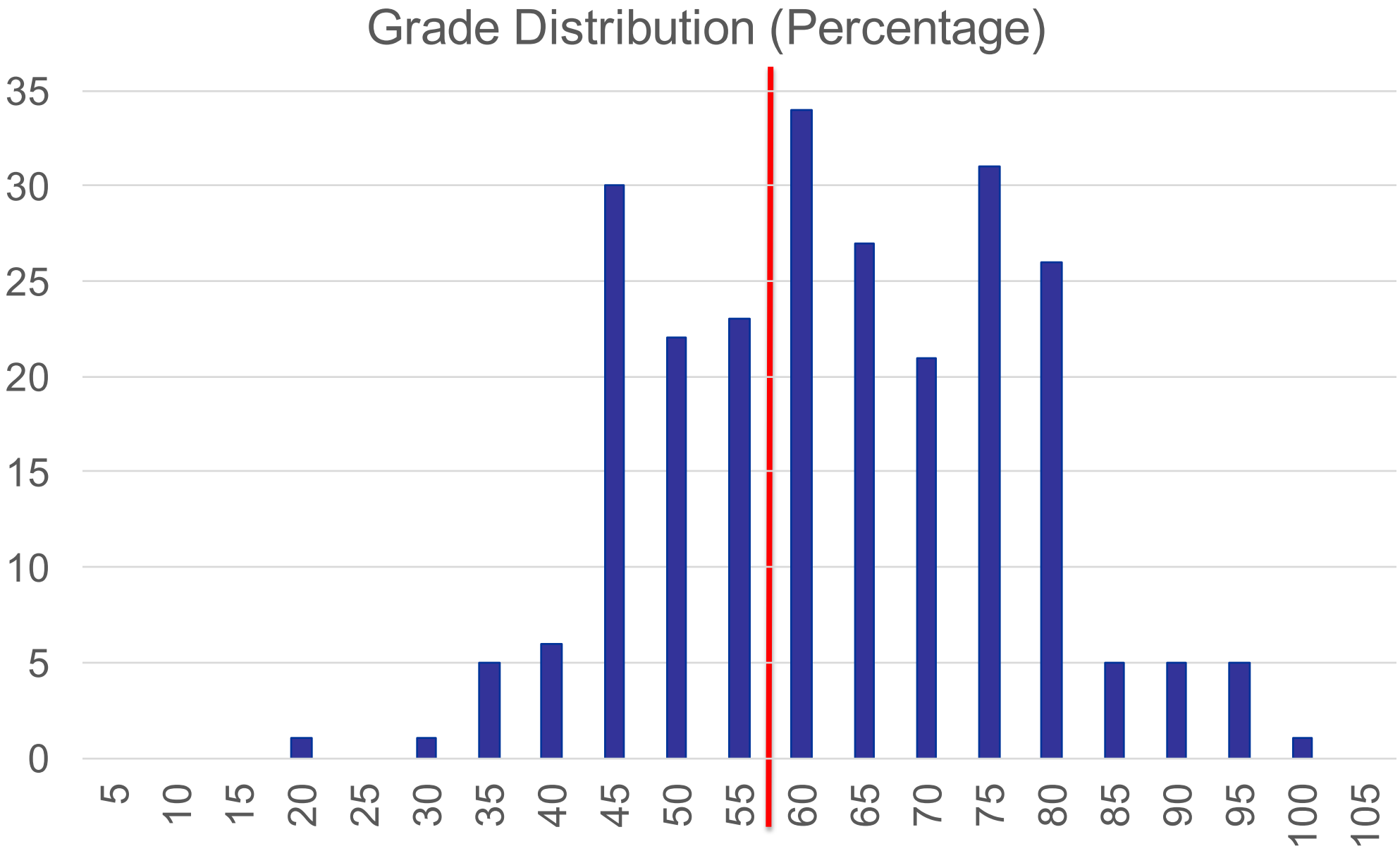
Coming soon:

- Chapter 18
- Discrete Math

Last Year's Test Results



This Year's Test Results



What this means:

- 36% of the class received <55% (less than 53.5/97)
- In other words 88 people got an F or a D compared to last year's 105 (out of 160)

Max Grade: 97%

Min Grade: 16.5%

Average: 60.5%

Overall the class did much better, so I'm really happy

NOTE: This test was not hard and the final exam will be harder.

Translating your grade into a letter grade:

A+: 93-100% (3 people)

A: 85-93% (8 people)

A-: 80-84% (8 people)

B+: 75-79% (26 people)

B: 70-74% (31 people)

B-: 65-69% (21 people)

C+: 60-64% (27 people)

C: 55-59% (34 people)

D: 50-54% (27 people) [11% of the class]

F: <50 (61 people) [25% of the class]

Midterm

Mutator functions change the value of class member fields.

