

Week5 : Statements, Arrays, and Functions

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 5

Outline

Statements

Arrays

Functions

Outline

Statetements

- Technicalities

- Introduction to statements

- Statement scope

- Conditional statements

- Iterative statements

- Jump statements

- Exception handling

Arrays

Functions

Overview of tonight

- ▶ While we want to focus on programming, we need to learn the language as well
- ▶ Language is only a tool
- ▶ C++ standard is 1300+ pages and written for experienced programmers
- ▶ We won't be so complete and comprehensive in our study
- ▶ Tonight we will look at some of the details for statements, declarations, definitions, arrays and functions
- ▶ This week is nominally based on Ch. 8 of the textbook, however the slides were made from notes on a different textbook
- ▶ The slides and textbook complement each other – do read Ch. 8, as it will help you review and gain further understanding

Simple statements

- ▶ Most statements in C++ end with a semicolon;
- ▶ **Expression statements** cause the expression to be evaluated and its result is discarded:

```
1 // useless statement:  
2 // add 5 and inum, throw away result  
3 inum + 5;  
4 // more useful statement, print to screen:  
5 cout << inum << endl;
```

- ▶ **Null statement** or empty statement is a single semicolon:

```
1 ; // null statement
```

- ▶ **Null statement** can be useful in a loop where we do not want to do anything in the body of the loop:

```
1 while ( cin >> inum && inum > 0 )  
2 ; // do nothing in loop
```

null statement caveats

- ▶ Null statements should be commented to make them easier to see
- ▶ Null statement is legal anywhere a statement is expected.

```
1 inum += 5;; // ok. second semicolon is null
   statement
```

- ▶ Not all null statements are harmless:

```
1 vector<int> vec;
2 while( cin >> inum && inum > 0 ) ;
3     vec.push_back(inum);
4 if ( ! vec.empty() ) ;
5     cout << vec[0] << endl;
```

- ▶ Above while body has empty statement ; as its body
- ▶ Only the last integer read in (the first negative value) is put in the vector
- ▶ Probably code was supposed to add integers to vector until a negative value was entered
- ▶ Also body of if statement is empty statement ;
- ▶ Meaning that check for empty vector doesn't stop printing of first element of an empty vector

Code blocks

- ▶ **Compound statements**, or a code block, is a sequence of statements surrounded by curly braces { }
- ▶ A code block may be an **empty statement** as a pair of curly braces { }
- ▶ A code block is a scope – names introduced inside a block are only in that block and blocks nested inside that block.
- ▶ Code blocks are used when language requires a single statement, but we need to put more than one statement
- ▶ For example, the body of a **while** or **if** must be a single statement, but we can use more than one statement by enclosing the statements in curly braces
- ▶ Code blocks are *not* terminated by a semicolon:

```
1 int num = 0, tot = 0;
2 while ( num <= 10 ) {
3     tot += num;
4     ++num;
5 } // no semicolon
```

- ▶ We can use an empty block in place of a null statement:

```
1 // read until a non-negative integer is entered
2 while ( cin >> num && num < 0 )
3     { } // empty block
```

Scope

- ▶ Variables defined in control structure of `if`, `switch`, `while`, and `for`:
 - ▶ only within that statement
 - ▶ go out of scope when the statement ends
- ▶ If we need those variables outside of the control structure, define them before the control structure

```
1 // Example 1: variable i only seen inside while
  loop
2 while ( int i = readnumber() )
3     cout << i << endl;
4 i = 0 ; // error: i not accessible outside loop
5
6 // Example 2: define variable before loop
7 auto iter = vec.begin();
8 while ( iter != vec.end() && *iter >= 0 )
9     ++iter;
10 // below is okay, iter defined outside while loop
11 if ( iter = vec.end() )
12     cout << "No negative values in vector" << endl;
```


if Statement

- ▶ **if** statement conditionally executes another statement depending on whether a condition is **true**.
- ▶ **if** statement can optionally have **else** statement to be executed if the condition is **false**

```
1 // Simple form of if statement:  
2 if ( condition )  
3     statement1;  
4  
5 // Form for if else statement:  
6 if ( condition )  
7     statement1;  
8 else  
9     statement2;
```

- ▶ condition must be enclosed in parentheses, and must be convertible to a **bool** type
- ▶ If condition is **true** evaluate statement1, otherwise evaluate statement 2
- ▶ statement1 and statement2 must be a single statement
- ▶ The single statement may be a code block (that way more than one statement is evaluated)

if statement example

- ▶ Consider problem of assigning letter grade to a percentage for some hypothetical grading scheme:

```
1 unsigned grade = 0;
2 vector<string> letters= {"F", "D", "C", "B", "A",
   "A++"};
3 string lettergrade;
4 if ( cin >> grade ){
5     if ( grade < 60 )
6         lettergrade = letters [0];
7     else
8         lettergrade = letters [ (grade-50)/10 ];
9 }
```

- ▶ We can nest if statements, for example if we want to add a + or minus to the grades:

```
1 if ( grade % 10 > 7 )
2     lettergrade += '+';
3 else if ( grade % 10 < 3 )
4     lettergrade += '-';
```

