Part III

Further C++ constructs

Outline

8 Block and Scope

9 Switch-case

10 Iteration

11 Operator Precedence

12 More file-handling

Blocks and Scope

- A *block* is a set of statements surrounded by {}.
- These can be placed anywhere that a single statement can.
- A definition of a variable (or other object) extends from its point of declaration to the end of the block in which it has been defined.

Block example

```
if( a == 3 )
  a = 4;
  std::cout << "a is now 4" << std::endl;
```

will compile correctly, but will always print the message. The `if` statement only applies to the `a=4`.

The programmer probably meant to write:

```
if( a == 3 ){
  a = 4;
  std::cout << "a is now 4" << std::endl;
}
```

Even if your `if` statement applies to only a single statement, it is good practice *always* to use braces.

Scope example

In the following, the scope of `b` is inside the braces only

```
int a = 2;
if( a == 2 ){
  int b = 4;
  a += b; // OK – b is in scope
}
a += b; // Compiler error – b is not known here
```

- Scope also applies to functions, classes, and other constructs.
- Scope is also important if you have multiple variables with the same name
- C++ does define which variable is referred to, but you shouldn't reuse variable names in the first place
- You are more likely to get confused than the compiler.
- Also, the storage for variables is freed once they go out of scope, so there is no way of recovering their data.

Outline

8 Block and Scope

9 Switch-case

10 Iteration

11 Operator Precedence

12 More file-handling

Switch-case

When selecting from a finite list of options:

```
int a;
std::cin >> a;
switch(a){
  case 0:
    std::cout << "a = 0" << std::endl;
    break;
  case 1:
    std::cout << "a = 1" << std::endl;
    break;
  default:
    std::cout << "Neither 0 nor 1" << std::endl;
}
```

- Avoids chains of `if else`
- Can only switch on integral types (`int`, `char`, and similar)
- The `break`s cause execution to jump to after the `switch` block.

Break-usage

- If **break** is not used, execution falls through to the next statement.

```
switch(a){
  case 0:
  case 1:
  std::cout << "a is 0 or 1" << std::endl;
  break;
  case 2:
  std::cout << "a is 2" << std::endl;
  case 3:
  std::cout << "a is 2 or 3" << std::endl;
  break;
  case 4:
  case 5:
  case 6:
  std::cout << "a is larger than 3" << std::endl;
}
```

Variables in switch-case

- Note that it is not permitted to declare variables directly inside a `switch case` block:

  ```
  switch(a){
  case 0:
    int b = x*y; // Not valid
    break;
  };
  ```

  Either contain the new variable and related statements in an enclosing set of braces, or declare it before the `switch case`:

  ```
  switch(a){
  case 0:
    {
      int b = x*y; // Valid
    }
    break;
  };
  ```

Outline

8 Block and Scope

9 Switch-case

10 Iteration

11 Operator Precedence

12 More file-handling

Looping

- We may want to repeat a series of instructions multiple times, possibly for a sequence of values of a variable
- A loop is a set of instructions that are carried out multiple times (anywhere from zero to infinity)
- Number of iterations is (probably) only known at run-time.
- Number of iterations may not be known even as the loop starts.

For loop

A “for” loop is usually used where the number of iterations is known at the start of the loop:

```
For all values of i from 1 to 10:
  Calculate i'th triangular number
  Print i'th triangular number
End Loop
```

In C++ the specification of a for-loop is:

```
for( initialization ; condition ;
     per-iteration-update )
{
  // Code to loop over
}
```

For loop

The simplest example is:

```
for( int i=0 ; i < 10 ; i++ ){
  std::cout << "Iteration " << i << std::endl;
}
```

which will print:

```
Iteration 0
Iteration 1
...
Iteration 9
```

- The initialization `i=0` is carried out once only.
- The condition `i < 10` is checked at the beginning of each iteration over the contained code.
- The update `i++` is carried out at the end of each iteration.

More for loops

The following code:

```
int j=0;
int N=10;
for(int i=1 ; i <= N ; i++){
  j += i;
  std::cout << "Triangular number " << i
            << " is " << j << std::endl;
}
```

will print the triangular numbers up to 55.

- We initialize `j` to be zero
- At every iteration, `j` is increased by `i` and printed
- The loop stops when `i == 11`;
  the instruction block is not evaluated in this case.

