Week3 :

Scientific Computing
Strings, Vectors, Errors and Arrays

Blair Jamieson

University of Winnipeg

Class 3

Outline

Strings, Vectors, Errors and Arrays

Outline

Strings, Vectors, Errors and Arrays
- `using` declarations
- The standard library string
- Standard library `vector`
- Iterators
- Errors
- Arrays
- Multidimensional arrays

Namespace `using` declarations

- A `namespace` is used to put functions and variables outside of the `global` scope to avoid name clashes.
- Example: declare function `factorial` in its header file `myfuncs_factorial.h` and want it in a namespace that we call `myfcns`:

```
1 namespace myfuncs {
2   double factorial( int an );
3 }
```

- Function definition in `myfuncs_factorial.cpp`:

```
1 #include "myfuncs_factorial.h"
2 double myfuncs::factorial( int an ){
3   double retval=0.;
4   // code to calculate factorial here ...
5   return retval;
6 }
```

- To use the function, include the header then either:
  - Put `using namespace myfuncs;` on a line after include,
  - put `using myfuncs::factorial;` on a line after include, or
  - explicitly specify namespace on every call to `myfuncs::factorial( someValue )`

The `std::` namespace

- As we have seen the namespace `std` holds the `classes` from `iostream` for input `cin` and output `cout`
- `std` namespace also holds `classes` to represent `string`, and `vector`
- One way to use these classes, is to explicitly specify the `std` namespace on every use of the class
- If we want to have the facility to use all of the classes in the `std` namespace without specifying `std::` on every use, add the line of code:
  `using namespace std;`
- If we only want to use the vector `class` we should use the line:
  `using std::vector;`

Defining and initializing `std::string`

- For the next few slides, assume that we have included the `string` header, and put a `using` statement:

```
#include <string>
using std::string;
```

- Example initialization of `string` objects:

```
string s1;                  // empty string
// Below: s2 init by copy of string literal,
// including trailing null
string s2 = "Hi";
string s3 = s2;             // s3 init by copy of s2
string s4 ( 10, 'c' ); // s4 is ccccccccccc
// Below: s4 init by literal, no trailing null
string s5 ( "Bye" );
```

- Note that in C, end of character array is given by null ('\0').

Operations on `string` objects

- Reading and writing strings:

`stringIO.cpp`

```
1 // Note #include and using decl. must be above
2 int main(){
3   string s;
4   cout << "Enter a string:" << endl;
5   cin >> s;
6   cout << s << endl;
7   return 0;
8 }
```

- Note that by default » operator for a `string` will read input until a white space is found.
- If we enter “Hello World”, then only “Hello” will be put in our `string s`.

Reading until end-of-file condition

- Similar to our examples reading in multiple `int` objects, we can test stream state when reading in `string` objects.

```
int main(){
  string inword;
  cout << "Enter a word: " << endl;
  while ( cin >> inword ) {
    cout << inword << endl;
    cout << "Enter a word: " << endl;
  }
  return 0;
}
```

- Input continues until end-of-file (Ctrl-d, or Ctrl-z) is entered

Table of `string` operations

`os << s`	Writes `s` to output stream `os`, return `os`
`is >> s`	Read input stream `is` until space, return `is`
`getline(is,s)`	Read line from `is` into `s`, return `is`
`s.empty()`	Return `true` if `s` is empty, else return `false`
`s.size()`	Return number of characters in `s`
`s[n]`	Return reference to `char` at position `n` in `s`
`s1 + s2`	Return `string` that is concatenation of `s1` and `s2`
`s1 = s2`	Replace characters in `s1` with a copy of `s2`
`s1 == s2`	Return `true` if strings contain same characters
`s1 != s2`	Return `true` if strings differ in any way
`<,<=,>,>=`	Comparisons are case-sensitive and use dictionary ordering

Reading entire line: `getline`

- In cases where we want the whole line of input including spaces use `getline` function
  - Gets string input until a newline is entered
  - Arguments: input stream to read from, string to put line into
  - Returns `istream`, so it can be used to test condition of `stream`

`readline.cpp`

```
1  using std::getline;
2  using std::cout;
3  using std::cin;
4  int main(){
5    string line;
6    cout << "Enter a sentence: " << endl;
7    while ( getline( cin, line ) ){
8      cout << line << endl;
9      cout << "Enter a sentence: " << endl;
10   }
11   return 0;
12 }
```

String empty and size operations

- `empty` method of `string` class returns `true` if the string is empty, and `false` if the string is not-empty

```
string word;
cout << "What is your favorite colour?" << endl;
if ( cin >> word ){
  if ( !cin.empty() ) {
    cout << "Your favorite colour is " << word <<
        endl;
  } else {
    cout << "Empty string entered" << endl;
}
```

- `size` method of `string` returns the number of characters in the string as a `string::size_type`
  - This is an unsigned int like type
  - `string::size_type` is a bit long to type – in C++ 11 we can let the compiler provide an appropriate type using `auto` declaration:

```
string myname("Dr. Blair Jamieson");
auto len = myname.size();
```

Comparing strings

Rules for string comparisons (<,<=,>,>=):

1. Strings are == if they contain the same characters and same length (case sensitive).
2. If two strings have different lengths and if every character in the shorter string is equal to the corresponding character in the longer string, then the shorter string is < longer string.
3. If any characters at corresponding positions in two strings differ, than the result of the string comparison is the result of comparing the first character at which the strings differ

String concatenation

- Adding two strings with the + or += operator concatenates two strings

```
string s1 = "Hello ", s2 = "World!\n";
string s3 = s1 + s2;
// s3 now holds "Hello World!\n"
s1+=s2; // s1 now holds "Hello World!\n"
```

- Can also concatenate string and char literals, but must start with a string type:

```
string s4=s1+" Hi."; // ok: add string and literal
string s5="Hi " + s1; // error: no string operand
string s6=s1 + "  " + "Hi."; // ok: starts w/string
string s7="Hello"+"World"; // error: no string in +
```

Changing characters in a string

- Functions to check character type are declared in `cctype` header file
- Note in C, these same functions are declared in `ctype.h`. Typically C++ versions of C header *name.h* are in *cname*.