Careful to use block if needed

- ▶ Below notice that we put the else statement inside a block to allow more than one statement to be evaluated

```
1  if (grade < 60)
2      lettergrade = letters [0];
3  else {
4      lettergrade = letters [(grade - 50)/10]; //
5          fetch the letter grade
6      if (grade != 100) // add plus or minus only
7          if not already an A++
8              if (grade % 10 > 7)
9                  lettergrade += '+'; // grades ending
10                     in 8 or 9 get a +
11             else if (grade % 10 < 3)
12                 lettergrade += '-'; // grades ending
13                     in 0, 1, or 2 get a
14     }
```

Careful to use block if needed

- ▶ Below notice that we **forgot** to put the else statement inside a block resulting in **incorrect** evaluation

```
1 if (grade < 60)
2     lettergrade = letters [0];
3 else
4     lettergrade = letters [(grade - 50)/10]; //
5         fetch the letter grade
6     // Below is not inside the else statement!
7     // Instead, even failing grades could get a +
8     // or - added
9     if (grade != 100) // add plus or minus only
10        if not already an A++
11            if (grade % 10 > 7)
12                lettergrade += '+'; // grades ending
13                    in 8 or 9 get a +
14            else if (grade % 10 < 3)
15                lettergrade += '-'; // grades ending
16                    in 0, 1, or 2 get a
```

- ▶ To avoid these types of errors:
always use braces after an if or else statement

Dangling else statement

- ▶ When we nest if statements we need to know which if each else statement belongs to
- ▶ Each else is matched with the closest preceding unmatched if
- ▶ Indentation should be made to indicate our intent.

```
1 // example of else that is wrongly indented
2 if ( grade % 10 >= 3 )
3     if ( grade % 10 > 7 )
4         lettergrade += '+';
5 else
6     lettergrade += '-';
7
8 // same example but with correct indentation
9 if ( grade % 10 >= 3 )
10     if ( grade % 10 > 7 )
11         lettergrade += '+';
12     else
13         lettergrade += '-';
14
15 // same example with braces to aid clarity
16 if ( grade % 10 >= 3 ) {
17     if ( grade % 10 > 7 ) {
```

Avoid dangling `else` using braces

- ▶ Adding braces can make it clear which `else` each `if` is paired with

```
1 // same example with braces to aid clarity
2 if ( grade % 10 >= 3 ) {
3     if ( grade % 10 > 7 ) {
4         lettergrade += '+';
5     }
6 } else {
7     lettergrade += '-';
8 }
```

switch statements

- ▶ **switch** statement provides way of selecting among a possibly large number of alternatives
- ▶ can be used to replace long string of nested if then else statements.

countVowels.cpp

```
// count vowels that are input
unsigned Na = 0, Ne = 0, Ni = 0, No = 0, Nu = 0;
char ch;
while ( cin >> ch ) {
    switch ( ch ) {
        case 'a': ++Na; break;
        case 'e': ++Ne; break;
        case 'i': ++Ni; break;
        case 'o': ++No; break;
        case 'u': ++Nu; break;
    }
}
cout << "Number of: a = " << Na << " e = " << Ne
      << " i = " << Ni << " o = " << No
      << " u = " << Nu << endl;
```

Notes on `switch` statements

- ▶ `switch` evaluates expression in `()` after `switch`, converting it to **integer** type
- ▶ Result of evaluation is compared with value associated with each **case** label that must be an **integer**.
- ▶ If the expression matches the value of the **case** label, execution begins with the first statement following that label.
- ▶ Execution continues from that statement including any statements after subsequent **case** labels
- ▶ **n.b.** execution flows across **case** labels.
- ▶ After a **case** label is matched, all remaining **cases** are executed unless a **break** statement is found.
- ▶ **break** statement is used to exit the **switch** statement, rather than executing statements in next case.
- ▶ If no matching **case** label is found, execution continues to the first statement after the `switch` statement

Another switch example

- ▶ We can use flow-through of case statements to execute a particular code if one of several integer-like values is found
- ▶ For example lets count total number of vowels:

```
1 unsigned Nvowels = 0;
2 unsigned Nother = 0;
3 char ch;
4 while ( cin >> ch ){
5     switch ( ch ) {
6         case 'a': case 'e': case 'i':
7         case 'o': case 'u':
8             ++Nvowels;
9             break;
10        default:
11            ++Nother;
12            break;
13    }
14 }
```

- ▶ Special case label **default** has its statements executed if no other **case** label matches the **switch** expression.

