PHYS4070/7270 Worksheet: Week 7 (15/04/2021)

As before, the worksheet is broken into two parts.
Part A is a mini c++ tutorial on functions, templates, and functionals (functions which take functions as arguments).
As before, part A is completely optional - you may find these higher-level c++ features useful/interesting, but they are not required at all for the course.
Part B is the worksheet problems.

(Since we had only a single workshop last week, you may also want to use this time to finish last weeks worksheet.)

Part A: C++ functions, templates, functionals

While this is specific to c++, some general ideas hold for many languages (except templates)

1. Recap: function arguments; pass by value, reference, pointer

Pass by value (aka pass by copy):

- Gets a new copy of variable
- Changes to this variable inside function do not affect original variable

```
double sum_by_value(double x, double y) {
  // Gets own copy of x and y
  x += y;
  return x;
}
```

Pass by reference:

- Gets a *reference* to existing variable
- Changes to this variable inside function **do** affect original variable

```
double sum_by_reference(double &x, double &y) {
  // Get *reference* to x and y - operate on existing variable!
  x += y;
  return x;
}
```

Pass by *const* reference:

- Gets a *reference* to existing variable (so, no copy)

- Changes to this variable inside function are not allowed

```
double sum_by_constref(const double &x, const double &y) {
  // x += y; // would not compile, cannot modify 'const' variable
  return x + y;
}
```

Why?

- When passing around simple data (e.g., an `int`, `double`) usually doesn't matter
- If instead we are passing large data structures (like, an entire matrix), we don't want to unnecessarily copy data (this is slow)
    - Note: sometimes we still *do* want to copy the data, but this allows us to control when that happens
- Beware passing by reference -- errors may be hard to debug. Prefer passing by value or by const reference
- You can also pass by pointer, const pointer, etc. etc.

Example:
Interactive version: https://godbolt.org/z/xcWsjbnWs

```
#include <iostream>
// ... include 3 above functions ...
int main() {
  double x = 2.0;
  double y = 3.5;

  std::cout << "x=" << x << ", y=" << y << "\n";

  double result1 = sum_by_value(x, y);
  std::cout << "x=" << x << ", y=" << y << ", result=" << result1 << "\n";

  double result2 = sum_by_reference(x, y);
  std::cout << "x=" << x << ", y=" << y << ", result=" << result2 << "\n";

  double result3 = sum_by_constref(x, y);
  std::cout << "x=" << x << ", y=" << y << ", result=" << result3 << "\n";
}
```

Output:

```
x=2, y=3.5
x=2, y=3.5, result=5.5
x=5.5, y=3.5, result=5.5
x=5.5, y=3.5, result=9
```

Optional/Default arguments

- We can specify optional function arguments
- They must be at the end of the input list
- We must give them a default value in the function signature (declaration)

```
// declare function:
double func(double x, double y, double z = 2.5);
// z is optional - if not given, will assume value is 2.5

// Then, on calling the function:
func(3.1, 2.7);      // call with default argument
func(3.1, 2.7, 2.5); // exactly same as above
func(3.1, 2.7, 9.5); // call with different argument
```

Recursive functions

- We can write functions which call themselves
- Often very useful; but be careful not get stuck in an infinite loop - must have an exit condition
- Consider this simple example that calculates x^n:

```
double my_pow(double x, int n) {
  if (n == 0) {
    return 1.0;
  } else if (n < 0) {
    return 1.0 / my_pow(x, -n);
  } else {
    return x * my_pow(x, n - 1);
  }
}
```

2. Function overloading, and templates

Say we want a function that will work with more than one type, for example:

```
double sum(double a, double b) { return a + b; }
int sum(int a, int b) { return a + b; }
```

- C++ allows *function overloading* -- where the same function name can be used for different arguments (not allowed in C)
- Which version of the function will be called will depend on the *type* of input variable (this has potential for confusion, and hard-to-debug errors)
- This example may appear silly, since int can be converted to double; however, it becomes important for more complex types that cannot be so easily converted between (e.g., an array of `int` cannot be simply converted to an array of `double`) - we may also want to avoid conversions
- This gets tedious quickly - there are an infinite number of types (since types can be user defined)
- To be more generic, we may use *templates*

```
template <typename T>
T sum(T a, T b) {
  return a + b;
}
```