C is NOT a declarative programming language (imperative)

Unions can be hierarchical.

Structures CAN be hierarchical

Structures can have both member functions and member variables in C++.

Midterm

The unnamed namespace and global namespace are NOT accessible from anywhere.

It is NOT impossible to write an imperative styled program using C++.

A derived class is also known as a child class.

The comparison operator is NOT automatically defined for user defined types.

A member function can return a value that is a class type.

Midterm

Private member variables DO exist in derived classes.

Destructors should NOT be private member functions.

int is an Abstract Data Type.

Class member functions CAN use recursive calls.

By default, the assignment operator is overloaded in a user-defined types.

Midterm

Stacks are LIFO Data Structures.

Queues are NOT LIFO Data Structures. (FIFO)

The only advantage of implementing ADTs using separate .cpp files from an application (i.e. separate compilation of ADTs) is NOT the reduced compilation time for updates to application code.

The unnamed and global namespace are both unnamed.

Midterm

Arity is the number of parameters in a function call.

Overloaded functions do not need to have parameters of different types.

Friend functions have access to a class' private data members.

Midterm

Objects have two aspects: State and Behaviour

A class is the data type; the object is the instantiation of that type:

```
myClass  my_object;  //my_object is an instantiated object of
                     //myClass type with a default constructor
```

Midterm

Similarities between data types and abstract data types:

Both can be types for program defined objects/variables.

Differences between data types and abstract data types:

Abstract data types must provide a defined interface (including comments); users are not allowed to “see”/manipulate private member fields; the implementation of how the data is stored is hidden.

Midterm

Constructors must:

1) Have the same name as the class.
2) Not return a value
3) Be a public member function.

3 characteristics of object oriented programming:

1) Encapsulation= Provides information hiding & abstraction
   (e.g. classes-> public versus private members)
2) Inheritance= Code should be reusable/extendable
   (e.g. allowed to derive classes)
3) Polymorphism= A single name may have multiple meanings
   (e.g. overloading functions **virtual functions**)

Midterm

Similarities between public and private member functions:

Both functions are able to manipulate private data fields

Differences between public and private member functions:

Only public functions can be accessed outside of the class; private functions can only be accessed by other class member functions.

Similarities between “.” and “::”:

Both are used to define the scope of a function/data member field.

Midterm

Differences between “.” and “::”:

“::” is used as part of a function definition to define locality; whereas “.” is used to access/execute a member field function

(different order of priority (:: before .).

Example: my_struct.data = 10;

//my_struct is a struct with member field data that is assigned //10

Example: myclass::myclass(void)

//A declaration for the default constructor of myclass.

Midterm

Similarities between Copy Constructors and Default Constructors:

Both are used to create and initialize objects of a class type.

Differences between Copy Constructors and Default Constructors:

Default constructors have no parameters, whereas copy constructors must have a parameter that is of the same type as the class itself.

Midterm

Pointers are always the same size as pointers ARE addresses and address sizes are independent of variable/object memory requirements.

Default constructors are created by you OR the compiler. The compiler will automatically create a default constructor as long as the user does not define a constructor.

Default constructors are needed to create an object with no initialization parameters. For example:

```
my_midterm my_test; //Instantiates a my_midterm object
                    //called my_test.
```

Midterm

You need to be able to explicitly make constructor calls to create dynamic memory.

Array Pro: Fast singular access of any value at any index

Array Con: Difficult/Impossible to remove data from list (need additional memory to represent “invalid” data locations for *all* array members).

Linked List Pro: Able to dynamically shrink/grow list for exact data requirements; Can easily insert data at any location as a singular operation

Linked List Con: Requires more memory than an array of the same size: memory to store data *and* memory to provide connectivity

Midterm

You need a copy constructor when an object has member fields with dynamically allocated memory AND the object will be passed by value as part of function calls (e.g. overloaded assignment operator).

The reason you need a copy constructor is for any behaviour of data storage the compiler cannot know about – for example, when dynamic memory is allocated with in the object. So when you create a copy of an object, by default what you are copying are the member fields of the object (i.e. the addresses and not what the pointers point to- or a shallow copy).

Midterm

Example: Any example that shows what goes wrong without a copy constructor and how using a custom copy constructor solves the problem.

For example, say you are only copying the pointers to dynamically allocated memory as in a default assignment operator a=b. When you manipulate the contents of object a (i.e. what the pointers point to), it will also manipulate the contents of object b (because they point to the same content).

Furthermore, if you destroy a, then b will be left with dangling pointers.

Midterm

You harden the separation of the interface and implementation by putting them in separate files (the interface in a .h file and the implementation in a .cpp file).

The interface comprises:

- The public member functions of the class, and
- The comments that tell the user how to use the public member functions.