Variable definitions in switch

- ▶ `switch` execution can jump across `case` labels
- ▶ It is illegal to jump from a place where variable with initializer is out of scope to a place where variable is in scope.
- ▶ For example:

```
1 switch (ch){
2     case 'a':
3         int aVal = 0; // error: var init maybe bypassed
4         string s; // err: string implicit init
5         int aaVal; // ok: aaVal not initialized
6         break;
7     case 'b':
8         aaVal = get_number(); //ok: aaVal is in scope
9     case 'c':
10        {
11            int cVal = 0; // ok: inside statement block
12            // ...
13        }
14 }
```

- ▶ Anyhow, it is best to define variables outside the `switch` statement

while statements

- ▶ While statement executes a section of code so long as a condition is true
- ▶ Form of while statement is:

```
1 while ( condition )  
2     statement ;
```

- ▶ if **condition** evaluates to **true** then **statement** is executed
- ▶ if **condition** evaluates to **false** then next line after while loop is executed
- ▶ **condition** may not be empty
- ▶ Condition can be an expression or an initialized variable declaration – however note that these variables are created and destroyed on each iteration of loop.

while loop use

- ▶ While loop is useful when we want to iterate indefinitely – such as when reading an unknown number of inputs
- ▶ Example:

```
1 vector <int> v;  
2 int i;  
3 while ( cin >> i )  
4     v.push_back( i );  
5 auto iter = v.begin();  
6 while ( iter != v.end() && *iter >= 0 )  
7     ++iter;  
8 if ( iter == v.end() ){  
9     // we know all elements in v are >= 0  
10 }
```

Traditional for statements

- ▶ Used to loop a fixed number of times, is of the form:

```
for ( initializer ; condition ; expression )  
    statement ;
```

- ▶ **initializer** is used to initialize or assign a starting value that is modified over course of the loop
- ▶ **condition** serves as the loop control
 - ▶ If **condition** is **true** then **statement** is executed
 - ▶ If **condition** is **false** then continue execution with next line after the for loop
- ▶ **expression** is evaluated after each iteration of the loop.
- ▶ **statement** can either be a single statement or { a compound statement } in a block

Traditional for loop example

```
1 for ( decltype( s.size() ) i = 0 ;  
2     i != s.size() && ! isspace( s[i] ) ; ++i )  
3     s[i] = toupper( s[i] ) );
```

- ▶ first **initializer** is executed once at start of loop – in this example defines `i=0`
- ▶ next **condition** is evaluated
 - ▶ in this case it checks if `i` is not equal to `s.size()` and that first character in string is not a space
 - ▶ if above condition is **true** statement is evaluated
 - ▶ if above condition is **false** execution continues at the next line after the for loop
- ▶ In this case if the condition is **true** the character in the string at position `i` is changed to uppercase
- ▶ Finally expression is evaluated – in this example `i` is incremented by one

Multiple counters in for loop

- ▶ Like other declarations, for loop initialization can define several objects (that must have the same base type)
- ▶ For example we might copy only positive values from one vector to another:

```
1 vector<int> v1, v2;
2 int j;
3 cout << "enter numbers separated by spaces or
   ctrl-z:" << endl;
4 while ( cin >> j ) {
5     v1.push_back( j );
6 }
7 for ( auto n = v1.size(), i = 0; i != n; ++i ){
8     if ( v1[ i ] > 0 ) v2.push_back( v1[ i ] );
9 }
```

Omitting parts of the for header

- ▶ Can omit any (or all) of initialization, condition or expression in header of `for` loop
- ▶ In example below we get iterator to first entry in vector that is negative (or the end iterator):

```
1 auto iter = v.begin();
2 for ( /* no init */ ;
3     iter != v.end() && *iter >= 0; ++iter )
4     ; // null statement
```

- ▶ In example condition is left out, so we loop forever, unless there is a `break;` or `return;` statement to get us out:

```
1 for ( int i = 0 ; /* no condition */ ; ++i ){
2     // code in loop must stop the iteration somehow!
3 }
```

- ▶ Finally, can have no expression, such as if no counter is needed:

```
1 vector<int> v;
2 for ( int i; cin >> i ; /* no expression */ ){
3     v.push_back( i );
4 }
```


Range for loops in C++ 11

- ▶ Range for loop is used to iterate through elements of a container. Form of range for loop is:

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

- ▶ **expression** must represent a sequence type such as **vector**, **string**, array
- ▶ **expression** must have **begin** and **end** members that return iterators
- ▶ **declaration** defines a variable that it is possible to convert each element of the sequence to the variable's type
- ▶ on each iteration the **declaration** variable is initialized with the next iterator in the sequence
- ▶ If the **declaration** variable isn't the **end** iterator, then **statement** evaluated
- ▶ If the **declaration** variable is the **end** iterator, then execution continues at the next line after the for loop

Range for loop example

- ▶ Example: double the value of each element in a vector

```
1 vector<int> vec = { 1, -2, 3, -4, 5, -6, 7 };
2 // below auto is of type vector<int>::iterator &
3 for ( auto & curval : vec ) {
4     curval *= 2;
5 }
```

- ▶ Notice we defined curval as a reference, since we want to change the value in the vector
- ▶ Above range for loop is equivalent to traditional for loop:

```
1 for ( auto val = v.begin(), end = v.end();
2     val != end; ++val ){
3     auto &curval = *val;
4     curval *= 2;
5 }
```

do while statement

- ▶ Similar to `while` statement, but the condition is tested after the statement body completes – regardless of condition, the statement is executed at least once.
- ▶ Form for `do while` statement is:

```
1 do
2     statement;
3 while ( condition );
```