- Here, 'T' is the name of a generic type - 'T' may be anything (called 'T' by convention, but can be anything; can be more than one, e.g., `<typename T, typename U>` etc.)
- Technically, this will generate code for you, at compile time; it will write each function overload for you, based on which you actually use in your code
- Must be defined either in a header file, or in the same .cpp file you use them
- There are fancy ways to restrict which types T is allowed to take -- but we will ignore this complexity for now.
- This is just an extremely basic introduction to templates; the template system in c++ is its own entire language, and is extremely powerful (though sometimes difficult to use)

Example:
Interactive version: https://godbolt.org/z/6974asEax

```
#include <iostream>
#include <string>

// Declare a template: T is generic type, may be any type
template <typename T>
T sum(T a, T b) {
  return a + b;
}

int main() {
  int i1 = 1;
  int i2 = 2;
  int result1 = sum(i1, i2);  // calls sum(int, int)

  double d1 = 1.01;
  double d2 = 2.02;
  double result2 = sum(d1, d2);  // calls sum(double, double)

  // Even works with strings! (this may not be what you want)
  // Works with any type for which (a+b) is defined
  std::string s1 = "Hello ";
  std::string s2 = "world!";
  std::string result3 = sum(s1, s2);  // calls sum(string, string)

  std::cout << result1 << " " << result2 << " " << result3 << "\n";
}
```

3. Functionals: Passing functions to functions

- Sometimes it is extremely useful to have a function that takes another function as one of its arguments
- For example, we may want a function called 'integrate' that takes a function, `f(x)`, and integrates it between `x = [a,b]`, e.g.:

```
// nb: This doesn't work quite yet...
double integrate(Function f, double a, double b); //??
```

Can we do this? Yes! But not quite so simply. There are three key ways to do this: function pointers, templates, and using c++ library std::function

..using function pointers

- Old; we typically try to avoid this, because complicated
- We can pass the memory adress of a function
  - In c++, a function name converts implicitely to the memory address of a function (function pointer), so we do not need to use the `&` operator (though, we can)
- For a function:

```
OutType funcName(InType1 x, InType2 y, ...);
```

- Function pointer has the form:

```
OutType (*funcName)(InType1, InType2, ...)
```

- So, we could write our 'integrate' function as

```
double integrate(double (*f)(double), double a, double b);
```

- and call it like:

```
double result = integrate(f, a, b);
// double result = integrate(&f, a, b); // equivalent to above
```

..using templates

- Since a template can be *any* type (including function pointer), this gives a simple way to pass functions to functions
- This is powerful, however, it is complex; if you get the code wrong, sometimes the error messages will be extremely hard to decipher
- We could write our 'integrate' function as

```
template<typename Function>
double integrate(Function f, double a, double b);
```

..using std::function

- C++ provides a general class to hold a function (called *function objects*, or *callables*)
- Requires c++11 or newer; you may need to add -std=c++11 compile option
- need: `#include <functional>`
- Avoids complexity of using templates and function pointers
- When problems happen, usually get nice error messages
- Has type of form

```
std::function<OutType(InType1, InType2, ...)>
```

- Often used with `using` keyword (like an alias) to save typing

```
using FuncType = std::function<OutType(InType1, InType2, ...)>;
// Then, just use 'FuncType' as the type when needed
```

- e.g., our f(x) = x^2 function would be simply:

```
std::function<double(double)>
```

Example:
Functions that takes a function and integrates it (trapezoid rule) - using the three methods form above.
Interactive version: https://godbolt.org/z/KGnMKbxPc

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

// Simple function, f(x)=x^2, which we will integrate
double f(double x) { return x * x; }

// Function that integrates another function; uses function pointer
double integrate_fp(double (*f)(double), double a, double b) {
  int n_pts = 100;
  double dx = (b - a) / (n_pts - 1);
  double integral = (f(a) + f(b)) * (dx / 2.0);
  for (int i = 1; i < n_pts - 1; ++i) {
    double x = a + i * dx;
    integral += f(x) * dx;
  }
  return integral;
}

// ...; uses templates
template <typename Function>
double integrate_tmpl(Function f, double a, double b) {
// ... same as above ...
}