Midterm

```
struct fractions
{
        public:
        char fixedpoint;                //How many bytes = 1
        float exp_fraction;             //How many bytes = 4
        double long_exp_fraction;       //How many bytes = 8
};
```

Total memory needed for variables of type fraction: 16

Reason: An additional 3 bytes is needed after the char field to ensure memory alignment of the exp_fraction and long_exp_fraction data fields.

Midterm

```
struct fractions
{
        public:
        char fixedpoint;                //How many bytes = 1
        float exp_fraction;             //How many bytes = 4
        double long_exp_fraction;       //How many bytes = 8
};
```

If I didn't include the "public" label inside of the structure definition, the member variables would still be public as all member fields in a struct are public by default.

Midterm

The problem with the code is the while loop:

```
while (locate !=array[current_index])//Search the entire array until you find
                                                    //the value locate.
        current_index++;
```

This code could attempt to access memory outside of the array as the exit case is wrong.

To fix it:

```
while ((locate !=array[current_index]) && (current_index <
MAX_ARRAY_SIZE))
 //Search the entire array until you find the value locate or reach the end of
//the array (MAX_ARRAY_SIZE-1).
```

Midterm

Criteria 1: No possiblity for infinite recursion (NOT A LOOP)

Sol'n: Decrement n (by 1 and two respectively) for each success recursive call.

Criteria 2: There are stopping/base cases that will calculate the correct values.

Sol'n: F(0) = 0; F(1) = 1

Criteria 3: The results of the recursive calls are correct meaning/proving that the overall (general) case is correct.

Sol'n: Our recursive case should calculate:

$$F(n) = F(n-1) + F(n-2)$$

Midterm

```
int Fibonacci (int n)
{
    if (n < 0)              //Error case- anything less than 0 is illegal
        exit(1);
    else if (n < 2)         //Base case: return F(0) = 0 and F(1) = 1
        return 1;           //This is an error; should be: return n;
    else                    //This is a recursive call…
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

Midterm

You might chose to use an iterative solution instead of a recursive solution because:

1) The recursive solution will be much slower. It will generally have to make numerous recursive calls, which will be much slower than running a single loop iteratively. Furthermore, most Fibonacci values will require more recursive function calls than loop iterations.
2) If the max Fibonacci value that may be calculated is extremely high (or unknown), you may run out of memory and crash the program as each successive function call will store a new stack frame on the stack and it may not be big enough.

Midterm

```
int Fibonacci (int n)
{
    int i, f_n, f_n_1, f_n_2; // Counter, f(n), f(n-1), f(n-2)
    if (n < 0)                  //Error case- anything less than 0 is illegal
        exit(1);
    else if (n < 2)             //Base case: return F(0) = 0 and F(1) = 1
        return 1;               //This is an error; should be: return n;
    else                        //This is the iterative case replacing recursion
    {
        for (f_n_2 = 0, f_n_1 = 1, i = 2; i<=n; f_n_2 = f_n_1, f_n_1 = f_n)
            f_n = f_n_2 + f_n_1;
        return f_n;
    }
}
```

And now, templates….

Templates

Some functions are extremely generic and we want to be able to use them for multiple types. For example, this swap function:

```
void swap_values(int& variable1, int& variable2)
{
    int temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

It is useful for swapping integers, but …

Templates

It is equally useful for swapping characters:

```
void swap_values(char& variable1, char& variable2)
{
    char temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Note that the only thing to change in these two pieces of code is the types being swapped.

Templates

As such, even though we can write these overloaded functions for the different types, what really want is something that can be generically applied to any type:

```
void swap_values(Type_Of_The_Variables& variable1,
                 Type_Of_The_Variables& variable2)
{
    Type_Of_The_Variables temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Let's remind ourselves of the template class vector (from Section 8.3)

Vectors

Vectors are basically arrays that can grow and shrink in size (unlike actual arrays).

They do this by allocating a “capacity” amount of memory, but only letting you access a “size” amount of memory.

- Once you want to store more data than the capacity allows, the vector is allocated more memory to increase its capacity.

Let’s see what this looks like.

Vectors

You declare a variable v for a vector with the base type int using: `vector<int> v;`

The <Base_Type> notation with the vector allows you to substitute the type for the vector (similar to arrays)

You can use any type (including class types).

Vectors are indexed starting at ‘0’ (same as arrays).

Vectors

You can use the square brackets notation to read or change any previously initialized element in a vector.

- You ***cannot*** use it to initialize an element.

```
v[i] = 42;
cout << "The answer is " << v[i];
```

You cannot use the above statement to initialize v[i].

However, once it has been initialized, you can use square bracket notation.