`isalnum(c)`	*true* if *c* is letter or digit
`isalpha(c)`	*true* if *c* is letter
`iscntrl(c)`	*true* if *c* is control char
`isdigit(c)`	*true* if *c* is a digit
`isgraph(c)`	*true* if *c* is not a space but is printable
`islower(c)`	*true* if *c* is lowercase letter
`isprint(c)`	*true* if *c* is printable
`ispunct(c)`	*true* if *c* is punctuation
`isspace(c)`	*true* if *c* is a whitepace
`isupper(c)`	*true* if *c* is uppercase letter
`isxdigit(c)`	*true* if *c* is hexidecimal digit
`tolower(c)`	Changes letter `c` to lowercase
`toupper(c)`	Changes letter `c` to uppercase

Accessing characters in string

Consider `string s="Something";`:

- Use subscript operator [] taking a `string::size_type` that gives position of character we want to access
- Subscripts start at 0:
  `s[0]` returns the first character in the string.
- The last character in the string:
  `s[ s.size()-1 ]`
- Be careful not to try accessing a character in an empty string, or from a subscript beyond the last character.

Example: change letters to be uppercase

```
1 string s="Something sounds loud!";
2 for (string::size_type i=0; i<s.size(); i++){
3   s[i] = toupper( s[i] );
4 }
5 cout << s << endl;
```

Range for loops

- Range `for` loop introduced in C++ 11 can be used to iterate through elements. Form of **range for** is:

```
for ( declaration : expression )
      statement;
```

- `declaration` defines the variable used to access elements
- `expression` is object of a type representing a sequence; `string` represents a sequence of characters
- Example: use reference to element in sequence to change characters in string to lowercase:

```
string s2 = "STOP talking SO Loud!";
for ( auto & c : s2 ) {
  c = tolower( c );
}
```

More loops and string manipulations

- Can also do range for loop with value, rather than reference
- Example: Count the punctuation marks:

```
string s3 = "So! much... punctuation?";
unsigned count = 0;
for ( auto c : s3 ){
  if ( ispunct( c ) ) count++;
}
```

- When using regular `for` loop can use `decltype` to automatically get type of loop variable.
- Example: count spaces in `s3`

```
unsigned spcount = 0;
for ( decltype(s3.size()) i=0; i<s3.size(); i++){
     if ( isspace( s3[i] ) spcount++;
}
```

Example: Convert number to hexadecimal

Enter numbers between 0 and 15 and convert them into a string of hexadecimal digits until an invalid number is entered.

PrintAsHex.cpp

```
#include <iostream>
#include <string>
using std::string;
using std::cin;
using std::cout;
int main(){
  const string hexdigs = "0123456789ABCDEF";
  string hexoutput;
  string::size_type dig;
  while ( cin >> dig ){
    if ( dig < hexdigs.size() ){
      hexoutput += hexdigs[ dig ];
    }
  }
  cout << "You entered 0x" << hexoutput << endl;
  return 0;
}
```

std::vector

- A `vector` is a sequence of elements you access by an index
- Example:

```
1    vector<int> v(6); //vector with 6 elements
2    v[0] = 5; // first element has index zero
3    v[1] = 7;
4    v[2] = 9;
5    v[3] = 4;
6    v[4] = 6;
7    v[5] = 8; // last element is at size()-1
```

- The type of object in each element is specified between < and > after the `vector` type

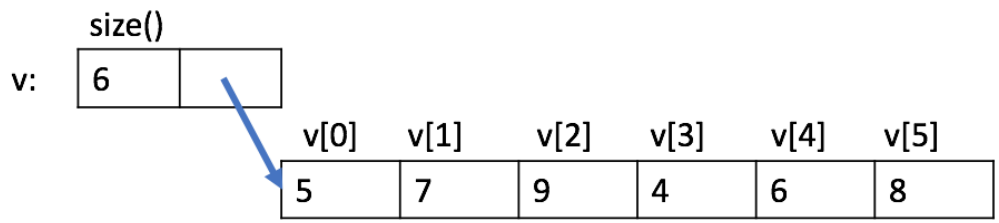

Figure: `vector` v's elements.

std::vector

- A `vector` is a collection of objects that all have the same type.
- A `vector` is a container because it contains other objects
- To use `vector` we need to include the vector header, and have a using declaration:

```
1 #include <vector>
2 using std::vector;
```

- `std::vector` is a **class template**, which we can use quite easily, even though implementing your own class templates requires a deep understanding of C++
- templates are instructions to the compiler for generating classes or functions
- The process of creating classes or functions from templates is called **instantiation**
- When using a `vector` **class template** we specify inside angle brackets <> the type of objects the `vector` will hold.

Defining and instantiating vectors

- Some examples of vector definitions:

```
vector<int> ivec; // ivec: vector of integers
vector<double> dvec; // dvec: vector of doubles
vector<string> svec; // svec: vector of strings
// below: vvs holds a vector of vectors of strings
vector<vector<string>> vvs;
vector<Box_st> vbox; // vector of Box_st
```

- Some examples of vector initialization:

```
vector<string> svec; // init as empty vector
// since C++11 can use list initialization
vector<string> scols = {"red", "green", "blue"};
vector<string> sv2( scols ); // copy scols into sv2
vector<string> sv3 = scols;  // copy scols into sv3
//below error: sv4 cant hold int
vector<string> sv4( ivec );
```

More vector initialization

- Can initialize with list using {} brackets
- Can initialize with value using () brackets

```
1 vector<int> vec1(10);    // vec1 holds 10 zeros
2 vector<int> vec2(10,5);  // vec2 holds 10 fives
3 vector<int> vec3{10};    // vec3 holds 1 ten
4 vector<int> vec4{10,5};  // vec4 holds 10 and 5
5 vector<int> vec5={10,5};// vec5 holds 10 and 5
6 vector<int> vec6(vec5);  // vec6 holds 10 and 5
```

