COSC 151: Intro to Programming: C++

Chapter 5
Statements

Chapter 5: Statements

- **Objectives**
  - We should
    - Understand basic statements and compound statements
    - Have a better understanding of statement scope
    - Understand conditional statements (`if`/`else`/`switch`)
    - Understand iterative statements (`for`/`while`/`do;while`)
    - Understand jump statements (`break`/`continue`/`goto`)
    - Have an introduction to **exceptions**

C5: Simple Statements

- **Most statements end with a semicolon**
  - Examples:

```
ival += 5;
std::cout << name;
```

- **A statement can be empty – the null statement**

```
; // null statement
```

  - A null statement can be useful when the language requires a statement, but we have no useful work to do

C5: Null Statement and Bad Logic

- **The null statement (simply ;) can be used anywhere the language requires a statement, but we have no meaningful work to do**

```
int x = 0;
for(; x < 50; ++x);

cout << x << endl;  // prints 49...
```

- **The null statement can be a source of frustrating errors**

```
int x = 0;
while(x < 10); // null statement here, causes infinite loop!
{
   // So this code is never executed
   std::cout << x++ << '\n';
}
```

C5: Compound Statements (Blocks)

- **We can group statements (called blocks) by putting them in {}**
  - This is useful when the language requires a statement, but we have more work to do than can be done in a single statement

```
string name;
vector<string> names;
while(cin >> name)
{
    std::cout << “Hello, “ << name << endl;
    names.push_back(name);
}
```

C5: Statement Scope

- **Variables can be defined inside of a statement**
  - These variables exist only within the statement they are defined in

```
while(int i = get_num()) // i is created and initialized each iteration
{
   cout << i << endl;
}

i = 0; // ERROR, i is not accessible here
```

C5: Conditional Statements

- **There are two main conditional statements**
  - `if` determines flow based on a condition
  - `switch` evaluates an integral expression and chooses one of several execution paths based on the expression’s value

C5: if/else if/else Statement

```
if(condition)  // Simplest form, statement will be executed
    statement  // if condition is true

if(condition)  // Which statement is executed depends on the
    statement  // condition
else
    statement

if(condition)  // White statement is executed depends on one
    statement  // of (potentially many) conditions
else if(condition)
    statement
else
    statement
```

C5: `if`/`else if`/`else` Statement (contd)

- **`else` is optional**
  - For `if`
  - For `else if`
- **If an `else` is present, the statement under it will be executed if no other tested conditions were true**

C5: `if`/`else if`/`else` Statement (contd)

- **There are complicated rules for matching if with else.**
  - Don’t believe me, read the book
- **To ensure we always get the expected outcome, it’s a good idea to always use braces around the statements under conditions**

```
if(x % 2 == 0)
{
    cout << x << “ is even.” << endl;
}
```

C5: `switch` Statement

- **Switch statements allow for selecting among a possibly large number of fixed alternatives.**

```
switch(ch) // assume ch is of type char…
{
   case ‘a’:
      ++a_cnt;
      break;
   case ‘e’:
      ++e_cnt;
      break;
   case ‘i’:
      ++i_cnt;
      break;
   case ‘o’:
      ++o_cnt;
      break;
   case ‘u’:
      ++u_cnt;
      break;
}
```

C5: switch Statement (contd)

- **In a switch, execution starts at the first case label that matches and continues through until it reaches a break, or the end of the statement.**

```
switch(ch) // assume ch is of type char…
{
   case ‘a’:    // execution flows through all cases
   case ‘e’:    // giving us the count of all vowels
   case ‘i’:
   case ‘o’:
   case ‘u’:
      ++vowel_cnt;
      break;
}
```

C5: switch Statement (contd)

- **Switch statements can have an (optional) default, that is executed if no other case labels match**

```
switch(ch) // assume ch is of type char…
{
   case ‘a’:    // execution flows through all cases
   case ‘e’:    // giving us the count of all vowels
   case ‘i’:
   case ‘o’:
   case ‘u’:
      ++vowel_cnt;
      break;
   default:   // below executed only if this was not in a,e,i,o,u
      ++non_vowel_cnt;
      break;
}
```

C5: Iterative Statements

- **We're already familiar with some of the iterative statements**
  - `while`
  - `for`
  - Range-based `for`
- **There is one more we will learn about**
  - `do-while`

C5: `while` Statement

- **Useful when**
  - We want to iterate indefinitely
  - Need access to the loop control variable after the loop finishes

```
vector<int> v;
int i;

while(cin >> i) // read until “complete”
   v.push_back(i);

auto beg = v.begin();

// This loop continues until the end of the vector
// or until the current number in the vector is negative
while(beg != v.end() && *beg >= 0)
   ++beg;

if(beg == v.end())
{ // we know all the numbers are positive
}
```

C5: Traditional for Statement

- For loop allows us to initialize loop variables, check the loop condition, and perform a post loop expression, all in the header of the loop.