- ▶ `condition` cannot be empty
 - ▶ if `condition` is `true`, then `statement` is executed again
 - ▶ if `condition` is `false`, then execution continues on the next line after the `while`
- ▶ Note that `do while` ends with a semicolon after the parenthesized condition.

do while example

- ▶ Example where we sum numbers until user has had enough:

sumNumbers.cpp

```
1 char yesno; // use in cond. must be outside "do"
2 do {
3     cout << "Enter two numbers: ";
4     int n1 = 0, n2 = 0;
5     cin >> n1 >> n2;
6     cout << "Sum of " << n1 << " and " << n2
7         << " is " << n1 + n2 << endl;
8     do {
9         cout << "Do another sum (y or n)? ";
10        yesno = ' ';
11    } while ( cin >> yesno &&
12             yesno != 'n' && yesno != 'y' );
13 } while ( yesno != 'n' );
```

break statement

- ▶ `break` terminates nearest `while`, `do`, `for` or `switch`
- ▶ A `break` only affects the nearest enclosing loop or switch, for example:

```
1 string buf;
2 while (cin >> buf && !buf.empty()) {
3     switch(buf[0]) {
4         case '-':
5             // process up to the first blank
6             for (auto it = buf.begin()+1; it !=
7                 buf.end(); ++it){
8                 if (*it == ' ')
9                     break; // #1, leaves the for loop
10                    // . . .
11                }
12                // break #1 transfers control here
13                // remaining '-' processing:
14                break; // #2, leaves the switch statement
15            case '+': // . . .
16        } // end switch
17    // end of switch: break #2 transfers control here
18 }
```

continue statement

- ▶ `continue` statement terminates the current iteration of the nearest enclosing loop, and begins next iteration
- ▶ `continue` can only appear inside `for`, `while`, or `do while` loop
- ▶ `continue` interrupts current iteration, and execution continues by evaluating expression (in `for` loops), and then testing condition to do another loop
- ▶ For example read words, and do something only for capitalized words:

```
1 string s;  
2 while ( cin >> s && !s.empty() ){  
3     if ( ! isupper( s[0] ) ){  
4         continue;  
5     }  
6     // do something with words that start  
7     // with uppercase letter...  
8 }
```

goto statement

- ▶ **Programs should not use goto statements!**
- ▶ `goto` statement makes a jump to another statement in the same function.
- ▶ `goto label`, jumps to location with a labelled statement.
- ▶ `label` is any statement preceded by a colon:

```
1 //..
2 goto mylabelname;
3 int i = 1; // error: goto bypasses this
4 mylabelname:
5     i = 42; // error: goto bypassed declaration of i
6 // ...
7     if ( n < 0 ) goto end;
8 // ...
9 // labelled statement may be target of a goto
10 end: return;
```

Exceptions

- ▶ Exceptions are run-time problems such as:
 - ▶ losing database connection
 - ▶ unexpected input
 - ▶ overflow of arithmetic
- ▶ Exception handling is used to detect problem, signal what happened, and stop processing
- ▶ Code that raises an exception often has another part to handle whatever happened
- ▶ C++ exception handling involves:
 - ▶ `throw` expressions to indicate exception found
 - ▶ `try` blocks which have a handling part to deal with an exception using
 - ▶ `catch` clauses to try to handle exception.

throw expression

- ▶ Suppose we have some code to add two short int
- ▶ We can **throw** an exception, which will then indicate when we are about to overflow the maximum size

```
1 #include <climits>
2 #include <stdexcept>
3 short add( short a, short b ){
4     short maxb = SHRT_MAX - a;
5     if ( b > maxb ){
6         throw overflow_error( "add() result larger than
7             short " );
8     }
9     return a+b;
}
```

- ▶ Above we throw an expression that is an object of type `runtime_error`
- ▶ Throwing an exception terminates the current function and transfers control to a handler
- ▶ `overflow_error` is one of the standard library exception types defined in the `<stdexcept>` header
- ▶ `runtime_error` is initialized with a string or c-style character string to provide more info on the error

try blocks

- ▶ The general form of a `try` block is:

```
1 try {  
2     program-statements  
3 } catch ( exception-declaration1 ) {  
4     handler-statements1  
5 } catch ( exception-declaration2 ) {  
6     handler-statements2  
7 } // ...
```

- ▶ `try` keyword is followed by a block of statements enclosed in curly braces
- ▶ A `catch` has three parts:
 - ▶ keyword `catch`
 - ▶ declaration of an object (**exception declaration**)
 - ▶ a block of statements
- ▶ program-statements inside try block are normal logic of program
- ▶ variables declared within try block not accessible outside block

Example exception handler

shortAddTry.cpp

```
1 short a=0, b=0;
2 cout << "Enter two numbers :" << endl;
3 while ( cin >> a >> b ){
4     try {
5         short maxb = SHRT_MAX - a;
6         if ( b > maxb ){
7             throw overflow_error("add result larger than
8                 short");
9         }
10        cout << a << " + " << b << " = " << a+b << endl;
11        cout << "Enter two numbers :" << endl;
12    } catch ( overflow_error err ) {
13        cout << err.what()
14            << "Try again? y or n: " << endl;
15        char ch;
16        if ( cin >> ch && ch == 'n' ){
17            break; // break out of while loop
18        }
19    } // end catch block
20 } // end while block
```

Exception safe code

- ▶ Exception safe code is hard to write
- ▶ Exception bypasses part of program that would have run
- ▶ Objects may be left in incomplete state
- ▶ Resources may not be freed, and so on
- ▶ Programs that properly "clean up" in exception handling are **exception safe**
- ▶ Some programs use exceptions simply to terminate program when exception happens – in that way exception safety is not worried about.

