Object-Oriented Programming in C++

Pre-Lecture 2:

Prof Niels Walet (Niels.Walet@manchester.ac.uk)

Room 7.07, Schuster Building

January 5, 2020

Prelecture 2

Outline

In this pre-lecture we cover a few more basic aspects of C++'s functionality, specifically

- Overloading functions
  - Many functions, one name
- The use of pointers and references
  - How to pass information efficiently
- Dynamically allocating (and freeing) memory
  - What to do if you suddenly need some storage
- Using filestreams
  - read from and write to files on your hard disk

Overloading functions

Overloading functions

- Overloading a function is often useful when we want to code functions to perform the same task with more than one data type, or having slightly different (but closely related) functionality linked to the data type.
- Common Naive Solution: Write a separate function for each type
- Problem: usually each function must be called something different
- C++ removes this barrier; functions can be overloaded, as long as arguments and/or return differ.
- Later will show an even more efficient technique for a class of these using templates

Overloading functions

Example: naive way

```
4  #include<iostream>
5  double max_double(double double1, double double2)
6  {
7    if(double1>double2) return double1; else return double2;
8  }
9  int max_int(int int1, int int2)
10 {
11   if(int1>int2) return int1; else return int2;
12 }
13 int main()
14 {
15   int integer1{1};
16   int integer2{2};
17   double real1{4.};
18   double real2{3.};
19   std::cout<<max_int(integer1,integer2)<<std::endl
20        <<max_double(real1,real2)<<std::endl;
21   return 0;
22 }
```

Listing 1 : selection of PL2/max1.cpp

(Compact inconsistent layout for size!)

Overloading functions

Example: C++ way

```
4  #include<iostream>
5  double max(double number1, double number2)
6  {
7    if(number1>number2) return number1; else return number2;
8  }
9  int max(int number1, int number2)
10 {
11   if(number1>number2) return number1; else return number2;
12 }
13 int main()
14 {
15   int integer1{1};
16   int integer2{2};
17   double real1{4.};
18   double real2{3.};
19   std::cout<<max(integer1,integer2)<<std::endl
20        <<max(real1,real2)<<std::endl;
21   return 0;
22 }
```

Listing 2 : selection of PL2/max2.cpp

- Compiler checks argument type to decide which version to use
- Substantial further simplifications can be made using templates.

Pointers and References

Pointers and References

- The above functions contain two parameters (`number1` and `number2`)
- Arguments were passed by value: local copies were made inside functions
- Sometimes it is better to pass by reference (i.e. pass starting address of data so function has direct access)
- Two main reasons for passing by reference:
  - You want to pass a complex argument (object) to the function (and don't want to spend all the time on copying!)
  - You want to modify the variables being passed inside the function
- An alternative is to do this using pointers. A pointer is a variable that holds the location (address) of a variable, which is used as follows:

```
float *x; // x is declared as a pointer to a float
float y{2}; // y is an ordinary float, set to 2
x=&y; // x now points to y (so x contains address of y)
cout<<*x<<endl; // de-reference x (prints out 2)
```

Pointers and References

Passing an array

An array is really a pointer, set up to point to each element, as can be seen from the code

```
1  // PL1/by_reference2.cpp
2  // Arrays vs. pointers
3  // Niels Walet, last updated 04/12/2019
4  #include<iostream>
5  void print_array(const int array_entries, int *array)
6  {
7    // show that array[i] is equivalent to *(array+i)
8    for(int i{};i<array_entries;i++)
9      std::cout<<array[i]<<" "<<*(array+i)<<std::endl;
10   return;
11 }
12 int main()
13 {
14   const int array_entries{3};
15   int array[array_entries];
16   for(int i{};i<array_entries;i++) array[i]=i;
17   print_array( array_entries,array);
18   return 0;
19 }
```

Listing 3 : PL2/array.cpp

which outputs

```
0 0
1 1
2 2
```

Pointers and References

Using a reference

- When you want to pass a single object[1] by reference, it is often easier to use a reference
- Simplifies syntax
- Less room for errors
- Detailed comparison:
  - passing by value,
  - using a pointer and
  - passing by reference

[1] Will become clearer later, for now read object as single int, double etc.

Pointers and References

Example using a pointer

First pass by value:

```
4  #include<iostream>
5  void double_value(double value)
6  {
7    value*=2;
8  }
9  int main()
10 {
11   double number{4.};
12   std::cout<<number<<std::endl;
13   double_value(number);
14   std::cout<<number<<std::endl;
15   return 0;
16 }
```

Listing 4 : selection of PL2/byvalue.cpp

which outputs

```
4
4
```

Pointers and References

Example using a pointer

Using a pointer

```
4  #include<iostream>
5  void double_value(double *ptr_double)
6  {
7    (*ptr_double)*=2;
8  }
9  int main()
10 {
11   double number{4.};
12   std::cout<<number<<std::endl;
13   double_value(&number);
14   std::cout<<number<<std::endl;
15   return 0;
16 }
```

Listing 5 : selection of PL2/byreference.cpp

which outputs

```
4
8
```

Pointers and References

Example using a pointer

Problems with the use of a pointer

- Pointers can sometimes get confusing
- The function was defined to take a pointer (to a double) as its sole argument

```
void double_value(double *ptr_double)
```

- Since the pointer stores an address of a double, the address of number had to be taken

```
  double_value(&number);
```

- The pointer then had to be de-referenced in the function

```
  (*ptr_double)*=2;
```

- There is a better way!

