Typedefs

- Tired of syntax like `std::map<std::string, arma::mat>` ?
- Try a typedef !

```
typedef std::map<std::string, arma::mat> map_str_mat;
map_str_map my_named_matrices;
```

`map_str_map` ↑ can use this as typename

`map_str_mat` ↑ whatever you want

- Other examples

`map_str_dbl`
`vec_dbl`
`vec_vec_dbl`

Pointers

- An object (variable) lives somewhere in memory
- It has an address that specifies the location in memory

- The reference operator & returns the address

  &x returns address of x (0x7AF2C3)

- A pointer is an object (variable) that can store an address
- Declaration of pointer

```
int* ip;          variable of type pointer-to-int
                  or just "int pointer"
double* dp;       ---"--- pointer-to-double
arma::mat* my_ptr; ---"--- pointer-to-arma::mat
```

- Example:

```
double d = 3.14;
double* d_ptr = &d;
```

```
double* d_ptr;
d_ptr = &d;
```

- We can use a pointer to access the memory (the object stored in memory)
- Do this with the dereference operator *

```
cout << *d << endl;      prints "3.14" to screen
*d_ptr = 1.23;
cout << d << endl;       prints "1.23" to screen
```

- Pointers can be reassigned

(Cannot be ~~done~~ done with references!)

```
int* ip;

int x = 1;
ip = &x;          Now *ip would return 1

int y = 2;
ip = &y;          Now *ip would return 2
```

- Can pass pointers as function arguments:

```
void add_one(int* iptr)
{
    *iptr += 1;        adds 1 to the integer that iptr points to
}


int i = 10;
add_one(&i);           now i = 11

int* ip = &i;
add_one(ip);           now i = 12
```

- This use of pointers is similar to passing references

```
void add_one(int& iref)
{
   iref += 1
}

int i = 10;
add_one(i);
```

- Can also return pointers as function return type

```
int* get_int_ptr(...)      (More on this later)
```

- Working with pointers is <u>usually quick!</u>
  (Not copying/passing around large chunks of memory, just a tiny address)

- Example: Project 4, periodic boundary conditions

arma::Mat<int> S ⟷ $\begin{bmatrix} S_{00} & S_{01} & S_{02} \\ S_{10} & S_{11} & S_{12} \\ S_{20} & S_{21} & S_{22} \end{bmatrix}$

std::vector<std::vector<int*>> Sptrs

```
Sptrs[0][1] = &S(0,2);
Sptrs[1][1] = &S(0,0);
⋮
```

$\begin{bmatrix} \text{NULL} & \&S_{20} & \cdot & \cdot & \text{NULL} \\ \&S_{02} & \&S_{00} & \&S_{01} & \&S_{02} & \&S_{00} \\ \cdot & \&S_{10} & \&S_{11} & \&S_{12} & \cdot \\ \cdot & \&S_{20} & \&S_{21} & \&S_{22} & \cdot \\ \text{NULL} & \&S_{00} & \cdot & \cdot & \text{NULL} \end{bmatrix}$

- Changing a value like Sptrs[0][1] = -1 would set S(0,2) = -1
  (actually means *(Sptrs[0][1]) = -1)

- The fastest method we've found so far for the periodic boundary conditions in Project 4. (Mikkel has run tests)

[Ended here]

- Useful concept: pointer-to-function ("function pointer")

- Very ugly syntax for declaration

```
double (*fptr)(int, double);
```

fptr is now a pointer that can hold the address of functions like `double some_func(int, double)`

- Example:

```
double power_func(int n, double x)
{
    ...        (computes x^n)
}
```

```
double (*fptr)(int, double);
fptr = &power_func
```

- Can call it like

```
(*fptr)(2, 3.14)        returns 3.14^2
```

or simply `fptr(2, 3.14)`

— " —

- This mean we can treat functions much like other objects, e.g.

  - Pass in a function as an argument to another function
  - Have objects keep functions as member variables (e.g. provided by the user)
  - Put functions in containers, e.g.:

```
typedef double (*fptr_t)(double, double);
std::vector<fptr_t>;
std::map<std::string, fptr_t>;
```

Examples:
- You write an integrator and the user should provide the integrand
- You write a particle sim, and user should provide interaction function

- If this sounds tempting, but you don't like the syntax:
  Look up doc. for std::function (C++11)

```
#include <functional>

void print_num(int i)
{
   std::cout << i << std::endl;
}


std::function<void(int)> my_func = print_num;
my_func(10);
```

(or see later: functors)

- Operator -> is used to dereference member functions and variables

Example:

```
arma::mat M;
M.fill(1.0);

arma::mat* Mptr = &M;
Mptr -> fill(2.0);
```

(M is now filled with 2.0 everywhere)

This is the same as `(*Mptr).fill(2.0);`

```
*Mptr.fill(2.0);
```

would give compilation error

- Important use of pointers: **Control lifetime of objects**

Local variables

```
{
    double d = 3.14;      ← Local variable (memory on stack)
    ⋮
}   ← Here d and other local variables die (go out of scope)

double x = d;   ← Not allowed, d is dead.
```

- Can use **new** keyword to allocate memory that is **not deallocated** until we say so (with **delete**)

Much used with so-called "factory functions"

```
{
    Particle* my_particle_ptr = new Particle(0.511, -1)
    double* dptr;
    *dptr = 3.14;
}
double x = *dptr;   ← Not OK. The object dptr is dead.
double m = my_particle_ptr->mass;   ← OK, since the Particle instance still lives
⋮
delete my_particle_ptr;   ← Here the Particle instance is deconstructed and the memory freed
```

Forgetting this → memory leak!

