HPCSE I

Some C++11

Some C++11 features

- I was asked to explain the new features of the C++11 standard that we’re going to use in the class besides the threading library
  - Random number generators in <random>
  - std::function template in <functional>
  - auto keyword
  - lambda functions
- Are many students already familiar with these?

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

```
#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:

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

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

  - seeding from a seed sequence

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

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

// fill the vector, ideally with a true entropy source or (not as good)
// by another generator such as std::minstd_rand
…

// 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:

```
#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?

```
#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**

```
#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**

```
#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

```
#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:

```
#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:

```
#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

```
#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)

```
#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

nullptr

- Is 0 a pointer or an integer?

```
// overload the function foo
void foo(char*);
void foo(int);

// what is called?

int main()
{
  foo(0);       // calls second foo
  foo(nullptr); // calls first foo
}
```

Fixed size integer types

- What is the name of the unsigned 32 bit integer type?
- Solution: new C99 / C++11 standard types

	signed	unsigned
8 bit	`int8_t`	`uint8_t`
16 bit	`int16_t`	`uint16_t`
32 bit	`int32_t`	`uint32_t`
64 bit	`int64_t`	`uint64_t`

decltype

- Do you remember the type traits for getting the type of a sum?
- Why is this so hard if the compiler already knows it?

```
// this should be int

decltype(5+3) x;

// and here is the traits class

template <class T, class U>
struct sum_type {
  typedef decltype(T()+U()) type;
};
```

Avoiding traits

- Another example
- Why is this so hard if the compiler already knows it?

```
// the type calculation problem
template <class T, class U>
Array<???> operator+(Array<T> const& x, Array<U> const& y);

// was solved by traits
template <class T, class U>
Array<typename sum_type<T,U>::type> operator+(Array<T> const& x, Array<U> const& y);


// is now made easier
template <class T, class U>
Array<decltype(T()+U())> operator+(Array<T> const& x, Array<U> const& y);
```

Suffix return type syntax

- Another example

```
// the type calculation problem
template <class T, class U>
??? add(T x, T y)
{
  return x+y;
}

// can be solved by same traits
template <class T, class U>
typename sum_type<T,U>::type> add(T x, T y)
{
  return x+y;
}


// could be made easier, but this does not compile since x and y are unknown yet
template <class T, class U>
decltype(x+y) add(T x, T y)


// simply declare the type at the end
template <class T, class U>
auto add(T x, T y) -> decltype(x+y)
{
  return x+y;
}
```

Simpler loops

- Simplify looping

```
std::vector<int> v;

// lots of typing
for (std::vector<int>::iterator it=v.begin(); it != v.end() ; ++it)
  std::cout << *it;

// ugly indexed access, will not work when changing to list
for (int = 0; i<v.size(); ++i)
  std::cout << v[i];

// nicer C++11 version
for (int x: v)
  std::cout << x;

// even nicer C++11 version
for (auto x: v)
  std::cout << x;
```