Adding elements to a vector

- Say we want to build a vector holding numbers from 0 to 99.
- Create empty vector, then fill it using `vector` member function called **push_back**.
- `push_back` method adds an element as a new last element (at the back) of the vector. For example:

```
vector <int> v1;
for (int i=0; i<100; i++)
  v1.push_back( i );
```

- As another example, we might want to read input strings and store the values using a vector:

```
vector<string> words;
string inword;
while ( cin >> inword ){
  words.push_back( inword );
}
```

`vector` size

- In C and Java memory use is more efficient to define `vector` at its expected size.
- In C++ it is **better to add elements at run time**, rather than specify a size in the declaration.
- This is because the C++ standard has been made to allow efficient addition of elements to `vector`
- Exception to this rule is if `vector` needs to have all the same values.
- Programming implications for adding elements to `vector`:
  - We cannot use a `range for` loop if the body of the loop adds elements to the `vector`

Operations on `vector`

Consider the following operations on `vector v, v1, v2;`:

`v.empty()`	return `true` if the vector is empty
`v.size()`	return the number of elements in vector
`v.push_back( val )`	adds element of value `val` to end of vector
`v[n]`	returns the value in vector at index `n`
`v1 = v2`	replace elements in `v1` with copy of elements in `v2`
`v1={a,b,c,...}`	replace elements in v1 with elements in list
`v1 == v2`	returns `true` if vectors have same elements
`v1 != v2`	returns `true` if vectors have any difference
<,<=,>,>=	Have normal meanings using dictionary ordering

- As with accessing characters in `string` we can access element in a vector using [] operator.
- Subscript of first element in vector is `v[0]`, and last is `v[vec.size()-1]`.
- Be careful not to subscript an element in an empty vector or beyond the size of the vector.

Example vector use

- Lets write a short program that:
  - Asks the user to input a list of grades
  - Counts the number of grades in each 10% increment (0-9, 10-19, ... , 90-99, 100).

CountGrades.cpp

```
1  // Above add required #include and using statements
2  int main(){
3    unsigned grade;
4    vector<int> counts(11);
5    while ( cin >> grade ){
6      if ( grade <= 100 ){
7        counts[ grade/10 ] ++;
8      }
9    }
10   // print table etc...
11   return 0;
12 }
```

Subscripting a vector does not add elements

- Use `push_back` to add elements to a vector; size of the vector does not change by using subscripting []
- Cautionary examples:

```
vector<int> iv; // empty vector
for ( decltype( iv.size() ) i=0 ; i < 10 ; i++ ){
  iv[ i ] = i;// error: iv has no elements!
}
cout << iv[0]; // error: iv has no elements!
vector<int> jv(10); //vector of 10 elements
cout << jv[10]; // error; jv has elements 0 to 9
```

Growing a `vector`

Use the `push_back()` member function to add elemnts to a vector:

```
1       vector<double> vd;
2       vd.push_back(2.7);
3       vd.push_back(5.6);
4       vd.push_back(7.9);
```

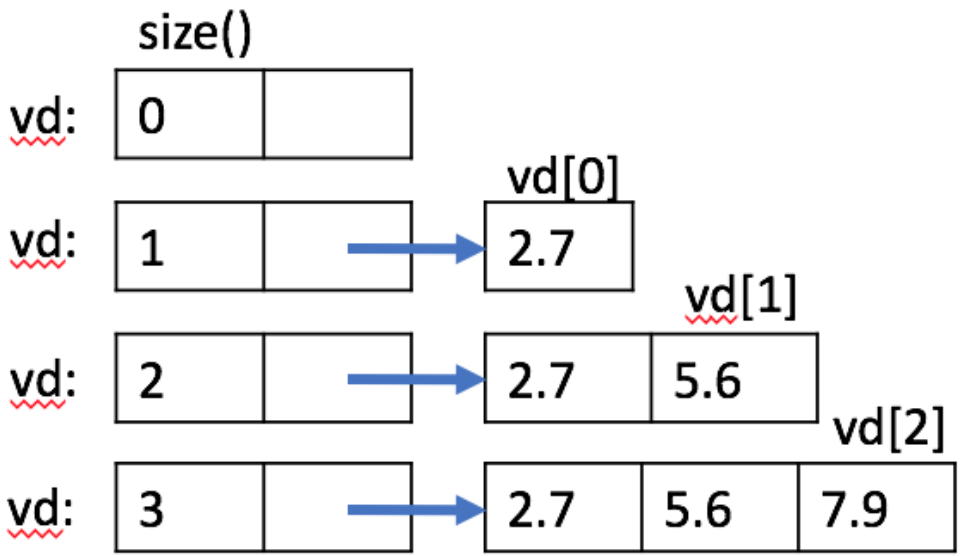

Figure: `vector` vd's elements after each line of code.

A numeric example (4.6.3)

Read in temperatures to a vector and compute mean, and median

`temperatures.cpp`

```
1  int main(){
2    vector<double> temps; // temperatures
3    double temp;
4    cout<<"Enter temperatures, non-numeric to end
         inputs"<<endl;
5    while ( cin>>temp ){
6      temps.push_back( temp );
7    }
8    double sum=0.;    // compute mean temperature
9    for ( int i=0; i < temps.size(); ++i ){
10     sum += temps[i];
11   }
12   cout<<"Average temperature:
         "<<sum/temps.size()<<endl;
13   // compute median
14   sort( temps.begin(), temps.end() );
15   cout<<"Median temperature: "<<temps[ temps.size()/2
         ]<<endl;
16   return 0;
```

Notes on numeric example (4.6.3)