Vectors

To add an element to an index position of a vector for the first time, you normally use the member function push_back

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

This adds elements to a vector in order of positions, where the position is dictated by the next available position that is currently unused.

In C++11 you can also initialize a vector similar to an array:

```
vector <double> sample = {0.0, 1.1, 2.2};
```

Vectors

The vector template also has a size member function that can be used to determine how many elements are in a vector.

Assuming the declaration here

```
vector <double> sample = {0.0, 1.1, 2.2};
```

sample.size() returns 3.

Vectors

You can use this to output all the elements stored in the array.

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

Note: sample.size() returns an unsigned int.

- So the compiler will need to use a coercion function to convert it to an int in the above example and
  - You may see a warning (which you can get rid of by type casting or declaring 'i' as an unsigned int)

```
for (unsigned int i = 0; i < sample.size( ); i++)
    cout << sample[i] << endl;
```

Let's see an example…

Source code

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

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " <<v.size( ) <<endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i <v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

Sample output

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8
```

Vectors

There is a vector constructor that takes one integer argument and initializes the number of positions specified in the argument

```
vector<int> v(10);
```

This initializes the first ten elements to zero for vector ‘v’

- In general, when you use the constructor with an integer argument, the vectors values are initialized to the “zero” of that type.
- If the base type is a class, the default constructor is used for initialization.
- To initialize the 11th position in the vector, you would use push_back.

Vectors

The vector definition is given in the library vector in the std namespace:

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

Note: if you try to use square brackets to set a value for v[i], where i >= v.size() (i.e. `v[i] = n;`), you may not get an error message, but your program won't work properly

Summary

Vectors

Vectors are used very much like arrays are used, but a vector does not have a fixed size. If it needs more capacity to store another element, its capacity is automatically increased. Vectors are defined in the library <vector>, which places them in the std namespace. Thus, a file that uses vectors would include the following (or something similar):

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

The vector class for a given *Base_Type* is written vector <*Base_Type*>. Two sample vector declarations are

```
vector<int> v; //default constructor
               //producing an empty vector.
vector<AClass> record(20); //vector constructor
                           //for AClass to initialize 20
                             elements.
```

Elements are added to a vector using the member function push_back, as illustrated below:

```
v.push_back(42);
```

Once an element position has received its first element, either with push_back or with a constructor initialization, that element position can then be accessed using square bracket notation, just like an array element.

Other Vector Notes

The assignment operator does an element-by-element assignment to the vector on the LHS of the assignment operator

- It will increase capacity if needed and reset the size of the vector on the LHS of the assignment operator.
- Therefore, if the provided (overloaded) assignment operator of the base type makes an independent copy of the element of the base type, then the assignment operator will make an independent copy.
- Therefore, the assignment operator on a vector is only as good (or bad) as that of the base type.

HINT: Another reason you need copy constructors.

Other Vector Notes

Remember vectors have both size and capacity:

Size and Capacity

The **size** of a vector is the number of elements in the vector. The **capacity** of a vector is the number of elements for which it currently has memory allocated. For a vector v, the size and capacity can be recovered with the member functions `v.size( )` and `v.capacity( )`.

As long as the size remains less than the capacity, than the efficiency of the vector data structure is similar to an array.

However, once you increase the size beyond that of the capacity, there is significant overhead.

Other Vector Notes

From a programming perspective, you can basically ignore the capacity issue as the increase in capacity will happen automatically if need be.

- It will only impact your efficiency.

You can use the reserve member function to explicitly increase a vector's size

`v.reserve(32);` sets the capacity to at least 32 elements

- You can only use reserve to increase the capacity of a vector; it does not necessarily decrease the capacity of a vector if the argument is smaller than the current capacity

Other Vector Notes

You can resize a vector as well

`v.resize(24);` resizes a vector to 24 elements:

- If the original vector size was less than 24 elements, then the new elements are initialized using the default constructor
  - Similar to the vector initialization with the integer argument.
- If the original vector size is greater than 24, all but the first 24 elements are lost.

Now that we've reminded of ourselves of using a template type (vectors), let's get back to how we use template functions.

Recall our swap_values function.

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

Here's sample code for the template function swap_values

The function lets you swap the values of any two variables of any type

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

The definition and function declaration begin with:

```
template<class T>
```

This is called the ***template prefix*** and tells the compiler that the definition that follows is a template and that `T` is a ***type parameter***.

In this context the word class actually means type.

- You can use the keyword `typename` instead of `class` in the template prefix, but most people use class.

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

The template function can be used with any type (predefined, user-defined, class/not class)

Within the body of the function definition, the type `T` is used like any other type.

The function template definition acts as a definition for a collection of functions, where each one is generated by replacing the type `T` with a different type.

This example shows integer and char examples

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

The compiler does not literally produce definitions for every possible type for the function name `swap_values`.

Instead it will produce definitions for each of the different types that use the template.

- In this example, it uses the template to produce function definitions for the types int and char

Note you don't need to do anything special when you call a function that is defined with a function template.

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

Note that the function template definition is located before the main part of the program.

- Also, there is no template function declaration.

This is because of varied compiler support.

Some compilers do not support template function declarations and do not support separate compilation of template functions.

- Therefore your safest strategy is to not use template function declarations.