// ...; uses std::function
double integrate_std(std::function<double(double)> f, double a, double b) {
// ... same as above ...
}

int main() {
  double exact = 2.0 / 3;
  double result1 = integrate_fp(f, -1.0, 1.0);
  double result2 = integrate_tmpl(f, -1.0, 1.0);
  double result3 = integrate_std(f, -1.0, 1.0);
  std::cout << exact << ", " << result1 << ", " << result2 << ", " << result3 <<
"\n";
}
```

4. Lambdas

- Lambdas are "un-named" inline functions
  - (un-named is a little confusing, since they can have names..)
- Requires c++11 or newer; you may need to add -std=c++11 compile option
- A nice way to define (usually short) functions inline (inside `main()`)
- They are often passed as input to other functions (like STL standard algorithms)
- Have general form: `[captures](parameters){function body;}`
- For example, a lambda version of our 'f' function from above, which takes a double x and returns a double x*x, is

```
[](double x){ return x * x; }
```

- Captures allow us to include local data in a function - usually there are no captures, so the '[]' are empty

```
double y = 7.5;
[y](double x){ return y * x * x; }
```

- Combining with our 'integrate' function from above, we could have:

```
double result = integrate_std([](double x) { return x * x; }, -1.0, 1.0);
```

- We can give lambdas names, which makes code more readable. But, you *must* use `auto` for the type:

```
auto my_lambda = [](double x) { return x * x; };
double result = integrate_std(my_lambda, -1.0, 1.0);
```

Part B: Worksheet tasks

Consider function:

$$f(x) = e^{-x}\ \sin(15x + 1/2)$$

- Over domain x = [0, 3]
- Exact value of the integral is: 0.059972633417316158....

1. Calculate the integral of the above function over [0,3] using

- Trapezoid rule

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{[f(x_0) + f(x_{N-1})]\delta x}{2} + \sum_{i=1}^{N-2} f(x_i)\,\delta x.$$

- Simpsons rule

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{\delta x}{3}\left[f(x_0) + f(x_{N-1}) + 2\sum_{\text{even } j} f(x_j) + 4\sum_{\text{even } j} f(x_j)\right]$$

- k=5 Mixed-quadrature rule:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{k-1} w_i\,(f(x_i) + f(x_{N-i-1}))\,\delta x + \sum_{i=k}^{N-k-1} f(x_i)\delta x$$

and compare the resulting error (difference between calculated and exact value).

Do for:

- N = 11, 51, 101
- Note: N must be odd for Simpsons rule; and we must have N>2*k for mixed quadrature rule

The weights for a k=5 mixed quadrature rule are:

- w = {475.0/1440, 1902.0/1440, 1104.0/1440, 1586.0/1440, 1413.0/1440}

2. How many points do we need to use in the trapezoid rule for the error of the trapezoid rule to become smaller than that of the k=5 mixed-quadrature rule with n=100?

- Code a loop that calculates the integral using trapezoid rule with varying number of points, starting from n=100
- Continue until the error drops below that of the mixed-quadrature rule (with n=100)

3. [optional] Code an adaptive integration algorithm

- Adaptive methods need two things:
  - A method to integrate function over a given domain
  - A method to estimate the error of the integral

- They continuously sub-divide the integral into smaller-and-smaller sub-domains until the error for each drops below a given target
- Estimate the error by performing integral twice using a different number of points. For example, once using Simpsons rule with 3 points (n=3), and once with 7. (we don't need large n, since this will be taken into account by the recursive nature of algorithm)
- Estimate the error in the integral as the difference between these two values.
- If the error is too large, divide the integration region into two - and perform the same algorithm for each half
    - By continuing in this manner, we continuously divide sub-domains up until the error for the integral of each sub-domain drops below a specified value
- A rough pseudo-code algorithm is provided below; use it to guide your code
- This will be much easier if we use some concepts from Part A (functionals and recursive functions)
- Recursive functions can easily get out-of-hand - you may want to code in a counter that keeps track of the depth, and kills the function if the depth becomes too deep
- Test it by integrating f(x) as above
- If you don't want to code the algorithm, use my simple one (end of document)

```
Adaptive(Function, a, b, error_target):