1. `cin»` temp reads a double → if it succeeds in getting a double we push it to the end of the vector
   - eg. if you type in: 1.2 3.4 5.6 7.8 9.0 |
   - then temps gets the five element above
   - the | isn't a double, so the while loop exits
2. We calculate the average by summing over the elements, and dividing by the size of the vector
3. To calculate the median (a value chosen so that half the values are smaller, and half are larger), we sort the elements
4. We used the standard library `sort` from the algorithm library – it uses the beginning and end of a sequence as arguments
5. We used `vector`'s `begin()` and `end()` methods to get the beginning and end of the sequence

A text example (4.6.4)

vector is particularly useful because it can be used for many problems

wordlist.cpp

```
1  int main(){
2    vector<string> words; // white space separated words
3    string s;
4    cout<<"Enter words separated by space (^Z to
         end):"<<endl;
5    while ( cin >> s ){
6      words.push_back( s );
7    }
8    cout<<"Number of words: "<<words.size()<<endl;
9    sort( words.begin(), words.end() );
10   for (int i=0; i<words.size(); ++i ){
11     if (i==0 || words[i-1] != words[i] ){
12       cout<<words[i]<<endl;
13     }
14   }
15   return 0;
16 }
```

Iterator intro

- In addition to using subscripts, **iterators** are another way to access elements of `string` and `vector`. **Iterators** exist for all of the standard library container types.
- Iterators are similar to pointers:
  - iterators give indirect access to an object that is an element of a container
  - iterators can be valid if it denotes an element of the vector or a position one past the end of the container, or invalid if it is any other value
- Instead of using an address to get an iterator, we use special member functions `begin()` and `end()` to get iterators referring to the first and one past the last elements in the container
- If the container is empty both the `begin` and `end` methods will return off-the-end iterators.

Using iterators

- Example of iterator use to change characters in string to uppercase:

TestIterator.cpp

```
1 string ss("stop shouting so loud!");
2 for ( auto iter = ss.begin(); iter != ss.end();
      iter++ ){
3   *iter = toupper( *iter );
4 }
```

- Note that the * operator is used to dereference the iterator, giving you the value of the object at that iterator location

Iterators and generic programming

- A key feature of C++ is the use of iterators to visit the elements of containers
- The `vector` container in addition to defining == and != operators, defines (<,<=,>,and >=) operators
- In general iterator types will define == and != operators, but depending on the contents of the container may not define other comparison operators (<,<=, >, or >=)
- By routinely using iterators == and != we don't have to worry about precise type of container we are looking at
- Iterator types:

```
1 vector<int> v1;
2 const vector<int> v2;
3 // itv1 with type vector<int>::iterator
4 auto itv1 = v1.begin();
5 // itv2 with type vector<int>::const_iterator
6 auto itv2 = v2.begin();
7 // itcv1 has type vector<int>::const_iterator
8 auto itcv1 = v1.cbegin();
```

- `const_iterator` type is used when we don't want to modify the contents of the container being iterated over

Iterator operations

`*iter`	return ref to element denoted by `iter`
`iter->mem`	fetches member named `mem` from element denoted by `iter`
`++iter`	increment `iter` to refer to next element
`--iter`	decrement `iter` to refer to prev element
`iter1 == iter2`	return `true` if `iter1` and `iter2` are equal
`iter1 != iter2`	return `true` if `iter1` and `iter2` differ
`iter + n`	adding (subtracting) an integer from iterator yeids an
`iter - n`	iterator that many elements ahead (back) in container
`iter += n`	assigns `iter` iterator `n` ahead in container
`iter -= n`	assigns `iter` iterator `n` back in container
`iter1 - iter2`	yields number when added to `iter2` gives `iter1`
`>,>=,<,<=`	Iterator is less than another if it refers to element coming before the one being compared to

Using iterator arithmetic

Example use of iterator arithmetic to do a binary search.

- A binary search looks for a particular value in a sorted sequence
- Algorithm used is to look at element at midpoint of sequence
  - If value is one we want, we are done;
  - If value is larger than one we want then calculate new midpoint of smaller sequence before current midpoint;
  - If value is smaller than one we want then calculate new midpoint of smaller sequence that is after midpoint.

Example iterator arithmetic

BinarySearch.cpp

```
1  // vector<string> words;  has already been defined and
       filled above
2  string valueSearched = "something";
3  auto first = words.begin();
4  auto last = words.end();
5  auto mid = words.begin() + (last-first)/2;
6  while ( mid != last && *mid != valueSearched ){
7    if ( valueSearched < *mid ){
8      last = mid;
9    } else {
10     first = mid + 1;
11   }
12   mid  = first + (last-first)/2;
13 }
14 // mid now refers to element that contains same string
       as valueSearched
15 // or points to one past end of container.
```

Types of errors encountered

- *Compile-time errors*: are found by the compiler. Examples are syntax errors and type errors
- *Link-time errors*: errros found by linker when trying to combine object files into an executable program
- *Run-time errors*: found by checking the runing program. They could be detected by hardware, a library, user code.
- *Logic errors*: are mistakes in the code found by a programmer.

Goals in programming

Your programs should:

1. Produce the desired result for all legal inputs
2. Give reasonable error messages for all illegal inputs
3. Not worry about misbehaving hardware
4. Not worry about misbehaving system software
5. Terminate after finding an error

Acceptable software

Approaches to producing accepatable sofware incude:

1. Organizing sofware to minimize errors
2. Eliminating most errors by debugging and testing
3. Making sure remaining errors are not too serious

sources of error

- *Poor specification*: If we don't conisder all edge cases, we may miss some cases that should be handled in our code
- *Incomplete programs*: Need a way to know when all of the cases have been coded
- *Unexpected arguments*: Functions take arguments – if a function is given unexpected arguments need to handle that. (eg. sqrt(-1.2) can't return a correct double value)
- *Unexpected input*: Need ways to handle if a user enters a string when we request a number, etc.
- *Unexpected state*: need a way to check if the data available is complete enough for the computation being done in the program
- *logical errors*: The code simply doesn't do what it is supposed to do – these have to be found and fixed