```
//Program to demonstrate a function template.
#include <iostream>
using namespace std;

//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}

int main( )
{
    int integer1 = 1, integer2 = 2;
    cout << "Original integer values are "
         << integer1 << " " << integer2 <<endl;
    swap_values(integer1, integer2);
    cout << "Swapped integer values are "
         << integer1 << " " << integer2 <<endl;

    char symbol1 = 'A', symbol2 = 'B';
    cout << "Original character values are "
         << symbol1 << " " << symbol2 <<endl;
    swap_values(symbol1, symbol2);
    cout << "Swapped character values are "
         << symbol1 << " " << symbol2 <<endl;

    return 0;
}
```

As such, you need to ensure the function template definition appears in the same file before it is used

- Alternatively, the function template definition can appear via a `#include` directive.
  - This means you can give the function template definition in one file and then `#include` that file in the file that uses the template function.

For more details check out your compiler's details

```
//Interchanges the values of variable1 and variable2.
template<class T>
void swap_values(T& variable1, T& variable2)
{
      T temp;

      temp = variable1;
      variable1 = variable2;
      variable2 = temp;
}
```

In the example template, we use the letter `T` as the parameter for the type.

This traditional, but not required (like using `e` for the exception parameter).

```
template<class VariableType>
void swap_values(VariableType& variable1,
                 VariableType& variable2)
{
    VariableType temp;

    temp = variable1;
    variable1 = variable2;
    variable2 = temp;
}
```

Here's an example of an alternative that is equivalent to the above definition.

It is possible for function templates to have more than one type of parameter as shown here:

```
template<class T1, class T2>
```

Generally, though a function template only uses one type of parameter.

NOTE: You cannot have an unused template parameter- each parameter must be used in your template function somewhere.

Also, since you don't want to use template function declarations (so that your code is portable and usable with different compilers), generally your function template definition will "appear" in the main file where you use it via an #include directive.

- Note you can include both header (.h) and implementation (.cpp) files.

Algorithm Abstraction

As we saw in our discussion of the `swap_values` function, there is a very general algorithm for interchanging the value of two variables, and this more general algorithm applies to variables of any type. Using a function template, we were able to express this more general algorithm in C++. This is a very simple example of *algorithm abstraction*. When we say we are using **algorithm abstraction,** we mean that we are expressing our algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part of the algorithm. Function templates are one feature of C++ that supports algorithm abstraction.

In CMPT 128 (from Chapter 7), you would have looked at a simple sorting algorithm to sort an array of values of type int. Here's the original code:

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used - 1; index++)
    {//Place the correct value in a[index]:
        index_of_next_smallest =
                index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of
        //the original array elements. The rest of the
        //elements are in the remaining positions.
    }
}
```

However, the base type of the array (int) isn't really used in the algorithm and this could be applied more generically to other types (e.g. doubles, floats, chars, …)

To make a generic function, what portions of the code would we need to fix:

```
void sort(int a[], int number_used)
{
    int index_of_next_smallest;
    for (int index = 0; index < number_used - 1; index++)
    {//Place the correct value in a[index]:
        index_of_next_smallest =
                    index_of_smallest(a, index, number_used);
        swap_values(a[index], a[index_of_next_smallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of
        //the original array elements. The rest of the
        //elements are in the remaining positions.
    }
}
```

The helper functions (index_of_smallest() and swap_values). What else???

We already looked at swap_values(); let's look at index_of_smallest()

Can we make this function generic and independent of its base type?

```
int index_of_smallest(const int a[], int start_index,
                      int number_used)
{
    int min = a[start_index];
    int index_of_min = start_index;
    for (int index = start_index + 1;
         index < number_used; index++)
    {
        if (a[index] < min)
        {
            min = a[index];
            index_of_min = index;
            //min is the smallest of a[start_index] through
            //a[index]
        }
    }
    return index_of_min;
}
```

If yes, what do we need to change to make this work for doubles?

Since these functions work basically “as is” by simply swapping types from int to double, can we extend them to user defined types?