Pointers and References

Example using a reference (C++-style)

by reference:

```
4  #include<iostream>
5  using namespace std;
6  void double_value(double& reference)
7  {
8    reference*=2;
9  }
10 int main()
11 {
12   double number{4.};
13   std::cout<<number<<std::endl;
14   double_value(number);
15   std::cout<<number<<std::endl;
16   return 0;
17 }
```

Listing 6 : selection of PL2/byreference2.cpp

which outputs

```
4
8
```

Pointers and References

Example using a reference (C++-style)

- To pass by reference, we **only** modify the argument in the function that was called by value by inserting an ampersand (&),

```
void double_value(double& reference)
```

- No pointers (and de-referencing) required anymore
- Code identical as using by value!
- Also safer – the compiler can check that something valid is being referenced. That can not be done for a pointer! (A null-pointer is quite legal, or even a pointer to arbitrary memory...)

Pointers and References

Example using a reference

- Can also declare reference variables within a piece of code (but must be assigned)

```
5  int main()
6  {
7    int number{4};
8    int &reference{number};
9    std::cout<<number<<" "<<reference<<std::endl;
10   number++;
11   std::cout<<number<<" "<<reference<<std::endl;
12   return 0;
13 }
```

Listing 7 : selection of PL2/byreference3.cpp

produces

```
4 4
5 5
```

- Note that references must refer to something! `int &c; // invalid as c not initialized (must refer to a valid object)`
- References must also be of the same type as the objects they refer to
  `int a;float &d(a); // invalid as a is of type int`

Dynamic Memory Allocation

Dynamic Memory Allocation

- Often we do not know how much memory we require in advance.
- Simple example: data of unknown size to be read in from file,
- So size of storage is determined dynamically.
- Solution: use dynamic memory allocation
- Let me show you the way!

Dynamic Memory Allocation: example

- In C++, dynamic memory allocation is handled using two functions: `new` and `delete`
- To allocate memory, create a pointer (as before) and then use the `new` command

```
double *my_data;  // declare mydata as a pointer to a double
my_data = new double[N]; // mydata now points to new array of N doubles
```

- Advantage: no need to work out the size of the array in bytes.
- This can be done more concisely (declare variables when we need them!)

```
    double *my_data{new double[N]};
```

Dynamic Memory Allocation: Syntax

- General syntax for allocating memory for a single variable

```
// Allocate memory for variable of type Data_Type_1
// Memory is accessed with pointer myscalar
data_type_1 *my_scalar{new data_type_1};
```

- And for an array

```
// Allocate memory for array of type DataType2
// Memory is accessed with pointer myarray
data_type_2 *my_array{new data_type_2[N]};
```

- Memory is freed as follows

```
delete my_scalar; // single variable
delete[] my_array; // array
```

- Always free up memory when you have finished with it! (Known as garbage collection)
- Dynamically allocated memory goes on the heap - could run out during program if not freed (memory leaks) - vital for mission critical applications

File Streams

File Streams

- We already met the standard I/O stream commands, `cin` and `cout`
- To access these, we needed to use `#include<iostream>`
- The devices connected to these streams are unambiguous (keyboard and screen) and these streams are opened and closed for you.
- But often we want to read from and/or write to a file on disk.
- C++ also provides streams to connect to files.
- But we now need to define the device (file) connected to our stream, the stream type and we need to open and close them ourselves.

File Streams

Opening and closing

- To use file streams we need to add `#include<fstream>`
- We then need to define our type of file stream

```
ifstream my_input_file; // a file stream for reading only
ofstream my_output_file; // a file stream for writing only
fstream my_file; // a file stream for reading or writing
```

- To open our file we can use
  `my_file.open("data1"); // open file called data1`
- Or we can define and open our file stream at once
  `fstream my_file{"data1"}; // Open and attach data1 to my_file`
- To close our file, after we are done with it, use `my_file.close();`

File Streams

Reading and writing

- Syntax is the same as for the standard I/O streams
- E.g. to read a double from a file

```
double my_data;
myfile>>my_data; // extract a double from the file stream myfile
```

- To write out an int to a file

```
int my_code{32762};
myfile<<my_code; // insert mycode into file stream myfile
```

File Streams

Useful checks

- It is useful to check if our file opened succesfully

```
fstream my_file("data1"); // Check file opened successfully
if(!my_file.good()) {
    // Print error message and exit
    cerr<<"Error: file could not be opened"<<endl;
    return(1);
}
```

Note: `cerr` is the standard error stream (usually the screen)

- Can also check the data is read or written successfully

```
myfile>>x;
if(myfile.fail()) {
    cerr<<"Error: could not read data from file"<<endl;
    return(1);
}
```

File Streams

Useful checks

- If such rogue data (e.g. character) detected can skip with commands

```
my_file.clear(); // take stream out of fail state
my_file.ignore(); // ignore one unwanted character
```

- Might need to apply above more than once
- We can also check for the end-of-file (or not!), e.g.

```
while(!my_file.eof())  {
    // Read in some more data ...
}
// Must have reached the end so close file
my_file.close();
```

Summary

We looked at

- Overloading functions (same name, different–type–arguments or return type)
- How to use of pointers and references
- The mechanism for dynamically allocating (and freeing) memory
- Using filestreams to read from a file on your hard disk