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

C5: Traditional for Statement

- **All parts of a for header are optional**

```
for(;;)  // legal, (but useless) infinite loop
{
}

for(int x = 0; ; ++x)
{
   if(x == 17)
      break;   // short circuit out if x == 17 (we will see break shortly)
}
```

C5: Range for Statement

- Range-based for loop allows us to iterate over an entire range of elements
- Should be the loop we chose by default, unless we *have to* use a different one

```
for(auto x : {1, 2, 3, 4, 5})
    if(x % 2 == 0)
        cout << x << endl;
```

C5: `do while` Statement

- **Like a while loop, but we execute the statement at least once, regardless of the condition**

```
do
    statement
while(condition);
```

C5: `do while` Statement

- **Useful when we want to guarantee we get a good value**
  - Validating input
  - Retry mechanism

```
const int max_retries = 5;
int retry_count = max_retries;
do
   if(send_message())
      break;
while(--retry_count);
```

C5: Jump Statements

- **Some statements allow us to modify the flow of execution**
  - **`break`** – terminates the nearest enclosing `while`, `do while`, `for` or `switch` statement
  - **`continue`** – terminates the *current iteration* of the nearest enclosing loop and i*mmediately begins the next iteration*
  - **`goto`** – provides an *unconditional jump* from the `goto` to another statement.
    - **DO NOT USE** – there are no use cases for goto that aren't better solved by other alternatives

C5: break Statement

- **Allows us to leave the nearest enclosing loop**

```
vector<string> vs;    // assume this is filled
uint64_t offset = -1; // that's a BIG number!

for(decltype(vs.size()) sz = 0; sz < vs.size(); ++vs)
{
   bool found = false;

   for(decltype(vs[sz].size()) idx = 0; idx < vs[sz].size(); ++idx)
   {
      if(vs[sz][idx] == 'X')
      {
         found = true;
         break;  // This terminates the inner loop
      }
   }

   if(found)
   {
      break;  // This terminates the outer loop
   }
}
```

C5: `continue` Statement

- **Allows to terminate the current iteration of the nearest enclosing loop**

```
vector<string> vs;    // assume this is filled

// these loops print out all the characters in the strings
// except for any punctuation present, one character per line

for(decltype(vs.size()) sz = 0; sz < vs.size(); ++vs)
{
  for(decltype(vs[sz].size()) idx = 0; idx < vs[sz].size(); ++idx)
   {
      if(ispunct(vs[sz][idx])
         continue; // stop this iteration, continue the NEXT one

      cout << vs[sz][idx] << endl;
   }
}
```

C5: `goto` Statement

- **DO NOT USE**
- **Seriously.**

C5: Exceptions

- **Exceptions are the language's defined manner to handle anomalous behavior**
- **They are somewhat complex, but provide a graceful means of handling errors**

C5: throw Expressions

- The first part of exception processing is to “throw an exception”
- Nearly any type can be thrown, but you should limit yourself to types specifically built to convey error information

```
struct bad_data {};  // a (too) simple user defined exception for “bad data”

if(!good_data())     // check if our data was “good”
   throw bad_data;   // it wasn’t! Throw a bad_data exception
```

C5: `try` Blocks

- If we want to catch an exception, we have to use a `try` block
- Inside the try block are our normal program statements.

```
try {
   gather_data_from_user();
   check_data_is_good();
   // if we get here, then everything was OK!
}
```

C5: Exception Handlers (catch Blocks)

- **To catch (handle) exceptions we need to have catch blocks**
  - A try block must have at least one exception handler
- **If any function throws an exception whose type is "caught" in an exception handler, flow will "unwind" to that point, and start inside the exception handler**

```
try {
   gather_data_from_user();
   check_data_is_good();
   // if we get here, then everything was OK!
}
catch(const bad_data&) {
   cout << "Error: User entered data was incorrect." << endl;
}
```

C5: Standard Exception Classes

exception	The most general kind of problem...
runtime_error	Problem that can be detected only at run time
range_error	A runtime error: result was outside the range of valid values
overflow_error	A runtime error: computation that overflowed
underflow_error	A runtime error: computation that underflowed
logic_error	Error in the logic of a program
domain_error	A logic error: argument for which no result exists
invalid_argument	A logic error: argument as invalid
length_error	A logic error: input too long
out_of_range	A logic error: used a value outside of valid range

Final Thoughts

- Watch for the null statement in your programs, it’s a common source of bugs
- Putting statements into blocks will force the compiler to take your desired action
- `if`/`else if`/`else` can be used to pick flow between some conditions
- `switch` can too, but it is limited to integral conditions

Final Thouhgts (contd)

- Make sure you use `break` appropriately in your `switch` statements, also a common source of bugs
- Be familiar with all types of loops, `while`, traditional `for`, range-based `for`, `do-while`, use the one that is appropriate for the job at hand

Final Thoughts (contd)

- **Jump statements disrupt the normal flow of operation, so use them only when necessary**
  - There are many cases where they are not necessary!
- **Don't use `goto`. Seriously.**
- **Exceptions are how we are expected to deal with *exceptional circumstances***
  - Start to think about how you could use them in your assignments!