Standard exception classes

- ▶ Classes defined in `<stdexcept>`:

<code>exception</code>	The most general kind of problem
<code>runtime_error</code>	Problem detected only at run time
<code>range_error</code>	Run-time error for results outside range of meaningful values
<code>overflow_error</code>	Computation overflow
<code>underflow_error</code>	Computation underflow
<code>domain_error</code>	Error in logic of program
<code>invalid_argument</code>	Error in argument passed to function
<code>length_error</code>	Attempt to make object larger than max size
<code>out_of_range</code>	Used value outside the valid range
- ▶ All initialized with a c-style character string (`const char*`)
- ▶ Get back string using `what()` method

Outline

Statements

Arrays

Multidimensional arrays

Functions

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 `type 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 elements 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`

```
1 unsigned count[11] = {}; // initialize with zeros
2 unsigned grade;
3 while ( cin >> grade ){
4     if ( grade <= 100 ) {
5         count[ grade / 10 ] ++ ;
6     }
7 }
```

- ▶ 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:

```
1 int arr [] = { 10, 20, 30, 40, 50, 60, 70, 80, 90,
                100};
2 int *pbeg = arr; // pointer to first element in arr
3 int *pend = &arr[10]; // pointer one past last
                      element in arr
4 for (int *i = pbeg; i != pend; i++){
5     cout << *i << endl; // print element in arr
6 }
```

- ▶ 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:

```
1 for ( auto i = begin(arr); i != end(arr); i++ ){
2     cout << *i << endl;
3 }
```

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.

```
1 int arr [] = { 0, 1, 2, 3, 4};
2 int i = arr [2];
3 int *p = arr;
4 i = *(p+2); // equivalent to i = arr [2]
5 p = &arr [2]; // p redefined to point to arr [2]
6 int j = p [1]; // p [1] is arr [3], j=3
7 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:

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

- ▶ List initializing elements of multi-dim. arrays:

```
1 int arr3[4][3] = {
2     {0, 1, 2 },
3     {3, 4, 5 },
4     {6, 7, 8 },
5     {9, 8, 7 }
6 };
7 int arr4[4][3] = {
8     0, 1, 2, 3, 4, 5,
9     6, 7, 8, 9, 8, 7 };
10 // initialize element 0 in each row:
11 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

```
1 const size_t nRows = 10;
2 const size_t nCols = 5;
3 int multTable[ nRows ][ nCols ];
4 for ( size_t i = 0 ; i != nRows; i++ ){
5     for ( size_t j = 0 ; j != nCols ; j++ ){
6         multTable[i][j] = (i+1)*(j+1);
7     }
8 }
```

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:

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

- ▶ 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];
        }
    }
}
```


Outline

Statements

Arrays

Functions

- Introduction to functions

- Argument passing

Introduction to functions

- ▶ A `function` is a block of code with a name
- ▶ We can execute that code by calling the function
- ▶ Function can have zero or more arguments (values we pass to and/or from the function)
- ▶ Function may have a return result
- ▶ Functions can be overloaded – meaning the same name may refer to several different functions.

Function basics

- ▶ function definition consists of:

```
1 return-type function-name( arg1, arg2, ... ){  
2     // function-body ...  
3 }
```

- ▶ Parameters of function are specified as comma-separated list in parentheses (arg1, arg2, ...)
- ▶ We call a function using **call operator** which is pair of parentheses
- ▶ Call operator takes expression that is a function or points to a function
- ▶ Inside parentheses of call operator is comma-separated list of arguments

Writing a function

- ▶ recall previous example of factorial function

```
1 double factorial( int an ){
2     if ( an < 0 ) return NAN; //not a number
3     double retval = 1.0;
4     for (int i = 2; i <= an; i++){
5         retval *= i;
6     }
7     return retval;
8 }
```

- ▶ Function is called **factorial**
- ▶ It takes one parameter that is an int
- ▶ It returns one value as a double

Calling a function

- ▶ To call our factorial function, we must supply it an int value

```
1 int main(){
2     // print the factorial of 10
3     int i = 10;
4     double fi = factorial( i );
5     cout << i << "! = " << fi << endl;
6     return 0;
7 }
```

Review: What happens when you call a function?

- ▶ Function's parameters from corresponding arguments are initialized
- ▶ Execution of the *calling* function is suspended, and execution of the *called* function begins
- ▶ Function execution ends when a **return** statement is encountered, and control is transferred back to the calling function.
- ▶ A return value is transferred back to the calling function.

Equivalent effect of calling factorial

- ▶ Equivalent code to our function call:

