Part IV

C++ as a better C

Nice C++ features

1. Comment style
2. Declare variables anywhere
3. Namespaces
4. Improved I/O approach
5. References
6. Improved memory allocation

Nice C++ features: Comment style

- C comments start with **/*** and end with ***/**
- C++ allows comments which start with **//** and last until the end-of-the-line.
- In addition, C-style comments are still allowed.
- C99 shares this nicety.

Example

C:

```
/* This is a C comment*/
```

C++:

```
// This is a C++ comment
```

Nice C++ features: Declare variables anywhere

- C: variables are declared at start of function or file.
- C++: you can mix statements and variable declarations.
- C99 shares this nicety.

Example

C:

```
double f() {
   double a,b;
   int c;
   a=5.2;
   b=3.1;
   for (c=0; c < 10; c++)
      a+=b;
   return a;
}
```

C++:

```
double f() {
   double a=5.2, b=3.1;
   for (int c=0; c < 10; c++)
      a+=b;
   return a;
}
```

Nice C++ features: Namespaces

- In larger projects, name clashes can occur.

  I had a 3d vector struct called `vector`*. Then came along the Standard Template Library, which defined* `vector` *to be a general array. Before namespaces, I had to rename* `vector` *to* `Vector` *in all my code.*

- No more: put all functions, structs, . . . in a namespace:

```
namespace nsname {
   ...
}
```

- Effectively prefixes all of `...` with `nsname::`
- Many standard functions/classes are in namespace `std`.
- To omit the prefix, do "`using namespace nsname;`"
- Can selectively omit prefix, e.g., "`using std::vector`"

Nice C++ features: I/O streams

Standard input/error/output

- Streams objects handle input and output.
- All in namespace **std**.
- Global stream objects (header: **<iostream>**)
  - **cout** is for standard output (screen)
  - **cout** is the standard error output (screen)
  - **cin** is the standard input (keyboard)
- Use insertion operator **<<** for output:

```
std::cout << "Output to screen!" << std::endl;
```

(**endl** ends the line and flushes buffer)

- Use extraction operator **>>** for input:

```
std::cin >> variable;
```

- These operators figure out type of data and format.

Nice C++ features: I/O streams

File stream objects (header: **`<fstream>`**)

- **`ofstream`** is for output to file.
  Declare with filename: good to go!

```
std::ofstream file("name.txt");
file << "Writing to file";
```

- **`ifstream`** is for input from a file.
  Declare with filename: good to go!

```
std::ifstream file("name.txt");
int i;
file >> i;
```

- Can also open and close by hand.

Nice C++ features: I/O streams

Example

C:

```
double a,b,c;
FILE* f;
scanf(f, "%lf %lf %lf", &a, &b, &c);
f = fopen("name.txt","w");
fprintf(f, "%lf %lf %lf\n", a, b, c);
fclose(f);
```

C++:

```
using namespace std;
double a,b,c;
cin >> a >> b >> c;
ofstream f("name.txt");
f << a << b << c << endl;
```

Nice C++ features: I/O Streams

Formatting (header: **<iomanip>**)

- Set width of next output:

```
double d = 14.545;
cout << "[" << setw(10) << d << "]" << endl;
```

```
[    14.525]
```

- Set significant digits of output to follow:

```
cout << "[" << setprecision(3) << d << "]" << endl;
```

```
[14.5]
```

- Set precision of next output:

```
cout << setw(9) << setfill('#') << d << endl;
```

```
#####14.5
```

- Change to scientific notation

```
cout << scientific << d << endl;
```

```
1.454e+01
```

(revert with **fixed**)

Nice C++ features: I/O Streams

Gotcha: text (ASCII) versus binary I/O

While easy, writing ASCII is rarely the best choice in scientific code.
“What is wrong with ASCII,” you ask, “isn’t it nice that it is readable?”

- ASCII typically doesn’t preserve the data’s accuracy.
- ASCII typically takes more space than writing binary.
- Writing and reading ASCII is much slower than binary:

Writing 128M doubles

Format	/scratch (GPFS)	/dev/shm (RAM)	/tmp (disk)
ASCII	173s	174s	260s
Binary	6s	1s	20s

Writing binary

std::ofstream has a **write(char*,int)** member function.

std::ifstream has a **read(char*,int)** member function.