 A1 = Integrate function (a,b) using Simpsons rule, n=3
 A2 = Integrate function (a,b) using Simpsons rule, n=7
 error = |A1-A2| (absolute value)

 if error is less than error_target:
   answer = A2
   return answer
   FINISHED

 otherwise:
   // call adaptive on sub-domains (a,m) and (m,b)
   // divide target_error by 2, since domain is half the size
   // (want the total error to be less than error_target)
   m = (a+b)/2
   answer = Adaptive(Function, a, m, error_target/2)
          + Adaptive(Function, m, b, error_target/2)
   return answer
   FINISHED
```

4. Use your (or my) adaptive method for 'tricky' function

Consider 'tricky' function

$$g(x) = \frac{f(x)}{x + 10^{-6}} = \frac{e^{-x}\sin(15x + 1/2)}{x + 10^{-6}}$$

Integrate this tricky below function over domain [0,1]:

- This function is very hard to integrate, due to near-singular
- Accurate value should be ~6.39019353...
- Integrate it using N=1001 using Simpsons and mixed-quadrature rules
    - You will notice the result is not even close to correct
- Now, integrate it using your adaptive method

5. Vacuum polarisation - Uehling potential

In quantum electrodynamics, the regular Coulomb interaction between two charges is perturbed by an effect called *vacuum polarisation*. It is caused by the creation of short-lived virtual electron-positron pairs out of the vacuum. This effect is largest in strong electric fields, and at very high energies. It must be taken into account for accurate calculations, including in atomic physics.

The Uehling potential, which describes the vaccuum polarisation correction to the regular Coulomb atomic potential, is (in atomic units):

$$
\begin{aligned}
V_{vp}(r) &= \int_1^\infty \mathrm{d}t\, \frac{-2\,Z\alpha^2}{3\pi\,r}\,\frac{\sqrt{t^2+1}}{t^2}\left(1+\frac{1}{2t^2}\right)e^{-2tr/\alpha} \\
&\approx \int_1^{1/r} \mathrm{d}t\, \frac{-2\,Z\alpha^2}{3\pi\,r}\,\frac{\sqrt{t^2+1}}{t^2}\left(1+\frac{1}{2t^2}\right)e^{-2tr/\alpha} \\
\text{where}\ \ \alpha &\approx 1/137.036
\end{aligned}
$$

The approximation in the second line is rough, but comes from fact that integrand becomes very small for t>r. This potential is largest at small r, where the electric field of the nucleus is extremely strong.

- The nuclear radius is on the order to ~1 fm (10^{-15} m) - which is around 10^-5 aB (10^-5 atomic units)
- Evaluate the Uehling potential at r = 1.0e-5 by performing the integral over dummy-variable t, using Simpsons rule with N=1001 and the adaptive method
- Accurate value should be ~-5.859605..
- To include the Uehling potential into calculations, we would need to evaluate it accurately at many points along r - the usefulness of adaptive methods in real-life examples should be clear

Example:
Simple function to integrate function f from [a,b], using adaptive method
Interactive version: https://godbolt.org/z/f865zExeY

```
#include <cassert>
#include <cmath>
#include <functional>
#include <iostream>

// 'using' (alias) for functional std::function
using Function = std::function<double(double)>;
```

```
// Function that integrates function f, over [a,b], using Simpsons rule.
// Note: n_pts must be odd (even sub-intervals)
double Simpsons(Function f, double a, double b, int n_pts) {
  assert(n_pts % 2 != 0 && "n_pts must be odd for Simpsons rule");
  double dx = (b - a) / (n_pts - 1);
  double integral = (f(a) + f(b)) * dx / 3.0;
  for (int i = 1; i < n_pts - 1; ++i) {
    // ternary operator: w = condition ? val_if_true : val_if_false;
    // i % 2 == 0 means (i, mod 2) - is 0 if i is even
    double w = (i % 2 == 0) ? (2.0 / 3) : (4.0 / 3);  // weight
    double x = a + i * dx;
    integral += f(x) * dx * w;
  }
  return integral;
}
```

```
// Integrates function f from [a,b], using adaptive method, until error drops
// below err_target. Recursive function.
// Note: This is very inefficient; I have tried to make it simple abd clear at
// the expense of performance
double adaptive(Function f, double a, double b, double err_target,
                int depth = 1) {
  // Calculate integral twice, once with 3, once with 7 points
  // 3 is minimum for Simpsons rule
  double integral_3 = Simpsons(f, a, b, 3);
  double integral_7 = Simpsons(f, a, b, 7);
  // Error is difference between these
  double err = std::abs(integral_3 - integral_7);
  if (err < err_target || depth > 100) {
    // if error is small, or depth exceeds limit, return best guess
    return integral_7;
  } else {
    double m = (a + b) / 2.0;  // mid-point
    // divide target error by 2, since each domain is half the size
    // Increase depth counter as we recursively call function:
    return adaptive(f, a, m, err_target / 2.0, depth + 1) +
           adaptive(f, m, b, err_target / 2.0, depth + 1);
  }
}
```