- One reason why programs tend to run slower after a long time
- Memory typically released when program ends

- Powerful concept, means that ownership of an object can be passed from one part of the code to another (The owner is responsible for calling delete.)

- Gets tricky if many parts of the code share ownership of some object (Many parts of the code has a pointer pointing to a new'd Particle)
  Who calls delete and when? Requires bookkeeping...

- Since C++11: Smart pointers
  - #include <memory>
  - Pointer classes that take care of memory management
    - std::unique_ptr ← Pointer type that only allows one pointer to a given object. When the unique_ptr goes out of scope, the object it points to is destructed
    - std::shared_ptr ← Allows many pointers to same object. The object is only destructed when the last such shared_ptr goes out of scope.

- Pointers can be used in typecasting (big topic)
  - Much used when writing programs with class inheritance
  - Four specific casting operators for typecasting pointers

```
dynamic_cast <new type> (input)
static_cast <---> (---)
reinterpret_cast <---> (---)
const_cast <---> (---)
```

  - Example: reinterpret_cast allows us to simply interpret a part of memory any way we want:

```
int i = 10;
cout << *reinterpret_cast<double*>(&i); ;   Garbage number
cout << *reinterpret_cast<string*>(&i); ;   Garbage string
cout << *reinterpret_cast<int*>(&i); ;      10
```

- Common usecase:
  - Create pointer to some object / function
  - Cast it to void*
  - Pass it to some other code as a void*
  - If the other code knows what sort of thing it really points to, it can cast it from void* back to correct type

[We'll see an example of this with dynamic loading (plug-ins)]

Operator overloading

- Why can we add two arma::vec but not two std::vector<double> ?
- Because the arma::vec class has "overloaded" the "+" operator
- We can do that for our own classes!
  - Examples of operators: +, -, ==, [], ++, --, () (and many more!)
  - Example of reasonable use cases:

    Class MatrixBag

    - std::vector<arma::mat> matrices;
    - arma::mat & operator[](int i)
      {
      return matrices[i];
      }
    - MatrixBag operator+(const MatrixBag & A, const MatrixBag & B)
      {
      [Construct a new MatrixBag with the combined content of A and B.]
      [Return combined MatrixBag]
      }

- Often useful! Makes for readable code *if* operators have intuitive functionality (Should behave similar to built-in operators)

I.e., dont use planetA + planetB to mean something like collide_planets(planetA, planetB) !

Templates

- Where all those strange <sometype> things come from
- Write a bit of general code that can work with many variable types (a code template)
- Compiler looks throug code and figures out which specializations it needs to generate code for
- Example: std::vector is a class template (templated class)
  - The compiler finds e.g. vector<double> and vector<Planet> in the code
  - Generates those two versions of the std::vector class
  - Compiles
- Can be used with functions and classes
- Function template example:

```
template<typename T>
T my_difference(const T& a, const T& b)
{
    return 3*a - 5*b;
}

int main()
{
  :
  double d_diff = my_difference<double>(d1, d2);
  int i_diff = my_difference<int>(i1, i2);
  arma::mat A_diff = my_difference<arma::mat>(A1, A2);
  :
}
```

powerful, but code can become difficult to read and debug.

Functors

[Is it a class? Is it a function?
No, it's a functor!!!
(well, it's actually a class)]

- A functor is a class where the () operator is overloaded
- That means we can use class instances as functions
- Often very useful!
  - Can customize function when we create instance
  - Can "hold a state", e.g remember past function results (caching)
  - Can be passed around like any object
    → alternative to passing around function pointers

Example 1 (customizable)

```
class add_x
{ private:
    int x;
  public:
    add_x(int x_in)          <- constructor
    {
      x = x_in;
    }

    int operator()(int input)    <- overload () operator
    {
      return input + x;
    }
};
```

Use it as:

```
add_x add_10(10);      <- create functor that can add 10
int i1 = add_10(1);
add_x add_5(5);        <- add 5
i1 = add_5(i);
```

Example 2 (Caching)

```
class slow_computation
{
    double current_input;
    double current_result;

    double operator()(double new_input)
    {
        if (fabs(current_input - new_input) < epsilon)
        {
            return current_result;          <-- Shortcut!
        }

        : (Perform slow computation)

        current_result = new_result;
        current_input = new_input;
        return new_result;
    }

};
```

If we call

slow_computation(3.14)

twice, it will only do the slow computation once.

Macros

- Preprocessor statements, start with # (like #include)
- Modifies the code before it is compiled
- Example:

```
#define HELLO   "Hello, world!"

int main()
{
    cout << HELLO << endl;
}
```

[HELLO is replaced by "Hello, world!" before compilation]

- Much used in C code for math constants. The <cmath> library has a #define M_PI 3.1415....
- Can check for macro definitions using #ifdef, #ifndef, #endif
- Used to switch blocks of code on/off
- Example:

```
#define DEBUG

int main()
{
   :
   #ifdef DEBUG
     cout << "Debug: x = " << x << endl;
   #endif
   :
}
```

← Can now switch debug output on/off by including/excluding this line!

- Useful debug trick:

```
#define DEBUG
#define DEBUG_PREFIX __FILE__ << ":" << __LINE__ << ": "

int main()
{
  #ifdef DEBUG
    cout << DEBUG_PREFIX << "x = " << x << endl
    :
  #endif
}
```

- Macros can be very complicated!
- Can define mult-line macros
- Macros can take arguments
- Useful if you for some reason need to generate very similar code multiple times with small changes (.eg slightly different names)
- But avoid if possible...