Yes, but what do we need to worry about:

-definition (overloading) the less than operator (‘<‘)

**Also possibly the definition of the assignment operator and a copy constructor.

The next slides look at the definition of the generic sorting function and an example of its use.

sortfunc.cpp:

```
template<class T>
void swap_values(T& variable1, T& variable2)
     <The rest of the definition of swap_values is given in Display 17.1.>

template<class BaseType>
int index_of_smallest(const BaseType a[], int start_index, int number_used)
{
    BaseType min = a[start_index];
    int index_of_min = start_index;

    for (int index = start_index + 1; index < number_used; index++)
        if (a[index] < min)
        {
            min = a[index];
            Index_of_min = index;
            //min is the smallest of a[start_index] through a[index]
        }

    return index_of_min;
}

template<class BaseType>
void sort(BaseType a[], int number_used)
{
int index_of_next_smallest;
for (int index = 0; index < number_used - 1; index++)
   {//Place the correct value in a[index]:
          index_of_next_smallest =
                index_of_smallest(a, index, number_used);
          swap_values(a[index], a[index_of_next_smallest]);
       //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
       //elements. The rest of the elements are in the remaining positions.
   }
}
```

Remember some compilers don't support template function declarations.

Since you have included the "sortfunc.cpp" file, the declaration is not needed.

- The comment just makes your code more readable

```
//Demonstrates a generic sorting function.
#include <iostream>
using namespace std;

//The file sortfunc.cpp defines the following function:
//template<class BaseType>
//void sort(BaseType a[], int number_used);
//Precondition: number_used <= declared size of the array a.
//The array elements a[0] through a[number_used - 1] have values.
//Postcondition: The values of a[0] through a[number_used - 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].

#include "sortfunc.cpp"
```

Many compilers will allow this function declaration to appear as a function declaration and not merely as a comment. However, including the function declaration is not needed, since the definition of the function is in the file `sortfunc.cpp`*, and so the definition effectively appears before* `main`*.*

main

(Part A):

```
int main( )
{
    int i;
    int a[10] = {9, 8, 7, 6, 5, 1, 2, 3, 0, 4};
    cout << "Unsorted integers:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;
    sort(a, 10);
    cout << "In sorted order the integers are:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << " ";
    cout << endl;

    double b[5] = {5.5, 4.4, 1.1, 3.3, 2.2};
    cout << "Unsorted doubles:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;
    sort(b, 5);
    cout << "In sorted order the doubles are:\n";
    for (i = 0; i < 5; i++)
        cout << b[i] << " ";
    cout << endl;

    char c[7] = {'G', 'E', 'N', 'E', 'R', 'I', 'C'};
    cout << "Unsorted characters:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;
```

main

(Part A):

```
    sort(c, 7);
    cout << "In sorted order the characters are:\n";
    for (i = 0; i < 7; i++)
        cout << c[i] << " ";
    cout << endl;
    return 0;
}
```

Example

Output:

```
Unsorted integers:
9 8 7 6 5 1 2 3 0 4
In sorted order the integers are:
0 1 2 3 4 5 6 7 8 9
Unsorted doubles:
5.5 4.4 1.1 3.3 2.2
In sorted order the doubles are:
1.1 2.2 3.3 4.4 5.5
Unsorted characters:
G E N E R I C
In sorted order the characters are:
C E E G I N R
```

So how do you design a function template?

1. Start by designing the function for a base type (e.g. int) and then debugging it so that you are sure it works for the base case.
2. Next step, replace the base type with the type parameters and try it for a couple of different types.
   - If it still works then you probably have the generic version properly defined.

Basically, it is easier to think of the concrete case first to develop the algorithm and then worry about the template syntax rules later.

Warning: you can use a template function with any type for which the code in the function definition makes sense.

In other words all the code in the template must behave in the appropriate way when you use it with a specific type.

For example, you cannot use the swap_values template function with a type parameter for which the assignment operator does not work at all/correctly,

In other words, the following will not work:

```
int a[10], b[10];
<some code to fill arrays>
swap_values(a, b);
```

Why? Because the assignment operator does not work on array types.

```
//Class for a pair of values of type T:
template<class T>
class Pair
{
public:
    Pair();

    Pair(T first_value, T second_value);

    void set_element(int position, T value);
    //Precondition: position is 1 or 2.
    //Postcondition:
    //The position indicated has been set to value.

    T get_element(int position) const;
    //Precondition: position is 1 or 2.
    //Returns the value in the position indicated.
private:
    T first;
    T second;
};
```

You can also use templates to create class definitions that are more general.

Note the similarities with function templates (both start with: `template<class T>`)

For example, if you instantiate objects values where `T` is set to int, then you have pairs of integers.

You declare objects of template classes using:

```
Pair<int> score;
Pair<char> seats;
```

```
//Class for a pair of values of type T:
template<class T>
class Pair
{
public:
    Pair();

    Pair(T first_value, T second_value);

    void set_element(int position, T value);
    //Precondition: position is 1 or 2.
    //Postcondition:
    //The position indicated has been set to value.

    T get_element(int position) const;
    //Precondition: position is 1 or 2.
    //Returns the value in the position indicated.
private:
    T first;
    T second;
};
```

You use objects based on template classes as you would any other object.