Compile time errors – syntax errors

- Compiler is first defense against errors in your program
- Only if your program conforms to the language specification will it allow you to proceed
- Example syntax errors in call:

```
1 int area( int length, int width );
2 int s1 = area(7;  // error: missing)
3 int s2 = area(7)  // error: missing;
4 Int s3 = area(7); // error: Int is not a type
5 int s4 = area('7); // error: non-terminated
      character
```

- It is easy to correct these when they are found
- Typically the compiler message is a bit cryptic until you get used to them
- Try these errors out in a code, to see what error messages compiler gives

Compile time errors – type errors

```
1 int area( int length, int width );
2 int x0 = arena(7); // error: undeclared function
3 int x1 = area(7);  // error: wrong number of arguments
4 int x2 = area("seven",2); //error: wrong first
    argument type
```

Try these errors out in a code, to see what error messages compiler gives

Non-errors

```
1 int x4 = area(10,-7); // ok, but what is width -7?
2 int x5 = area(10.7,9.3); // ok, but truncates to 10
    and 9
3 int x6 = area(100,9999); //get overflow of int in area
```

- As you get more practice, you'll get your code to compiler easier
- Just because your code compiles, doesn't mean it will run

Link-time errors

- Every function must be defined exactly once
- It must be decalared exactly the same in every file that uses it – we do that with include files
- Example of a possible link-time error:

```
1       int area( int length, int width);
2       int main(){
3         int x = area(2,3);
4       }
```

- Only an error if area is not defined in another file linked to this one
- Also, area function must e defined with same types, ie:

```
1       int area( int length, int width){ return
            length*width; }
```

Run-time errors

- Consdier this code that compiles:

```
1  int area( int len, int wid ){
2    // calculate area of rectangle
3    return len*wid;
4  }
5  int framed_area( int x, int y ){ // area in frame
6    return area( x-2, y-2 );
7  }
8  int main(){
9    int x = -1, y=2, z=4;
10   /* ... */
11   int area1 = area(x,y);
12   int area2 = framed_area(1,z);
13   int area3 = framed_area(y,z);
14   double ratio = double(area1)/area3;
15   return 0;
16 }
```

Run-time error example notes

- Calls to area functions use variables (good) – but could hide negative values
- area1 = -2, area2 = -4 and area3=0
- ratio gets a divide by zero error
- Two ways to fix this:
  - Let the caller of area() deal with bad argurments
  - Let area() deal with bad arguments

Error handling

- we can `throw` an exception
- here we wrap the `throw` call in a function called `error`, that is defined in "std_lib_facilities.h":

```
1 #include <exeption>
2 void error( const string & s ){
3   throw runtime_error( s );
4 }
```

- The function takes a string as argument, and doesn't return a value
- If the runtime_error isn't caught by the calling code, then the program will exit

Error handling – caller deals with errors

- We can protect calls to area by checking values passed to it:

```
if ( x<=0 || y <=0 ) error("non-positive area()");
int area1 = area(x,y);
```

- Protecting call to framed_area requires check like:

```
if ( y<=2 || z<=2 ) error("non-positive
    framed_area()");
int area3 = framed_area( y, z );
```

- This is a bit messy – we had to know that framed_area subtracts two – there is a hidden "magic number"
- If someone later changes framed_area to only subtract 1, need to modify all of the checks
- Code that breaks easily with a small change like this is called "brittle"

Error avoidance

- Use const variable instead of magic number:

```
const int frame_width = 2;
int framed_area( int x, int y ){
  return area( x-frame_width, y-frame_width );
}
```

- Then our error handling can use the const value:

```
if ( y <= frame_area || z <= frame_area ) {
  error("non-positive framed_area");
}
int area3 = framed_area( y, z );
```

- Disadvantage of this method is that it add “ugly” code to check arguments everywhere the function is called.

Error handling – function deals with errors

- modify the functions:

```
const frame_width=2;
int area( int x, int y){
  if ( x <= 0 || y <=0 ) error("non-positive
      area");
  return x*y;
}
int framed_area( int x, int y ){
  if ( x <= frame_width || y <= frame_width )
     error("non-positive framed_area");
  return area( x-frame_width, y-frame_width );
}
```

- Use this method when you can, since it should shorten your code
- Reasons given why you may not want to do this:
  - The function may be from existing library that you can't modify
  - The function doesn't know what to do in case of an error
  - Performance is an issue – checking takes extra time

Error reporting

- Throw an exception, or return a documented bad value
- Example 1 of returning bad value:

```
// ask user for a yes/no answer
// return 'b' to indicate a bad answer
char ask_yn( string question ){
  cout << question << "? (yes or no)\n";
  string answer=" ";
  cin >> answer;
  if ( answer == "yes" || answer == "y" ) return
      'y';
  if ( answer == "no" || answer == "n" ) return
      'n';
  return 'b';
}
```

Error reporting – second example

- Example 2 of returning bad value:

```
// calculate area of rectangle
// return -1 to indicate bad argument
int area( int length, int width ){
  if ( length <=0 || width <= 0 ) return -1;
  return length * width ;
}
```

- Disadvantages of this method are:
  - Both caller and function must test
  - Function may not have a good value to return as "bad"

Exceptions

- Exceptions seperate detection from handling of an error
- Function that finds an error `throws` exception
- Caller (direct or indirect) of function can `catch` exception by using a `try catch` block

```
class Bad_area{}; // type for errors from area()
int area( int len, int wid ){ // calc. area
  if ( len <= 0 || wid <= 0 ) throw Bad_area();
  return len*wid;
} //...
int main()
try {
  int x=-1;  int y=2;  int z=4;
  int area1 = area(x,y);
  int area2 = framed_area(1,z);
  int area3 = framed_area(y,z);
  dobule ratio = area1 / double(area3);
  return 0;
}
catch (Bad_area) {
  cout << "Oops! bad arguments to area()"<<endl;
  return 1;   }
```

