Programming techniques

Week 14
C++11

C++11

- New version of the C++ standard released in 2011, only slowly being adopted by compiler vendors.
- Partial support in latest versions of GNU, clang, Intel and Microsoft compilers
- Need to add the --std=c++11 option when compiling

Random number generators

- The <random> header contains
  - random number generator engines
  - random number distributions

- The distributions are called with the engine as argument to create random numbers

```cpp
#include <random>
#include <iostream>

int main()
{
    std::mt19937 mt;    // create an engine

    // create four distributions
    std::uniform_int_distribution<int>     uint_d(0,10);
    std::uniform_real_distribution<double> ureal_d(0.,10.);
    std::normal_distribution<double>       normal_d(0.,4.);
    std::exponential_distribution<double>  exp_d(1.);

    // create random numbers:
    std::cout << uint_d(mt)   << "\n";
    std::cout << ureal_d(mt)  << "\n";
    std::cout << normal_d(mt) << "\n";
    std::cout << exp_d(mt)    << "\n";

    return 0;
}
```

Random number engines in C++11

- Linear congruential generators
  - minstd_rand0
  - minstd_rand

- Mersenne twisters:
  - mt19937
  - mt19937_64

- Other generators
  - ranlux24
  - ranlux48
  - knuth_b

Seeding C+11 generators

- There are two member functions for seeding
  - simple seeding by an integer:

```cpp
// create an engine
std::mt19937 mt;

// seed the generator
mt.seed(42);
```

  - seeding from a seed sequence

```cpp
// create a vector of seeds
int N = ....;

std::vector<int> seeds(N);

// fill the vector, e.g.  by an LCG generator minstdrand
...

// create a seed sequence and use it to seed a generator

std::seed_seq seq(seeds.begin(), seeds.end());
mt.seed(seq);
```

  - exercise: seed multiple generators for use with parallel MC program

Distributions in C++11

- Distributions are templated on the type of return values and the parameters of the distribution are passed to the constructor
- Uniform distributions
  - `uniform_int_distribution<T>`
  - `uniform_real_distribution<T>`
  - `generate_canonical<T> // uniform real numbers in [0,1)`

- Bernoulli distributions
  - `bernoulli_distribution<T>`
  - `binomial_distribution<T>`
  - `negative_binomial_distribution<T>`
  - `geometric_distribution<T>`

- Sampling distributions
  - `discrete_distribution<T>`
  - `piecewise_constant_distribution<T>`
  - `piecewise_linear_distribution<T>`

Distributions in C++11 (cont.)

- Poisson distributions
  - `poisson_distribution<T>`
  - `exponential_distribution<T>`
  - `gamma_distribution<T>`
  - `weibull_distribution<T>`
  - `extreme_value_distribution<T>`

- Normal distributions
  - `normal_distribution<T>`
  - `lognormal_distribution<T>`
  - `chi_squared_distribution<T>`
  - `cauchy_distribution<T>`
  - `fisher_f_distribution<T>`
  - `student_t_distribution<T>`

---

std::function

Doug Gregor

- A runtime polymorphic function object constructible from any compatible
  - function pointers
  - member function pointers
  - function object
  - lambda functions

- Great for callbacks, collections of callbacks, and threading
- Declaration of the result and argument types
  - `std::function<Result(Arg1,Arg2,Arg3)>`

- Our example use:

```
double simpson(std::function<double(double)> f,
               double a, double b, unsigned int N)
```

auto

- The new **auto** keyword tells C++11 to deduce the type of a variable from the initializer argument:

```
auto x = 3.141+5;
auto y = call_to_function_with_horrible_return_type();
```

- It saves complicated typing of types:

```cpp
#include <iostream>
#include <functional>

int f(int x) { return x+1;}
int main()
{
  // function pointer
  int (*p1)(int) = f;

  // easier function pointer with auto
  auto p2 =f;

  // or here we could just have used std::function
  std::function<int(int)> p3=f;

  std::cout << (*p1)(42) << std::endl;
  std::cout << (*p2)(42) << std::endl;
  std::cout << p3(42)    << std::endl;
}
```

Integrating a function of 2 variables

- Integrate exp(-*a**x) with Simpson over *x*

- Solution 0: a function with two arguments?

```cpp
#include "simpson.hpp"
#include <iostream>

// a function with two variables
double expax(double a, double x)
{
  return std::exp(a*x);
}

int main()
{
  // where do we set a?
  std::cout << simpson(expax,0.,1.,100) << std::endl;
  return 0;
}
```

