Tute 11: Optimisations & Exceptions

Review of Dijkstra's Algorithm

I have provided the description of this algorithm as described in the reference book on mazes, "Mazes for programmers" by Jamis Buck.

https://drive.google.com/a/rmit.edu.au/file/d/0BzM8LtxnJauTV0ZBeWZTU2tSYUk/view?usp=sharing

You can get the whole book here: https://pragprog.com/book/jbmaze/mazes-for-programmers

Review: Exceptions, Lambdas & Profiling

What is a lambda function?

Note: this is a modern C++ feature.

A lambda function is an anonymous function, which means it has a definition but no declaration. C++ also supports closures, which means the lambda function can be stored in a variable for future use. They are often passed as arguments to other named functions or classes, or used in place when a function is required for simple/repetitive work without the need for writing a separate function declaration. Lambda functions have four parts:

```
[capture list](parameters) -> return type { function body }
```

eg:

```
[&z](int x, int y) -> int { return x + y; }
```

The capture list can be used in six different ways:

```
[]      //no variables defined. Attempting to use any external variables in the lambda is an error.
[x, &y] //x is captured by value, y is captured by reference
[&]     //any external variable is implicitly captured by reference if used
[=]     //any external variable is implicitly captured by value if used
[&, x]  //x is explicitly captured by value. Other variables will be captured by reference
[=, &z] //z is explicitly captured by reference. Other variables will be captured by value
```

What is an exception?

An exception is an event that occurs during program execution that disrupts the normal flow of the program's instructions by unwinding stack frames until encountering a stack frame that can handle the exception correctly. They are triggered using the keyword throw, and in C++ no special exception object needs to be used, as anything can be thrown, at any time. In order to handle an exception, any triggering code must be placed in a try block, and any handling code must be place in a catch block, eg:

```
void foo()
{
	throw std::string("Hello World");
}

int main(int argc, char **argv)
{
	try
	{
		foo();
	}
	catch (std::string &s)
	{
		std::cout << s << "\n";
	}

	return 0;
}
```

Note that the catch block cannot use the keyword auto and therefore must exactly match the type of exception thrown. If multiple exceptions can be thrown then multiple catch statements should be used. In the case where you do not care what is thrown and do not want to use it, the … statement should be used:

```
try
{
	foo();
}
catch (...)
{
	std::cout << "Caught\n";
}
```

C++ has no finally statement unlike Java, however this can be used as a substitute.

STL does provide default exception classes for easy use, which can be extended for custom exception classes (although this is not required):

```
#include <stdexcept>
#include <iostream>

void foo()
{
	throw std::runtime_error(“Hello World”);
}

int main(int argc, char **argv)
{
	try
	{
		foo();
	}
	catch (std::runtime_error &e)
	{
		std::cout << e.what() << “\n”;
	}
	return 0;
}
```

Note that exceptions should never be thrown inside destructors, as exceptions are used to unwind the stack and destructors are called during stack unwinding. This can be problematic if an exception is thrown during the stack unwinding of another exception and C++ guarantees that the program will terminate when this happens.

What is profiling?

Profiling is a form of dynamic program analysis (dynamic meaning at runtime) that is used to measure memory usage and/or performance of different parts of a program. It can be used to identify bottlenecks (ie, which functions take the most execution time) so that efforts to optimise can be focussed in those areas as optimisations in non-bottleneck areas will lead to smaller performance gains.

Programming Concepts: Lambdas & Profiling

How can a lambda function be passed to a function/method?

The function template can be used:

```
void foo(std::function<void()> t)
{
	t();
}

foo([]() { std::cout << “Lambda\n”; });
```

Or if the return type/arguments are unknown, templates can be used instead:

```
template <typename T>
void foo(T t)
{
	t();
}

foo([]() { std::cout << “Lambda\n”; });
```

Similarly, the easiest way to store a lambda function is using the auto keyword:

```
auto func = []() { std::cout << “Lambda\n”; };
foo(func);
```

Although the type of the variable will still be a std::function.

How can we time different parts of a program to determine what needs to be optimised?

Note: this is a C++11 feature.

Without using external profiling tools, the chrono header from the standard library can be used to time parts of a program’s execution with sub-millisecond accuracy, eg:

```
#include <chrono>

std::chrono::time_point<std::chrono::system_clock> lastTime;
…
auto thisTime = std::chrono::system_clock::now();
std::chrono::duration<float> deltaTime = thisTime - lastTime;
std::cout << deltaTime.count() * 1000.0f;
```

Compilation: Optimisation

What is the difference between the compile flags O1/O2/O3?

- -O1 enables a subset of gcc optimisation flags, usually flags that would not significantly increase compilation time.
- -O2 enables a larger subset of optimisation flags, including those that increase compilation time, however usually not optimisations that would significantly increase executable size.
- -O3 enables almost all optimisation flags, which includes flags that increase executable size. Note that this flag can significantly alter generated code, which may prevent gdb from giving accurate debugging information. If debugging information is needed with the -g flag, -O2 optimisation should normally be used instead.

What are loop unrolling and vectorisation?

Loop unrolling is not enabled by any O level flag and is instead enabled with -funroll-loops which unrolls loops whose number of iterations can be determined at compile time, while -funroll-all-loops will also unroll loops whose number of iterations is uncertain. Unrolling of loops can be done manually:

```
for (int i=0; i<n; i++)
{
        sum += data[i];
}
…
for (int i=0; i<n; i+=4)
{
        sum1 += data[i+0];
        sum2 += data[i+1];
        sum3 += data[i+2];
        sum4 += data[i+3];
}
sum = sum1 + sum2 + sum3 + sum4;
```

The potential advantage of this is that if there is a cache miss or branch prediction failure (if the code has branches) or any other stall in on calculation, the other three calculations don't have to wait for the stall and can still execute when using a super-scalar (parallel) CPU or on a CPU that has a SIMD instruction set that can perform the 4 calculations simultaneously.

Loop vectorisation is related, in that a CPU using SIMD can perform multiple calculations (in the above case, four) at once inside a loop, instead of one at a time. Loop vectorisation is enabled when using the -O3 flag. Note that vectorisation or unrolling will not necessarily improve performance, and in some rare cases can decrease performance, for example, some iterative sorting implementations may perform fast with -O2 than -O3.

Errors: None this week

Debugging: Exceptions

When is it preferable to use an assert or an exception?

Exceptions should be used whenever you want to provide the option to recover from an error, whereas an assert is used if you want the program's execution to fail. Typically asserts are used to check for things that should never happen or when debugging, while an exception is used to handle problems that will occur sometimes, and can be recovered from.

Exercises:

Use a lambda function to sort a vector of ints in reverse sorted order.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(int argc, char **argv)
{
	std::vector<int> v = {1, 10, 5, 6, 2, 8, 12, 7, 19, 14};
	std::sort(v.begin(), v.end(), [](int x, int y) { return x > y; });

	for (auto i : v)
		std::cout << i << " ";
	std::cout << "\n";
	return 0;
}
```

Write a template function capable of converting a string to an arbitrary data type, and have it throw an exception when it fails.

```
#include <iostream>
#include <sstream>
#include <stdexcept>
```

```
template <typename T>
T convert(const std::string &s)
{
	T t;
	std::istringstream ss(s);
	if ((ss >> t).fail())
		throw std::runtime_error("Failed to convert " + s);
	return t;
}

int main(int argc, char **argv)
{
	try
	{
		int i = convert<int>("100");
		std::cout << i << "\n";
	}
	catch (std::runtime_error &e)
	{
		std::cout << e.what() << "\n";
	}
	return 0;
}
```