For extended

The previous example could also be written:

```
for( int i=1, j=1 ; i <= 10 ; i++, j += i ){
  std::cout << "Triangular number " << i
            << " is " << j << std::endl;
}
```

- Here we see the comma operator, which allows multiple statements to be put together.
- It is only really used within the `for` loop, where a semi-colon is already used to separate the parts of the loop definition.
- However, you should not put too much into the `for()` statement.
- The above example is *not* a good example of a `for` loop.
- It is far less easy to read than the preceeding example and is not as obviously correct.
- Complex for-loop syntax can also stop OpenMP from working efficiently (or at all)

Infinite loops

- It is not necessary to have all of the components of the `for` specified.
- For example:

  ```
  for( ; ; ){
  }
  ```

  is valid, and corresponds to an infinite loop.
- Any of the components can be missing in any combination.

Comma operator

- Strictly speaking, the comma operator returns its right-hand argument
- It has the lowest precedence of all operators
- Therefore, it could be used to string statements together:

  ```
  int i=7, i++, i++;
  ```

  will result in `i==9`.
- However, the comma operator is very rarely used (outside of `for` loops), and any other use of it should be regarded as suspect.

Range-based for-loops

- An alternative form of the for-loop is a range-based for-loop:

```
for( int i : {0, 1, 4, 9, 16, 25} ){
  std::cout << "i = " << i << std::endl;
}
```

- Or, perhaps more usefully:

```
std::vector<int> myValues;
// Fill in elements of myValues ...
// Double all elements of myValues
for( int& i : myValues ){
  i = i * 2;
}
for( int i : myValues ){
  std::cout << "i = " << i << std::endl;
}
```

- The latter form will become more useful later.

While loop

If you want to repeat a calculation as long as a particular condition is satisfied, use a while loop:

```
while(condition){
 // Code to perform
}
```

- The condition is checked as the computer enters the loop, and after each evaluation of the loop.
- If the condition is false at this point, then jump to point directly after the loop.
- Note that the loop is not exited as soon as the condition is false, only when execution reaches the end of the loop (and the condition is still false).

While example

```
bool found = false;
int i = 0;
while( ! found ){
  if( isWantedObject(myObject[i]) ){
    found = true;
    std::cout << "I've found it!" << std::endl;
  }
  i++;
}
```

Just after the end of the loop, we know that `found` is true.

There are many other uses of `while` loops.

Do-While loop

Very similar to a plain `while` loop:

```
do{
 // Code to perform
}while(condition)
```

- The condition is checked *after* each iteration.
- So, the code in the loop is guaranteed to execute at least once.
- If the condition is false at this point, then execution jumps to the point directly after the loop.

Do-While example

```
bool found = false;
do{
  // Code to locate missing object
  // Evaluated at least once
}while(!found);
```

At the end of the loop, we know that **`found is true`**

Do/While/For equivalence

- With a little thought, any do-while/while/for loop can be written as any of these types
- The only reasons for the existence of all three are:
    - Historical (older languages had them)
    - Readability (Different forms are usually used for different purposes)
- Roughly, they are used as:
    - For: When number of iterations is known on entry to the loop
    - Do-while: When a condition is repeatedly checked throughout the loop
    - While: As before, but when a condition may be known before the loop starts and the loop may not need to be evaluated at all.

Getting out of loops

In some cases, we may want to get out of a loop early:

```
for(int i=0 ; i < 10 ; i++){
  double x = pow(y, i);
  if( x > 1e10 ){
    break; // Result too large - don't print any more
  }
  std::cout << y << "^" << i << " = " << x << std::endl;
}
```

`break` causes execution to immediately jump to directly after the loop.
It jumps out of any current `for`/`while`/`do` loop.

Continuing execution

In some cases, we may want to skip the rest of a loop

```
while(!endOfFile){
  char c = getNextChar();
  if( c == ’\n’ ){ // New−line − nothing to do
    continue;
  }
  // Do main processing work
}
```

- `continue` causes execution to jump to just before the end of the loop
- The loop-condition is checked directly after `continue`, before execution resumes at the loop-head.
- This applies to `for`/`while`/`do`

Goto and labels

- Unfortunately, `goto` appears in C++ and can be used as follows:

```
goto myLabel;
// Some code here
myLabel:
// More code goes here
```

- However, it must not be used to cause execution to jump across past initializations of variables, or in/out of functions
- Its use in practice should be regarded with extreme suspicion, unless there is a very good reason why `break`/`continue`/`if` could not be used.
- Its use tends to make the execution path hard to follow when debugging or trying to understand code, although it can be useful.
- See “Goto considered harmful” (Dijkstra, 1968), but also “ “Goto considered harmful” considered harmful” (CACM, March 1987) and “ ‘ “Goto considered harmful” considered harmful’ considered harmful?” (Comm. of the ACM, 1987).