Range errors

- Collections of data are kept in containers
- Most commonly useful standard container is `vector`
- What happens when we try to access an element that isn't in `vector v`'s range [0:`v.size()`) ?
- It throws an exception of type `out_of_range`
- Do you see an error below?

```
int main()
try {
  vector<int> v;
  for ( int x; cin >> x ) v.push_back(x);
  for ( int i=0; i <=v.size(); ++i )
    cout <<"v["<<i<<"]=="<<v[i]<<endl;
  return 0;
} catch ( out_of_range){
  cerr <<"Oops! Range eror\n";
  return 1;
} catch (...){
  cerr <<"Exception: something went wrong\n";
  return 2;
}
```

Dealing with bad input

- On bad input we use the "error" function:

```
void error ( const string & s ){
  throw runtime_error( s );
}
// we can catch the runtime_error in our main
    function:
int main()
try {
  //... our program
  return 0; // 0 indicates success
}
catch ( runtime_error & e ){
  cerr << "runtime_error:" << e.what() << endl;
  return 1;
}
```

Dealing with bad input, notes

- `e.what()` extracts the error message from the `runtime_error`
- You can read more on references (the &) in section 8.5.4-6
- Catching the `runtime_error` above doesn't catch the `out_of_range` error
- We can use several catch blocks to look for the different types of exception that might be thrown
- Note that std_lib_facilities.h also defines `void error( string &s1, string &s2)` that allows passing a second string:

```
void error( const string & s1, const string &s2){
  throw runtime_error( s1 + s2 );
}
```

Narrowing errors

- When a variable is implicitly converted from a large variable to one that is smaller (eg. `double` to `int`) and information will be lost
- We can use `narrow_cast` to try to catch narrowing errors:

```
int x1 = narrow_cast<int>(2.9); // throws
int x2 = narrow_cast<int>(2.0); // ok
char c1 = narrow_cast<char>(1066); // throws
char c2 = narrow_cast<char>(85); // ok
```

- The <...> brackets are the same as those used for `vector` – they are used to specify a type, rather than value

Logic errors

- Consider this code to find the lowest, highest and average temperature:

```
int main(){
  vector<double> temps;
  for ( double temp; cin >> temp; )
     temps.push_back( temp );
  double sum = 0, high_temp = 0, low_temp=0;
  for ( int x : temps ){
    if (x>high_temp) high_temp = x; //find high
    if (x<low_temp) low_temp =x; //find low
    sum += x;
  }
  cout<<"High temperature:"<<high_temp<<endl;
  cout<<"Low temperature:"<<low_temp<<endl;
  cout<<"Average
      temperature:"<<sum/temps.size()<<endl;
  return 0;
}
```

- Note initial values of high and low – what happens if temperatuers are never below or above zero?

Estimation

- How do you check that your program gives the correct answer?
- You should estimate what you expect to get to answer the question:
- Is this answer to the problem plausible?
- Example: what would you expect to get for the area of a regular hexagon with 2cm side? If we get an answer 10.55 is the answer right?
- (No it should be: $6 + 4\sqrt{2} = 11.664$ – draw it and check)

Error checking : insert invariants

- An invariant is a condition that should always hold
- An example invariant check:

```
int my_complicated_function(int a, int b, int c)(){
  // the arguments are positive and a<b<c
  if ( !( 0<a && a<b && b<c ) ){
    error("Bad arguments for mcf");
  }
  //...
}
```

- There are many valid ways of searching for bugs
- Its best to keep tidy, well formatted code to help decrease the debugging time

Pre-conditions

- A requirement on a function's arguments is called a *pre-condition*.
- It is a condition that must be met for the function to perform properly
- Document pre-conditions for functions in the comments
- Apply checks and give an error if conditions are not met
- You can always remove the checks after you finish debugging if they impose too searious of a performance burdent

Post-conditions

- *Post-conditions* check that the return value is in a reasonable range, for example:

```
int area( int len, int wid ){
  // calculate area of rectangle
  // pre-conditions: len and wid are positive
  // post-condition: returns a positive value that
      is the area
  if ( len <=0 || wid <=0 ) error("area()
      pre-condition");
  int a = len * wid ;
  if ( a<=0 ) error("area() post-condition");
  return a;
}
```

- Q: Is there a pair of values so that the pre-conditinos are met, but not the post-conditinon?

Testing

- Your program is ready once you find “The last bug”
- “The last bug” is a programmer’s joke – it is impossible to know when you’ve found it
- Testing is executing a program with a large set of systematically chosen inputs and comparing the results to what we expect
- Systematic testing with millions of different inputs can’t be done by humans – should be automated somehow.

Debugging

- After you finish writing program, often will have errors
- Finding and fixing the errors (bugs) is called debugging
- Debugging steps:
  1. Get program to compile
  2. Get program to link
  3. Get program to do what its supposed to do
- Debugging advice:
  1. Report errors using the "error()" function and catch the exception in main()
  2. Make your program easy to read
  3. Add comments to give details about the code that you can't get from reading the code itself (ie. what is it supposed to do?)
  4. Use meaningful variable and function names
  5. Use a consistent code layout
  6. Break code into small funcitons – each expressing a logical action
  7. Use library facilities rather than your own code when you can

Common compile time errors

- Is every string terminated?

```
cout << "Hello , << name << '\n'; //oops
```

- Is every character literal terminated?

```
cout << "Hello , " << name << '\n; //oops
```

- Is every block terminated

```
int f( int a ){
  if ( a>0 ){
    /* do something */
  else { //oops
    /* try something else */
  }
  //...
}
```

More common compile time errors

- Is every set of parentheses matched?

```
if ( a<=0    //oops
   x = f (y );
```

- Is every name declared?
  - Did you include needed header files?
  - Is every name declared before it is used?
  - Did you spell every name correctly and with consistent capitalization?
- Did you terminate each expression statement with a semicolon?

```
x = sqrt(y)+2 // oops
z = x + 2;
```