```
1 int i = 10; // initialize val from literal
2 double retval = 1.0; //
3 if ( i < 0 ){
4     retval = NAN;
5 } else {
6     for ( int j=2; j <= i ; j++ ){
7         retval *= j;
8     }
9 }
10 double fi = retval;
11 //...
```

Review: Parameters and arguments

- ▶ Arguments are initializers for a function's parameters.
- ▶ The first argument initializes the first parameter, and so on
- ▶ The type and number of parameters passed to a function must match the function definition

Example: single parameter factorial function

```
1 double ans=0.;
2 ans = factorial("fourty"); // error: wrong arg type
3 ans = factorial(); // error: too few arguments
4 ans = factorial(4,5,6); // error: too many
  arguments
5 ans = factorial(6.78); //ok: but 6.78 gets
  truncated to 6
```


Review: A function's parameter list

- ▶ Function needs to have parameter list, but it can be empty
- ▶ `void` keyword can be used in empty parameter list to be compatible with C
- ▶ Parameter list is a comma-separated list of parameters each of which looks like a declaration with a single named type
- ▶ Each parameter in list needs its own name

Example: function parameter lists

```
1 int f1() { /* ... */ } // implicit void par. list
2 int f2( void ) { /* ... */ } // explicit void par. list
3 int f3( int v1, v2 ) { /* ... */ } // error missing
  type of v2
4 int f4( int v1, int v2 ) { /* ... */ } // ok
```

Review: More notes on functions

- ▶ Function return types:
 - ▶ Most types can be used as a function return value
 - ▶ Return type can be declared `void` if no return value is needed
- ▶ Scope and lifetime of variables:
 - ▶ **Scope** of a name is the part of the program in which a name is visible
 - ▶ **Lifetime** of an object is the time during the program's execution that the object exists
 - ▶ variables defined in a function are local to the block of code; they are in the local scope of the function, and their lifetime ends when the function ends.

Automatic objects in functions

- ▶ Objects that exist only while block is executing are **automatic objects**
- ▶ Local variables and parameters of function are automatic variables
- ▶ automatic objects are created inside the function block and are destroyed at the end of the function call
- ▶ function parameters are initialized with the values of the arguments that the function was called with

Local static objects

- ▶ Defining a local variable as `static` makes its lifetime continue across calls to a function
- ▶ local `static` objects are initialized before the first time execution passes through the object's definition
- ▶ local `static` objects are not destroyed when the function ends – only destroyed when program terminates
- ▶ Example use of static counter:

```
1 size_t callcount() {
2     static size_t n = 0;
3     return ++n;
4 }
5 int main() {
6     for ( size_t i = 0; i != 100; ++i ) {
7         cout << callcount() << endl;
8     }
9     return 0;
10 }
```

- ▶ The program above prints 1 to 100 inclusive

Function declarations

- ▶ Name of a function must be declared before use
- ▶ Can be defined only once, but can be declared multiple times
- ▶ Can declare a function even if we never use that function
- ▶ Function declaration has no body, so no parameter names are needed, but can be provided to help with understanding.
- ▶ Example function declaration:

```
1 void processvec( vector<int>::const_iterator beg,  
2                 vector<int>::const_iterator end );
```

- ▶ Function declarations go in header files
- ▶ Source file that defines function should include the function's header file

Review: Reminder of compiling multiple sources

- ▶ Example:
 - ▶ main program `factMain.cpp` that has the `main` function (calls `factorial` and prints)
 - ▶ function declaration in `myfcns_factorial.h` and definition in `myfcns_factorial.cpp`
- ▶ First need to make object files for each source code:

```
# build factMain.o
g++ -c factMain.cpp
# build myfcns_factorial.o
g++ -c myfcns_factorial.cpp
```

- ▶ Then link the object files to build the executable:

```
# link factMain.o and myfcns_factorial.o
# to build factMain.exe
g++ -o factMain.exe factMain.o myfcns_factorial.o
```

Review: Compiler flags

- ▶ Can put additional options to the compiler as flags right after the compiler command
- ▶ Some useful flags are:
 - ▶ `-Wall` to give warnings for possible errors
 - ▶ `-std=c++11` to enable use C++ 11 standard features
 - ▶ `-O1`, `-O2`, `-O3` to enable increasing levels of compiler optimization of the code
 - ▶ `-g` to enable including debug information, which would get printed if the code crashes, or for viewing code in a debugger
- ▶ Example use of compiler flags:

```
g++ -Wall -std=c++11 -g -c myfcns_factorial.cpp
```

Passing arguments to functions

- ▶ Each time function is called, parameters are created and initialized by the arguments passed
- ▶ Parameter initialization works the same way as variable initialization
- ▶ The type of parameter determines how initialization is done
- ▶ If the parameter is a reference, then parameter is bound to its argument
- ▶ Otherwise, argument value is copied to the parameter
- ▶ When a parameter is a reference we say the argument is **passed by reference**
- ▶ When an argument value is copied, we say the argument is **passed by value**

Passing arguments by value

- ▶ When we initialize non-reference type variable, value of initializer is copied
- ▶ changes made to the variable have no effect on initializer