Outline

8 Block and Scope

9 Switch-case

10 Iteration

11 Operator Precedence

12 More file-handling

Operator Precedence (Partial)

Precedence	Op	Description	Associativity
2	++ -- ()	Suffix inc/dec Function call	left-to-right
3	++ -- + - ! ~	Prefix inc/dec Unary plus/minus Logical NOT and Bitwise NOT	right-to-left
5	* / %	Multiplication, division, modulus	left-to-right
6	+ -	Addition/subtraction	left-to-right
7	<< >>	Left/right shift	left-to-right
8	< <= > >=	Less-than (or equal) Greater-than (or equal)	left-to-right
9	== !=	(Non-)equality test	left-to-right
13	&&	Logical AND	left-to-right
14	||	Logical OR	left-to-right
15	?:	Ternary Conditional	right-to-left
16	= += -= *= /= %=	Assignment Assignment and add/subtract Assignment and mult/div/mod	right-to-left

Operator precedence examples

Some examples of operator precedence:

```
double x = 2.0 * 4.5 + 5.2; // Evaluates to 14.2
double x = 2.0 * (4.5 + 5.2); // Evaluates to 19.4
double a = 3.0 / 1.5 * 2.0; // Evaluates to 4
int b = 9 / 2 % 3; // Evaluates to 1
int c = 1 << 2 * 3; // Evaluates to 64
int c = (1 << 2) * 3; // Evaluates to 12
```

Parentheses control the evaluation order of operators.

Use whenever they are required, or when it improves clarity.

Outline

8 Block and Scope

9 Switch-case

10 Iteration

11 Operator Precedence

12 More file-handling

File-handling

- So far we have seen how to output to the terminal
- To output to a file, we create a stream which goes into/comes from a named file

```
#include <fstream>
std::ofstream outFile("/home/pmb39/MyFile.txt");
outFile << "Hello. I am in a file";
outFile << "5 * 10 = " << 5*10 << std::endl;
outFile.close();
std::ifstream inFile("/home/pmb39/dataFile");
int a;
inFile >> a; // read integer value from file
inFile.close()
```

- Note the similarity of the code to outputting to the terminal
- The differences between terminal and file have been abstracted away
- Both are effectively places to which a stream of characters can be sent

I/O modes

- There are various open-modes for a file:
  - `std::io_base::in` - open for input
  - `std::io_base::out` - open for output
  - `std::io_base::trunc` - truncate existing file when opening
  - `std::io_base::ate` - seek to end after opening
  - `std::io_base::app` - append to file (seek to end before each write - includes intervening writes by potential other processes)
- Opening a file for input is therefore:

```
std::ifstream inFile("MyFile.txt", std::io_base::in);
```

- To close a file, use

```
inFile.close()
```

File-errors

- In order to detect bad stream-states, the following tests can be used, all returning `bool`s:

```
myFile.eof(); // End of file seen
myFile.fail(); // Next operation will fail
myFile.bad(); // Stream is corrupted
myFile.good(); // None of the above hold
```

- Therefore:

```
while(!myFile.eof()){
  myFile >> i;
}
```

will read successive values into `i` until the end of the file is reached.

File-error examples

```cpp
int a;
std::cout << "Enter a: ";
std::cin >> a;
std::cout << "a = " << a << std::endl;

std::cout << "Fail = " << std::cin.fail() << std::endl;
std::cout << "Good = " << std::cin.good() << std::endl;
std::cout << "Bad = " << std::cin.bad() << std::endl;
```

```
Enter a: 1                 Enter a: x
a = 1                      a = 0
Fail = 0                   Fail = 1
Good = 1                   Good = 0
Bad = 0                    Bad = 0
```

More stream operations

- It is possible to perform low-level operations on streams:

```
char c;
myFile.get(c); // Get a single character
char line[BUFFER_SIZE];
 // Read a whole line into a char−array
 myFile.getline(line, BUFFER_SIZE);
```

- These should only be used when reading custom formats/files
- It is possible to create strings as if they were streams:

```
std::ostringstream myMsg;
std::string name = "Dave";
myMsg << "Hello " << name;
std::string msg = myMsg.str(); // msg contains "Hello Dave"
```