Debugging tools

- Sometimes you can find the problem by adding more printouts
- You can also use a tool called a debugger to step through the code line by line, and look at variable contents at each step
- Example adding temporary printouts to say where you get to in the code:

```
int my_fcn( int n, double b){
  int res=0;
  cerr << "my_fct("<<a<<", "<<b<<")\n";
  /*...*/
  cerr<<"my_fct() returns "<<res<'\n';
  return res;
}
```

Arrays

- Arrays are like fixed sized `vectors` (whose size cannot be changed)
- Arrays are built-in to C++ and C – no include file is needed to use arrays
- An array declaration is of the form `arr[ nelems ]`, where `arr` is the name being defined and `nelems` the number of entries in the array (>0).
- Example definitions and initialization of arrays:

```
1 unsigned nmax = 99;
2 const unsigned nelems = 99;
3 int arr[10]; // array of 10 ints
4 int *parr[nelems]; // array of 99 pointers to int
5 string sarr1[nmax]; // error: nmax not constexpr
6 // below ok only if get_size() is constexpr
7 string sarr2[ get_size() ];
```

Initializing array elements

- We can list initialize elements in an array.
- Size of array must match number of elments in the list initialization; size can be omitted when doing list init

```
1 const unsigned nelems = 3;
2 int arr1[ nelems ] = { 0, 1, 2 };
3 int arr2[] = {4, 5, 6, 7};
4 int arr3[5] = { 0, 1, 2}; // err: need 5 initialisers
5 string arr4[2] = {"hi", "bye"};
6 // below error: need only 2 initializers
7 int arr5[2] = { 0, 1, 2};
8 int arr6[7] = {}; // initialize with 7 zeros
```

Character arrays

- Character arrays can also be initialized from string literals
- Careful to remember string literals are null ('\0') terminated
- There is no copy initializer for arrays

```
char c1[] = { 'H', 'e', 'l', 'l', 'o' };
char c2[] = { 'H', 'i', '\0' }; // null terminated
char c3[] = "Hello"; // also null terminated
char c4[6] = "Hello!"; // error: no space for null
char c5 = c1; // error: no copy init
c1 = c2; // error: no array copy
```

Pointers and references and arrays

- Arrays can hold objects of most any type
- An array is an object, so we can define pointers and references to arrays
- Arrays can be defined to hold pointers or reference to an array

```
1 int arr[10];
2 int * par1[10]; // par is an array of 10 pointers to
    int
3 int &refs[10] = ?; // error: no arrays of references
4 int (*par2)[10] = &arr[0]; // par2 is pointer to array
    of 10 ints
5 int (&par3)[10] = arr; // par3 is a reference to array
    of 10 ints
6 int *(&par4)[10] =par1; // par4 is a ref to array of
    10 pointers to int
```

Accessing array elements

- An array of size N has indices from 0 to N-1.
- The [] operator is used to access the contents of an array
- Example : re-implementation of `CountGrades.cpp`

```
unsigned count[11] = {}; // initialize with zeros
unsigned grade;
while ( cin >> grade ){
  if ( grade <= 100 ) {
    count[ grade / 10 ] ++ ;
  }
}
```

- Care must be taken to check the subscript to an array is within the bounds of the size of the array.

Pointers and arrays

- The array name typically becomes a pointer to the first element of an array.
- We can then do pointer arithmetic to get elements other than the first element. For example:

```
1 string colors[] = {"red", "green", "blue"};
2 string * pcols = colors; //pointer to first
      element in colors
3 string * pcols2 = &(colors[0]); // same as pcols
4 string * pc1 = pcols+1; // pc1 now points to
      colors[1]
5 string * pc2 = pcols+2; // pc2 now points to
      colors[2]
6 pcols++; // pcols now points to colors[1]
```

Pointers are iterators

- Same iterator operations work on arrays as did on vectors and strings
- Example iterating over array:

```
int arr[] = { 10, 20, 30, 40, 50, 60, 70, 80, 90,
    100};
int *pbeg = arr;// pointer to first element in arr
int *pend = &arr[10];// pointer one past last
    element in arr
for (int *i = pbeg; i != pend; i++){
  cout << *i << endl; // print element in arr
}
```

- Can also use use library `begin()` and `end()` functions to help get front and back iterators to built in arrays. Using same `arr` as above:

```
for ( auto i = begin(arr); i != end(arr); i++ ){
  cout << *i << endl;
}
```

Pointers arithmetic

- Pointers to array elements can use all usual iterator operations
- When we add (subtract) an integer value to (from) a pointer, the result is a new pointer to that number of elements ahead (behind) in the array.
- Subscript to array should normally be defined as `size_t` type, declared in `cstddef` header file.

```
1 #include <cstddef>
2 const size_t nvals = 5;
3 int arr[ nvals ] = { 0, 1, 2, 3, 4 };
4 int *ip = arr; // equivalent to &arr[0]
5 int *ip2 = ip + 4; // equivalent to &arr[4]
6 int *pend = ip + nvals;// ok: but dont dereference
7 int *ip3 = ip + 10; // error: arr only has 5 values
8 auto n = end(arr) - begin(arr); // n is 5
```

Dereferencing and pointer arithmetic

- Result of adding value to a pointer is a pointer
- Assuming resulting pointer still points to an element we can dereference it.

```
int arr[] = { 0, 1, 2, 3, 4};
int i = arr[2];
int *p = arr;
i = *(p+2); // equivalent to i = arr[2]
p = &arr[2]; // p redefined to point to arr[2]
int j = p[1]; // p[1] is arr[3], j=3
int k = p[-2]; // p[-2] is arr[0], k=0
```

- Note that built in subscript operator can be negative.
- Again: subscript must refer to a location with an element that has been defined.