Remember **sizeof***!*

Nice C++ features: References

- A reference gives another name to an existing object.
- References are similar to pointers.
- Do not use pointer dereferencing (`->`), but a period `.`
- Cannot be assigned null.

Standalone definition (rare)

```
type & name = object;
```

- *object* has to be of type *type*.
- *name* is a reference to *object*.
- *name* points to *object*, i.e., changing *name* changes *object*.
- Members accessed as *name.membername* (as you would for *object*).

Definition as arguments of a function

```
returntype functionname(type & name, ...);
```

Nice C++ features: References

Example

To change a function argument, need a pointer in C:

```
void makefive(int * a) {
   *a = 5;
} ...
int b = 4;
makefive(&b); /* b now holds 5 */
```

C++: can pass by reference using **&**:

```
void makefive(int & a){
   a = 5;
} ...
int b = 4;
makefive(b); /* b now holds 5 */
```

Nice C++ features: References

Gotcha: Avoid copies of objects in function calls

Compare these two functions

```
struct Point3D {
   double x,y,z;
};
void print1(Point3D a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z << std::endl;
}
void print2(Point3D& a){
   std::cout << a.x << ' ' << a.y << ' ' << a.z << std::endl;
}
```

- Calling **print1** copies the content of **a** to the stack (24 bytes).
- Calling **print2** only copies the address of **a** to the stack (8 bytes).
- Memory copies are not cheap!
- If we do this with classes, a so-called constructor is called everytime **print1** is called, whereas **print2** still only copies 8 bytes.

Nice C++ features: Improved memory allocation

Basic allocation

```
type* name = new type;
```

Allocation with initialization

```
type* name = new type(arguments);
```

Array allocation

```
type* name = new type[arraysize];
```

Basic de-allocation

```
delete name;
```

Array de-allocation

```
delete [] name;
```

Nice C++ features: Improved memory allocation

Example

```
struct credit {
   long number, balance;
};
```

No more of this mess:

```
#include "stdlib.h"
struct credit* a;
double * b;
a = (struct credit*)malloc(sizeof(struct credit));
b = (double *)malloc(sizeof(double )*10000);
...
free(a); free(b);
```

Instead, simply:

```
credit* a = new credit;
double * b = new double [10000];
...
delete a; delete[] b;
```

HANDS-ON 1:

Use these nice c++ features to rewrite the matrix routines and the main function.

Hands-on 1 - instructions

- Make a directory for this course in your home directory, e.g.

```
$ mkdir scinetc++
$ cd scinetc++
```

- Copy the example directory from `scinetcppexamples.tgz`
  This is the matrix example that we looked at after the c review.
- Work from that new directory:

```
$ cd example
```

- Try to build the code

```
$ make
```

- If successful, try to execute the program

```
$ ./main
```

Every with me so far?

Hands-on 1 - instructions continued

- Copy the **example** directory to **example_nice**, and work there:

```
$ cd ..
$ cp -r example example_nice
$ cd example_nice
```

This will be the first c++ version of the matrix example.

- Rename a the .c files to .cpp files:

```
$ mv main.c main.cpp
$ mv mymatrix.c mymatrix.cpp
```

- Copy the makefile for this set of files from the **example_nice** directory in **scinetcppexamples.tgz**.
- Try to build and run the code

```
$ make
$ ./main
```

Still with me?

Hands-on 1 - instructions continued

Modify the code to use (one at a time):

1. C++ comment style
2. Declarations of iteration variables in for loops
3. Improved memory allocation
4. Improved I/O
5. References

Test that the code builds and runs after implementing each feature.

Hands-on 1 - answers

If you did not quite get there, or if you have a few remaining bugs:

- Copy the c++ version I made, from the **`example_nice`** directory in **`scinetcppexamples.tgz`**, so we can continue later.
- Test that the code builds and runs.
- Be sure to look at the source code and see if it make sense to you.