```
1 int num = 100;
2 int tot = num; // tot is a copy of value of num
3 num = 20;      // value in num is changed; tot
                 unchanged
```

- ▶ Passing by value works the same way – anything done to the parameter in the function does not change the argument

```
1 int add5( int num ){
2     num += 5;
3     return num;
4 }
5 int main(){
6     int val1 = 1;
7     int val2 = add5( val1 );
8     // val1 is still 1, val2 is 6
9 }
```

Passing arguments as pointers

- ▶ When we copy a pointer, value of the pointer is copied
- ▶ Pointer gives us indirect access to the object to which the pointer points
- ▶ We can change value of object pointed to by assigning to it through the pointer
- ▶ For example:

```
1 int i=10, j=20;
2 int *pi = &i , *pj = &j ;
3 *pi = 30; // value of i changed to 30
4 pi = pj; // pi now points to j
5 // setting pi=pj: values of i and j not changed
```

- ▶ same behaviour applies in pointer parameters

Example of passing arguments as pointers

- ▶ Example use of function that takes a pointer

```
1 void ptrfunc( int * ip ){
2     int j = 10;
3     *ip = 20; // changes value of object ip points to
4     ip = &j; // changes local copy of pointer
5 }
6 int main(){
7     int i = 30;
8     ptrfunc( &i ); // changes i but not address of i
9     cout << "i = " << i << endl; // prints 20
10    return 0;
11 }
```

- ▶ Programmers accustomed to programming in C often use pointer parameters to access objects outside of function
- ▶ C++ it is generally preferred to use reference parameters

Passing arguments by reference

- ▶ Recall that operations on a reference also act on the object to which the reference refers
- ▶ Example:

```
1 int a=10, b=20;
2 int &ra = a; // ra refers to a
3 ra = 30; // a is now 10
4 ra = b; // a is now 20 (same as b)
5 a = 40;
6 b = ra; // b is now 40 (same as a)
```

- ▶ Reference parameters to functions exploit this behaviour

Example of passing arguments by reference

- ▶ Example use of function that uses reference

```
1 void reffunc( int & rp ){
2     // changes value of object rp refers to
3     rp = 20;
4 }
5 int main(){
6     int i = 30;
7     reffunc( i ); // changes i
8     cout << "i = " << i << endl; // prints 20
9     return 0;
10 }
```

- ▶ **Using references avoids copies**
- ▶ We can access large objects using references rather than making copies
- ▶ Using references is often easier to read than using pointers

Another reference example

- ▶ Example use references for a function that compares two strings, which could be long

```
1 bool isStrShorter( const string &s1, const string
    & s2 ) {
2     return s1.size() < s2.size() ;
3 }
```

- ▶ Note use of **const** above as we do not intend to change the values of **s1** or **s2** in the function
- ▶ A function can only **return** one value, *however*
- ▶ Reference parameters let us effectively return multiple results

Example of multiple return values

- ▶ Here we use reference parameters to search string for both index to first occurrence of a character and count of occurrences of character

```
1 void char_loc( const string& s, char c,
2               string::size_type & loc,
3               string::size_type & count ){
4     count = 0;
5     loc = s.size();
6     for ( size_t i = 0; i != s.size() ; ++i ){
7         if ( s[i] == c ){
8             if ( loc == s.size() ) loc = i;
9             ++count;
10        }
11    }
12 }
```

- ▶ When we call above function need to give it four arguments:

```
1 string::size_type ns = 0, idx = 0;
2 string s = "garter snakes slither slowly";
3 char_loc( s, 's', ns, idx );
```

const parameters and arguments

- ▶ `const` object cannot be modified after initialization
- ▶ Example:

```
1 const int i = 10;
2 int j = i; // ok: we copy 10 into j
3 // below const refers to addr stored by pj,
4 // not the object pointed to
5 int * const pj = &j;
6 *pj = 20; // ok: copies 20 into j (non-const)
```

- ▶ Initialization rules for `const` apply in parameter passing

const in argument passing

- ▶ Example of passing `const` values to functions

```
1 int i = 10;
2 const int ci = i;
3 string::size_type n = 0;
4 ptrfunc( & i ); // calls function expecting int*
5 // error below: can't init int* from pointer to
   const int
6 ptrfunc( & ci );
7 reffunc( i ); // calls function taking ref to int
8 // error below: can't bind int reference to const
   int
9 reffunc( ci );
```

- ▶ **Use reference to const when possible**
- ▶ It is a mistake to define parameters that a function does not change as (plain) references

Effect of plain reference when not required

- ▶ Defining parameter as plain reference gives function caller misleading impression that function can change arg value
- ▶ Also cannot pass `const` object for that argument
- ▶ Effect can be surprisingly pervasive; for example suppose in `char_loc` we made first argument non-`const`

```
1 // bad design : first parameter should be const
  string&
2 void char_loc( string& s, char c,
  string::size_type & loc ,
3               string::size_type & count );
```

- ▶ Following call would fail at compile time:

