Week5 : Statements and Functions

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 5

Outline

Outline

Simple statements

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

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

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

```
  ; // null statement
```

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

```
while ( cin >> inum && inum > 0 )
  ; // 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:

```
unsigned grade = 0;
vector<string> letters= {"F", "D", "C", "B", "A",
    "A++"};
string lettergrade;
if ( cin >> grade ){
  if ( grade < 60 )
       lettergrade = letters[0];
  else
       lettergrade = letters [ (grade-50)/10 ];
}
```

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

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

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]; //
           fetch the letter grade
5      // Below is not inside the else statement!
6      // Instead, even failing grades could get a +
           or - added
7      if (grade != 100)  // add plus or minus only
           if not already an A++
8          if (grade % 10 > 7)
9              lettergrade += '+';  // grades ending
                   in 8 or 9 get a +
10         else if (grade % 10 < 3)
11             lettergrade += '-';  // grades ending
                   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`
- Indention 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

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

`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 `case`s 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:

```
unsigned Nvowels = 0;
unsigned Nother = 0;
char ch;
while ( cin >> ch ){
  switch ( ch ) {
    case 'a': case 'e': case 'i':
    case 'o': case 'u':
          ++Nvowels;
          break;
    default:
          ++Nother;
          break;
  }
}
```

- 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:

```
switch (ch){
  case 'a':
    int aVal = 0; // error: var init maybe bypassed
    string s;  // err: string implicit init
    int aaVal;  // ok: aaVal not initialized
    break;
  case 'b':
    aaVal = get_number(); //ok: aaVal is in scope
  case 'c':
    {
      int cVal = 0; // ok: inside statement block
      // ...
    }
}
```

- 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:

```
vector <int> v;
int i;
while ( cin >> i )
      v.push_back( i );
auto iter = v.begin();
while ( iter != v.end() && *iter >= 0 )
      ++iter;
if ( iter == v.end() ){
  // we know all elements in v are >= 0
}
```

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):

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

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

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

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

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

Range `for` loops in C++ 11

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

```
for ( declaration : expression )
      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 loo

Range `for` loop example

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

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

- 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:

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

`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:

```
do
      statement;
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 !=
             buf.end(); ++it){
7          if (*it == ' ')
8            break; // #1, leaves the for loop
9          // . . .
10       }
11       // break #1 transfers control here
12       // remaining '-' processing:
13       break; // #2, leaves the switch statement
14     case '+':  // . . .
15   } // end switch
16   // end of switch: break #2 transfers control here
17 } // end while
```

`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:

```
string s;
while ( cin >> s && !s.empty() ){
  if ( ! isupper( s[0] ) ){
    continue;
  }
  // do something with words that start
  // with uppercase letter...
}
```

`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
          short");
7   }
8   return a+b;
9 }
```

- 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:

```
try {
   program-statements
} catch ( exception-declaration1 ) {
   handler-statements1
} catch ( exception-declaration2 ) {
   handler-statements2
} //...
```

- `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
             short");
8      }
9      cout << a << " + " << b << " = " << a+b << endl;
10     cout << "Enter two numbers :" << endl;
11   } catch ( overflow_error err ) {
12     cout << err.what()
13          << "Try again? y or n: " << endl;
14     char ch;
15     if ( cin >> ch && ch == 'n' ){
16       break; // break out of while loop
17     }
18   } // end catch block
19 } // 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>:

`exception`	The most general kind of problem
`runtime_error`	Problem detected only at run time
`range_error`	Run-time eror for results outside range of meaningful values
`overflow_error`	Computation overflow
`underflow_error`	Computation underflow
`domain_error`	Error in logic of program
`invalid_argument`	Error in argument passed to function
`length_error`	Attempt to make object larger than max size
`out_of_range`	Used value outside the valid range

- All initialized with a c-style character string (`const char*`)
- Get back string using `what()` method

Outline

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 my have a return result
- Functions can be overloaded – meaning the same name may refer to several different functions.

Function basics

- function definition consists of:

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

- 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:

```
int i = 10; // initialize val from literal
double retval = 1.0; //
if ( i < 0 ){
   retval = NAN;
} else {
   for ( int j=2; j <= i ; j++ ){
      retval *= j;
   }
}
double fi = retval;
//...
```

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
  - **Livetime** 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:

```
size_t callcount(){
  static size_t n = 0;
  return ++n;
}
int main(){
  for ( size_t i = 0; i != 100; ++i ){
    cout << callcount() <<endl;
  }
  return 0;
}
```

- 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++0x` 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++0x -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

```
int num = 100;
int tot = num; // tot is a copy of value of num
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

```
int add5( int num ){
  num += 5;
  return num;
}
int main(){
  int val1 = 1;
  int val2 = add5( val1 );
  // val1 is still 1, val2 is 6
}
```

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

```
void char_loc( const string& s, char c,
               string::size_type & loc,
               string::size_type & count ){
  count = 0;
  loc = s.size();
  for ( size_t i = 0; i != s.size() ; ++i ){
    if ( s[i] == c ){
      if ( loc == s.size() ) loc = i;
      ++count;
    }
  }
}
```

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

```
string::size_type ns = 0, idx = 0;
string s = "garter snakes slither slowly";
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:

```
int sum_intarray( const int * beg, const int *
    end){
  int retval=0;
  while ( beg != end && beg != nullptr ){
    // sum the current element then advance the
        pointer
    retval += *beg++;
  }
  return retval;
}
```

- To use this function, pass two pointers:

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

```
int sum_intarray( const int ci[], const size_t n){
  int sum=0;
  for ( size_t i = 0 ; i != n ; ++i ){
    sum += ci[ i ];
  }
  return sum;
}
```

- An example use of this function:

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

- Midterm project topic is now posted – start working on the first project for this class!
- Note project due date is Oct. 25 (17:00).
- Due date is *firm* – will get marks back before course drop date of Nov. 1