For example, given:

```
Pair<int> score;
Pair<char> seats;
```

You can:

```
score.set_element(1, 3);
score.set_element(2, 0);
```

What about member function definitions?

```
//Uses iostream and cstdlib:
template<class T>
void Pair<T>::set_element(int position, T value)
{
    if (position == 1)
        first = value;
    else if (position == 2)
        second = value;
    else
    {
        cout << "Error: Illegal pair position.\n";
        exit(1);
    }
}

template<class T>
Pair<T>::Pair(T first_value, T second_value)
      : first(first_value), second(second_value)
{
     //Body intentionally empty.
}
```

Member functions for a class template are defined in the same way as member functions for ordinary classes.

- The only difference is that the member functions are themselves templates.

Note that the class name before the scope resolution operator is `Pair<T>`, not simply `Pair`.

You can also use a class template as the type for a function parameter as shown here:

```
int add_up(const Pair<int>& the_pair);
//Returns the sum of the two integers in the_pair.
```

However, the function declaration and definition should use this template type.

You can even use a class template within a function template.

- For example, instead of using the specialized version of `add_up` above, you can create the generic version shown here and apply it to all kinds of numbers:

```
template<class T>
T add_up(const Pair<T>& the_pair);
//Precondition: The operator + is defined for values of type T.
//Returns the sum of the two values in the_pair.
```

Type Definitions

You can specialize a class template by giving a type argument to the class name, as in the following example:

```
Pair<int>
```

The specialized class name, like `Pair<int>`, can then be used just like any class name. It can be used to declare objects or to specify the type of a formal parameter.

You can define a new class type name that has the same meaning as a specialized class template name, such as `Pair<int>`. The syntax for such a defined class type name is as follows:

```
typedef Class_Name<Type_Argument> New_Type_Name;
```

For example:

```
typedef Pair<int> PairOfInt;
```

The type name `PairOfInt` can then be used to declare objects of type `Pair<int>`, as in the following example:

```
PairOfInt pair1, pair2;
```

The type name `PairOfInt` can also be used to specify the type of a formal parameter.

The following slides contain the interface for a class template whose objects are lists and then the implementation of the class template.

Notes:

- Since the definition is a class template, the lists can be lists of items of any type.
  - You can create objects that are lists of ints, doubles, chars, user-defined types etc.
- We have overloaded the insertion operator (“<<“) so that it can be used to output an object of the class template GenericList.
  - In order to have a parameter that is of the same type as the class, the parameter type is GenericList<ItemType> (i.e. the class type).
  - It is a friend function and has been defined in the header file (common when using friend functions or operators within a template). This makes it easy for the compiler to find the definition and usable by any objects declared outside of the class

Header file: Interface

```
//This is the header file genericlist.h. This is the interface for the
//class GenericList. Objects of type GenericList can be a list of items
//of any type for which the operators << and = are defined.
//All the items on any one list must be of the same type. A list that
//can hold up to max items all of type Type_Name is declared as follows:
//GenericList<Type_Name> the_object(max);
#ifndef GENERICLIST_H
#define GENERICLIST_H
#include <iostream>
using namespace std;

namespace listsavitch
{
    template<class ItemType>
    class GenericList
    {
    public:
        GenericList(int max);
        //Initializes the object to an empty list that can hold up to
        //max items of type ItemType.
        ~GenericList( );
        //Returns all the dynamic memory used by the object to the freestore.

        int length( ) const;
        //Returns the number of items on the list.

        void add(ItemType new_item);
        //Precondition: The list is not full.
        //Postcondition: The new_item has been added to the list.
```

Header file: Interface Continued (NOTICE ALL THE COMMENTS)

```
        bool full( ) const;
        //Returns true if the list is full.

        void erase( );
        //Removes all items from the list so that the list is empty.

        friend ostream& operator <<(ostream& outs,
                                    const GenericList<ItemType>& the_list)
        {
                for (int i = 0; i < the_list.current_length; i++)
                        outs << the_list.item[i] << endl;
                return outs;
        }
        //Overloads the << operator so it can be used to output the
        //contents of the list. The items are output one per line.
        //Precondition: If outs is a file output stream, then outs has
        //already been connected to a file.
        //
        //Note the implementation of the overloaded << in the header
        //file! This is commonly done with overloaded friend templates.
        //Since << is a friend it is NOT a member of the class but
        //rather in the namespace, this is the simplest implementation
        //and may make more sense than putting it in genericlist.cpp.
    private:
        ItemType *item; //pointer to the dynamic array that holds the list.
        int max_length; //max number of items allowed on the list.
        int current_length; //number of items currently on the list.
    };
}//listsavitch
#endif //GENERICLIST_H
```