```
1 char_loc( "Bad Design", 'D', aloc, an );
```

Effect of plain reference when not required

- ▶ Another side effect is, suppose we write another function in which we want to reuse `char_loc`:

```
1 // function to look for '.' at end of sentence
2 bool is_sentence( const string &s ){
3     string::size_type idx=0, n=0;
4     char_loc( s, '.', n, idx );
5     return ( idx == s.size() -1 && n=1 );
6 }
```

- ▶ Above will not compile since we had not declared first argument of `char_loc` as `const string &s`.

Array parameters

- ▶ We cannot pass array to function by copy
- ▶ When we use array it is usually converted to a pointer
- ▶ We are actually passing pointer to array's first element when we pass an array to a function
- ▶ The following overloaded function declarations all take a single const int array:

```
1 void printarray( const int * );
2 void printarray( const int [] );
3 void printarray( const int [10] );
4 int i = 10, j[4] = {0, 1, 2, 3};
5 printarray( &i ); // ok &i is int*
6 printarray( j ); // ok j is converted to int* to
   j[0]
```

- ▶ **Warning:** functions using array must stay within bounds of array size

Using marker to specify array extent

- ▶ Approach of C-style character arrays is to put a null character '\0' for the last character of the array.
- ▶ For example get length of c-style char array

```
1 unsigned len_carray( const char *pc ){
2     unsigned retval = 0;
3     // check if character array pointer
4     // is not a null pointer
5     if ( pc ){
6         // check if character is not null character
7         while ( *pc ){
8             ++retval;
9         }
10    }
11    return retval;
12 }
```

- ▶ Using marker is good when there is a marker value that can be used – may not work so well for int where every value is okay

Using standard library convention to specify array extent

- ▶ Second method to handle start and end of array is to pass pointers to first element and one past end of array
- ▶ Example, get sum of integers in array:

```
1 int sum_intarray( const int * beg, const int *
    end){
2     int retval=0;
3     while ( beg != end && beg != nullptr ){
4         // sum the current element then advance the
           pointer
5         retval += *beg++;
6     }
7     return retval;
8 }
```

- ▶ To use this function, pass two pointers:

```
1 int arr[10] = {2, 4, 6, 8, 10,
2               12, 14, 16, 18, 20 };
3 cout << "sum(i=1,10: 2*i)="
4     << sum_intarray( begin(arr), end(arr) )
5     << endl;
```

Explicitly passing size to specify array extent

- ▶ Third method to handle start and end of array is to define a second parameter that indicates the size of the array:

```
1 int sum_intarray( const int ci[], const size_t n){
2     int sum=0;
3     for ( size_t i = 0 ; i != n ; ++i ){
4         sum += ci[ i ];
5     }
6     return sum;
7 }
```

- ▶ An example use of this function:

```
1 int arr[10] = {2, 4, 6, 8, 10,
2               12, 14, 16, 18, 20 };
3 cout << "sum(i=1,10: 2*i)="
4     << sum_intarray( arr , end(arr) - begin(arr) )
5     << endl;
```

Array reference parameters

- ▶ We can define a parameter that is a reference to an array
- ▶ Same example, but using reference to array:

```
1 int sum_intarray( int (&ia) [10] ){
2     int sum = 0;
3     for ( auto i : ia ){
4         sum += i;
5     }
6     return sum;
7 }
```

- ▶ Parentheses around (&ia) are required
- ▶ Because size of an array is part of type, can use its dimensions in the function
- ▶ however only works for integer array with ten elements

Passing a multi-dimensional array

- ▶ No multi-dim array, but instead array of arrays
- ▶ Example, suppose we want to sum elements of array

```
1 int sum_2darray( const int (*matrix)[5], const
    size_t m ){
2     int sum = 0;
3     for ( size_t i = 0 ; i < m ; ++i ){
4         for ( size_t j = 0 ; j < 5 ; ++j ){
5             sum += matrix[i][j];
6         }
7     }
8     return sum;
9 }
```

- ▶ Need to be careful to pass dimensions correctly:

```
1 int arr [2][5] = { {2, 4, 6, 8, 10},
2                   {12, 14, 16, 18, 20} };
3 cout << "sum="
4     << sum_2darray( arr , 2 );
5     << endl;
```

main function command line options

- ▶ Can define `main` function with parameters to get command line options
- ▶ For example may run a program `example1` with options:
> `example1 -o out.txt -config conf.txt`
- ▶ Such options are passed as a number of arguments `argc` and array of pointers to c-style character arrays `argv`
- ▶ For example program that just prints number of arguments passed, and value of each argument:

printArgv.cpp

```
1 int main( int argc , char * argv[] ){
2     cout <<" number of arguments = " <<argc<<endl;
3     for ( size_t i = 0 ; i != argc ; ++i ){
4         cout << " argv[" << i << "] = "
5             << argv[i] << endl;
6     }
7     return 0;
8 }
```

- ▶ `argv[0]` is always the name of the program

Week5 Done!

- ▶ Homework 4 given out last week is due Oct. 6 by 17:00
- ▶ Homework 5 given out today is due Oct. 20 by 17:00