C++ References

Tom Latham

THE UNIVERSITY OF WARWICK

Function arguments

- We have seen this morning that functions receive *copies* of the arguments passed to them
  - This is referred to as ‘*passing by value*’
  - For many situations this is OK
- However, consider the following two scenarios where this causes problems:
  - The function cannot change the value of the argument in such a way that it is also changed in the calling scope
  - If the object being passed to the function is complicated or large, e.g. a big `std::vector` or a long `std::string` there can be considerable overhead from making the copy, both in terms of time and memory
- In these cases, it would be better if a function could act on the original object

References

- References are essentially a means of creating a new variable for an existing object
  - Here the new variables are ‘a’ and ‘b’
  - When the swap function is called they are assigned the objects currently referred to by the variables ‘x’ and ‘y’
- The ampersand symbol (&) declares that ‘a’ and ‘b’ are references
- This then allows functions to act on the *actual objects passed* to them rather than copies of them. This is referred to as *‘passing by reference’*.

```
#include <iostream>


void swap( double& a, double& b )
{
   double tmp {b};
   b = a;
   a = tmp;
}


int main()
{
   double x {42.3};
   double y {11.2};

   std::cout << x << "\t" << y << "\n";

   swap(x,y);

   std::cout << x << "\t" << y << "\n";

}
```

References

- References are essentially a means of creating a new variable for an existing object
  - Here the new variables are ‘a’ and ‘b’
  - When the swap function is called they are assigned the objects currently referred to by the variables ‘x’ and ‘y’
- The ampersand symbol (&) declares that ‘a’ and ‘b’ are references
- This then allows functions to act on the *actual objects passed* to them rather than copies of them. This is referred to as *‘passing by reference’*.

```
#include <iostream>

void swap( double& a, double& b )
{
   double tmp {b};
   b = a;
   a = tmp;
}

int main()
{
   double x {42.3};
   double y {11.2};

   std::cout << x << "\t" << y << "\n";

   swap(x,y);

   std::cout << x << "\t" << y << "\n";

}
```

References

- References are essentially a means of creating a new variable for an existing object
    - Here the new variables are ‘a’ and ‘b’
    - When the swap function is called they are assigned the objects currently referred to by the variables ‘x’ and ‘y’
- The ampersand symbol (&) declares that ‘a’ and ‘b’ are references
- This then allows functions to act on the *actual objects passed* to them rather than copies of them. This is referred to as *‘passing by reference’*.

```
#include <iostream>

void swap( double& a, double& b )
{
   double tmp {b};
   b = a;
   a = tmp;
}


int main()
{
   double x {42.3};
   double y {11.2};

   std::cout << x << "\t" << y << "\n";
   swap(x,y);
   std::cout << x << "\t" << y << "\n";

}
```

Const references

- But what if you do not want to change the object being passed but you still want to use pass by reference because it is large?

- Thankfully there is a way to ensure (compiler enforced) that the argument is not changed, by using a reference to a const object, often termed a 'const reference'

```cpp
#include <iostream>

void swap( double& a, double& b )
{
  double tmp {b};
  b = a;
  a = tmp;
}

void print( const double& a,
            const double& b )
{
  std::cout << a << "\t" << b << "\n";
}

int main()
{
  double x {42.3};
  double y {11.2};

  print(x,y);
  swap(x,y);
  print (x,y);
}
```

Rule of thumb for function arguments

- If the argument is a built-in type you can use pass by value
  - You'll most likely want to use const here – it is surprisingly rare that you actually want to change the value of the arguments and it is good to enforce the lack of change to avoid silly mistakes

```
double square( const double x );
```

- For everything else use const references

```
void printString( const std::string& s );
```

- Unless you need to change the object, in which case drop the const

```
void toLowerCase( std::string& s );
```

Returning references

- Should large objects also be returned by reference?
- This is a bit trickier since you have to think about the lifetime of the object being returned
- If it is local to the returning function, the answer is definitely 'No!'
    - It will be destroyed as soon as the function returns, so you'll be returning a reference to something that no longer exists!
- At the moment this is the only kind of object lifetime you've seen. We will see other cases in future weeks that mean that return by reference is viable and even desirable. But until then, don't do it. And even then, you always need to think carefully about it!

“Returning” via reference arguments

- Reference arguments can be used as a way of “returning” more than one object or of “returning” a very large object

- Instead of receiving the return of the function, a “blank” object (i.e. constructed with some default value(s)) is first constructed and is then immediately provided as a reference argument to the function

- When control returns to the calling scope, the state of the object passed could have been modified

Exercise on function arguments

- You can now also package the parsing of the command-line arguments into a function, which you should call "`processCommandLine`"

  - Let's think about and discuss what this needs to do…
  - Then we can decide how many and what types of arguments the function needs
  - Finally, we need to bear in mind the rule-of-thumb to choose the best forms (value/reference, const/non-const) for each of those arguments

Exercise on function arguments

- You can now also package the parsing of the command-line arguments into a function, which you should call "`processCommandLine`"

```
bool processCommandLine(
    const std::vector<std::string>& args,
    bool& helpRequested,
    bool& versionRequested,
    std::string& inputFileName,
    std::string& outputFileName )
{
    ...
}
```

Ranged-based for loops

- Now we can use references, we can look at a new way of looping over a container that was introduced in C++11: the range-based for loop
- Useful when one needs to simply use or operate on each element in turn
    - If you want to modify the element, just drop the `const` – see the second example
- We couldn't have used it in processing the command line arguments, for example, since we sometimes need to know where we are in the container and even operate on two elements at once

```
std::vector<int> my_ints = {1, 2, 3};

for ( const int& element : my_ints ) {
       std::cout << element << "\n";
}

for ( int& element : my_ints ) {
       ++element;
}

for ( const int& element : my_ints ) {
       std::cout << element << "\n";
}
```

The 'auto' keyword

- We can also see the first use case for the new 'auto' keyword
- The compiler is able to deduce the type of the element based on the type held in the container
- The 'auto' keyword allows us to take advantage of this and so avoids too much retyping when reusing or modifying code
  - We'll meet other places where it comes in very useful
  - Important, however, not to overuse it since it can make code harder to understand if used too much or in the wrong places

```
std::vector<int> my_ints = {1, 2, 3};

std::string my_str {"Hello world!"};

for ( const auto& elem : my_ints ) {
        std::cout << elem << "\n";
}

for ( const auto& elem : my_str ) {
        std::cout << elem << "\n";
}
```