Implementation file: GenericList

```
//This is the implementation file: genericlist.cpp
//This is the implementation of the class template named GenericList.
//The interface for the class template GenericList is in the
//header file genericlist.h.
#ifndef GENERICLIST_CPP
#define GENERICLIST_CPP
#include <iostream>
#include <cstdlib>
#include "genericlist.h" //This is not needed when used as we are using this file,
                         //but the #ifndef in genericlist.h makes it safe.
using namespace std;

namespace listsavitch
{
    //Uses cstdlib:
    template<class ItemType>
    GenericList<ItemType>::GenericList(int max) : max_length(max),
                                                  current_length(0)
    {
        item = new ItemType[max];
    }

    template<class ItemType>
    GenericList<ItemType>::~GenericList( )
    {
        delete [] item;
    }
```

Implementation file: GenericList Continued

```
template<class ItemType>
int GenericList<ItemType>::length( ) const
{
    return (current_length);
}

//Uses iostream and cstdlib:
template<class ItemType>
void GenericList<ItemType>::add(ItemType new_item)
{
    if ( full( ) )
    {
        cout << "Error: adding to a full list.\n";
        exit(1);
    }
    else
    {
        item[current_length] = new_item;
        current_length = current_length + 1;
    }
}

template<class ItemType>
bool GenericList<ItemType>::full( ) const
{
    return (current_length == max_length);
}
```

Implementation file: GenericList Continued

```
    template<class ItemType>
    void GenericList<ItemType>::erase( )
    {
        current_length = 0;
    }
}//listsavitch
#endif // GENERICLIST_CPP Notice that we have enclosed all the template
       // definitions in #ifndef. . . #endif.
```

The following is a demonstration program that uses the GenericList class template in an example application program.

It is a simple example used only to illustrate the syntax details.

Since we included genericlist.cpp, only the main program needs to be compiled.

```
//Program to demonstrate use of the class template GenericList.
#include <iostream>
#include "genericlist.h"
#include "genericlist.cpp"
using namespace std;
using namespace listsavitch;

int main( )
{
    GenericList<int> first_list(2);
    first_list.add(1);
    first_list.add(2);
    cout << "first_list = \n"
         << first_list;
    GenericList<char> second_list(10);
    second_list.add('A');
    second_list.add('B');
    second_list.add('C');
    cout << "second_list = \n"
         << second_list;

    return 0;
}
```

Since `genericlist.cpp` *is included, you need compile only this one file (the one with the* `main`*).*

The following is the program's output:

```
first_list =
1
2
second_list =
A
B
C
```

Review Questions for Slide Set 10

- What do templates allow you to do?
- How should you go about designing a template function?
- Recall the vector class: what is the difference between “size” and capacity?
- How does a vector differ from an array?
- How do you initialize a vector?
- When can you use square brackets to manipulate a vector
- Can you create vectors of User defined class types?
- What is the starting index of a vector?
- Can you initialize a vector using v[i] = 42;
- When are base type constructors called in vector initializations?

Review Questions for Slide Set 10

- If you try to access v[i], for values of i >= v.size, will you get an error message? Will your program work properly?
- How does the assignment operator work for a vector?
- Could using a vector for a user defined class mean you will need copy constructors? If yes, why?
- What happens if you want to set the size larger than the capacity of the vector?
- How efficient are vectors at runtime? What are the costs?
- What is the purpose/impact of using the reserve function?
- What is the purpose/potential impact of using the resize function?
- What is the template prefix?

Review Questions for Slide Set 10

- Can template functions be used with any type?
- What is the purpose of the function template definition?
- How does a compiler treat a template function definition?
- Why not use a template function declaration?
- Can you include .cpp files? Why is this useful?
- If a function template definition needs to appear in the same file before it is used, can it still be defined in a separate file? If yes, how do you work around this?
- Can function templates have more than one type of parameter? If yes, are there any restrictions?
- What is algorithm abstraction? Why is it useful?

Review Questions for Slide Set 10

- When making generic template functions from pre-existing could, what do you need to be aware of (what might you need to fix)?
- When can you not use a function template with a specific type parameter? Give an example using the swap_values function template we talked about.
- Can you use templates with class definitions? What is required?
- If you have a class template, what happens to the member functions?
- If you want to use a class template as a type for a function parameter, what do you need to do?

Review Questions for Slide Set 10

- Can you use a class template with a function template?
- Can class templates and function templates use dynamic memory? If a class template has dynamic memory, what do you need to worry about?
- If you include a .cpp file in your main executable, does it need to be compiled separately? Why/why not?