Multidimensional arrays

- Multidimensional arrays are really arrays of arrays:

```
int arr1[10][4]; // array of size 10; each element
    is an array of 4 int
// below: array of size 2; each element is array
    of 4 elements,
// whose elements are arrays of 6 elements.
int arr2[2][4][6]={0};   // all elements init to 0
```

- List initializing elements of multi-dim. arrays:

```
int arr3[4][3] = {
  {0, 1, 2 },
  {3, 4, 5 },
  {6, 7, 8 },
  {9, 8, 7 }
};
int arr4[4][3] = {
  0, 1, 2, 3, 4, 5,
  6, 7, 8, 9, 8, 7 };
// initialize element 0 in each row:
int arr5[4][3] ={ {0}, {1}, {2}, {3} };
```

- Nested braces are optional, though help with clarity

Example accessing elements of 2-dim array

- Use a nested `for` loops to access elements of multi-dim arrays
- One `for` loop indexes row, second indexes column. Example, fill a 2-d array with multiplication table.

MultTable.cpp

```
const size_t nRows = 10;
const size_t nCols = 5;
int multTable[ nRows ][ nCols ];
for ( size_t i = 0 ; i != nRows; i++ ){
  for ( size_t j = 0 ; j != nCols ; j++ ){
    multTable[i][j] = (i+1)*(j+1);
  }
}
```

Range `for` loop for multi-d arrays

- Can use range for loop to traverse multi-dim array using a nested loop. Example setting each element as its index in overall array.

```
1  const size_t nRows = 10;
2  const size_t nCols = 5;
3  size_t idx = 0 ;
4  int multTable[ nRows ][ nCols ];
5  for ( auto & row : multTable ){
6    for ( auto & col : row ){
7      col = idx;
8      idx++;
9    }
10 }
```

- Note that to use multi-dim array in a range `for`, loop control variable for all but the innermost loop must be references.

Pointers and multi-dim arrays

- When defining pointers to multi-dim array, remember that it is really an array of arrays.
- Also note the use of brackets below to differentiate an array of pointers from a pointer to an array

```
1 int arr[3][4];
2 int (*p)[4] = arr; // p points to first row :
      array of four ints
3 p = &arr[2]; // p now points to last row : array
      of four ints
4 int *ip[4];  // array of pointers to int
5 int (*ip2)[4]; // pointer to an array of four ints
```

A little matrix algebra

- Matrices are used to represent systems of linear equations.
- A matrix of dimensions $n \times m$ has n rows and m columns.
- For example u is a 3×1 matrix:

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

- Below A and C are 3×3 matrix:

$$A = \begin{pmatrix} 1 & 5 & 0 \\ 7 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \quad C = \begin{pmatrix} 2 & 4 & 6 \\ -1 & -4 & 5 \\ 9 & 0 & 4 \end{pmatrix} \tag{1}$$

- Adding two matrices, requires the two matrices to have the same dimensions, and the result is to add the corresponding components of the matrices. In above $A + C$ results in:

$$A + C = \begin{pmatrix} 3 & 9 & 6 \\ 6 & -3 & 7 \\ 9 & 0 & 5 \end{pmatrix}$$

Multiplying matrix by a scalar

- Multiplying a matrix by a scalar (a single number), multiplies every element of the matrix by that number. Eg. Multiply A by 3 results in:

$$3A = \begin{pmatrix} 3 & 15 & 0 \\ 21 & 3 & 6 \\ 0 & 0 & 3 \end{pmatrix}$$

Matrix multiplication

- Only defined iff number of columns in left matrix matches the number of rows in right matrix
- Multiplying an $m \times n$ matrix, Q, by an $n \times p$ matrix, R, results in an $m \times p$ matrix with elements:

$$[QR]_{ij} = \sum_{r=1}^{n} Q_{ir} R_{rj}$$

- Example 1: Using matrices on previous page Au is a 3×3 matrix multiplied by a 3×1 matrix with elements:

$$\begin{pmatrix} [Au]_{11} \\ [Au]_{21} \\ [Au]_{31} \end{pmatrix} = \begin{pmatrix} A_{11}u_{11} + A_{12}u_{21} + A_{13}u_{31} \\ A_{21}u_{11} + A_{22}u_{21} + A_{23}u_{31} \\ A_{31}u_{11} + A_{32}u_{21} + A_{33}u_{31} \end{pmatrix} = \begin{pmatrix} 1 + 10 + 0 \\ 7 + 2 + 6 \\ 0 + 0 + 3 \end{pmatrix} = \begin{pmatrix} 11 \\ 15 \\ 3 \end{pmatrix}$$

Matrix multiplication code example

- Find AB when A and B are:

$$A = \begin{pmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 4 & 6 & 8 \\ -1 & -2 & -3 & -4 \end{pmatrix}$$

- Note that we need three nested for loops to calculate the elements:

$$[AB]_{ij} = \sum_{r=1}^{n} A_{ir} A_{rj}$$

- One for loop over i, a second over j, and a third over r the number of columns in A (rows in B)

Matrix multiplication code example

MatrixMult.cpp

```
const size_t Nm = 4, Nr = 2, Nn = 4;
double A[Nm][Nr] = { { 1.0, 5.0 },
                     { 2.0, 6.0 },
                     { 3.0, 7.0 },
                     { 4.0, 8.0 } };
double B[Nr][Nn] = { { 2.0, 4.0, 6.0, 8.0 },
                     {-1.0,-2.0,-3.0,-4.0 } };
// A is 4x2, B is 2x4, therefore AB is 4x4
double AB[4][4] = {};
// [AB]_ij = sum( r=1 to n, A_ir Brj );
// i labels row of AB
// j labels col of AB
// r labels sum for each entry in new matrix
for ( size_t i = 0 ; i < Nm ; i++){
  for ( size_t j = 0 ; j < Nn ; j++ ){
    for ( size_t r = 0 ; r < Nr; r++ ){
      AB[i][j] += A[i][r] * B[r][j];
    }
  }
}
```

Week3 Done!

- Homework 2 given out last week due Sept. 21 by 17:00
- Homework 3 given out today due Sept. 28 by 17:00
- Lets work on these together now!