- It does not even compile

```
lambda0.cpp:13:16: error: no matching function for call to 'simpson'
  std::cout << simpson(expax,0.,1.,100);
               ^~~~~~~
./simpson.hpp:7:15: note: candidate function not viable: no known conversion from 'double
(double, double)' to
      'std::function<double (double)>' for 1st argument;
inline double simpson(std::function<double(double)> f, double a, double b, unsigned int N)
```

Integrating a function of 2 variables

- Integrate exp(-*a**x) with Simpson over *x*

- Solution 1: a global variable .... **ugly**

```cpp
#include "simpson.hpp"
#include <iostream>

// an ugly global variable
double a;

// the function to be integrated
double expax(double x)
{
  return std::exp(a*x);
}

int main()
{
  a=3.4;
  std::cout << simpson(expax,0.,1.,100) << std::endl;
}
```

Integrating a function of 2 variables

- Integrate exp(-*a**x) with Simpson over *x*

- Solution 2: a function object .... **cumbersome**

```cpp
#include "simpson.hpp"
#include <iostream>
#include <cmath>

// a function object for exp(a*x)
class expax
{
public:
  // set the parameter a in the constructor
  expax(double a) : a_(a) {}

  // the function call operator calculates the function
  double operator()(double x) { return std::exp(a_*x);}

private:
  double a_; // the fixed parameter a
};

int main()
{
  double a=3.4;
  std::cout << simpson(expax(a),0.,1.,100) << std::endl;
}
```

Integrating a function of 2 variables

- Integrate exp(-*a**x) with Simpson over *x*

- Solution 3: create a function object using **std::bind** ... better

```cpp
#include "simpson.hpp"
#include <iostream>
#include <cmath>
#include <functional>

// a function with two variables
double expax(double a, double x)
{
  return std::exp(a*x);
}

int main()
{
  using namespace std::placeholders;

  double a=3.4;
  // bind one argument
  // _1, _2, .... are used for unbound arguments of the resulting function
  auto f = std::bind(expax,3.4,_1);
  std::cout << simpson(f,0.,1.,100) << std::endl;
}
```

Better solutions: lambda functions

- Lambda functions are unnamed functions declared inside a statement:

```cpp
#include <iostream>

int main()
{
  // create a function and store a pointer to it in f
  auto f = []() {std::cout << "Hello world!\n";};

  // call the function
  f();
}
```

Better solutions: lambda functions

- Lambda functions are unnamed functions declared inside a statement:

```cpp
#include <iostream>
#include <thread>

int main()
{
  // create a function and store a pointer to it in f
  auto f = []() {std::cout << "Hello world!\n";};

  // call the function in a thread
  std::thread t(f);
  t.join();
}
```

Integrating with a lambda

- Integrate exp(-*a**x) with Simpson over *x*

- Solution4: create a lambda function

```cpp
#include "simpson.hpp"
#include <iostream>
#include <cmath>

int main()
{
  double a=3.4;

  // create a lambda function
  // [=] indicates that the variable a should be used inside the lambda
  auto f = [=] (double x) { return std::exp(a*x); };

  std::cout << simpson(f,0.,1.,100) << std::endl;
  return 0;
}
```

Integrating with a lambda (shorter)

- Integrate exp(-*a**x) with Simpson over *x*
- Solution5: create a lambda function (even shorter)

```cpp
#include "simpson.hpp"
#include <iostream>
#include <cmath>

int main()
{
  double a=3.4;
  std::cout << simpson([=] (double x) { return std::exp(a*x); },0.,1.,100) << std::endl;
  return 0;
}
```

The name capture specification

- The [] indicate a lambda function, and how variables from the enclosing scope should be used (captured) inside the lambda

[]	Capture nothing (or, a scorched earth strategy?)
[&]	Capture any referenced variable by reference
[=]	Capture any referenced variable by making a copy
[=, &foo]	Capture any referenced variable by making a copy, but capture variable foo by reference
[bar]	Capture bar by making a copy; don't copy anything else
[this]	Capture the this pointer of the enclosing class

Recall type traits

- Remember the array addition problem

```
template <class T, class U>
Array<?> operator+(Array<T> const& x, Array<U> const& y)
{
  ...
}
```

- we solved it by using traits

```
template <class T, class U>
Array<typename sumtype<T,U>::type> operator+(Array<T> const& x, Array<U> const& y)
{
  ...
}
```

- but this traits class was hard to implement
- and why do we need to write a complex type trait if the compiler knows the type of a sum of T and U?

typeof

- C++11 has a typeof operator, giving the type of an expression

```
template <class T, class U>
struct sumtype
{
  typedef typeof(T()+U()) type;

}
```

- and we can just use it directly

```
template <class T, class U>
Array< typeof(T()+U()) > operator+(Array<T> const& x, Array<U> const& y)
{
  ...
}
```