INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

WHITESPACE?

- C/C++ is a whitespace-independent language — ie. those spaces, tabs, and newlines are basically ignored in the source code. The following codes are all doing the same thing:

```
int add(int x, int y)
{
    return x+y;
}
```

```
int add(int x, int y)
{  return x+y; }
```

```
int add(int x,int y){return x+y;}
```

- However the space within a string (within a pair of “) does mean something:

```
std::cout << "3.14159265\n";
std::cout << "3.14   159   265\n";
std::cout << "3.14" "159" "265\n";
std::cout << "3.14"
"159"
“265\n";
```

terminal output

```
3.14159265
3.14   159   265
3.14159265
3.14159265
```

Quoted text separated by whitespace (spaces, tabs, or newlines) will be concatenated!

```
true) { if (min==max) { min = mean.getVal() - sigma.getVal()*10.; max = mean.getVal() + sigma.getVal()*10.;
oRealVar main_var(key, "", mean.getVal(), min, max); main_var.setError(sigma.getVal()); RooGaussian gauss(F
gma); if (sigma.getVal()==0.) main_var.setConstant(true); cout << ">>> BuildGaussianConstaint: " << key <<
gma.getVal() << " [" << min << ", " << max << "]" << endl; wspace->import(gauss); } void BuildGaussianConst
uble mean, double sigma, double min = 0., double max = 0., bool always_positive = true) { RooRealVar mean_c
oRealVar sigma_const(Form("%s_sigma",key.Data()), "", sigma); BuildGaussianConstaint(wspace, key, mean_cons
id BuildLognormalConstaint(RooWorkspace* wspace, TString key, RooAbsReal& mean, RooAbsReal& kappa, double m
ways_positive = true) { if (min==max) { min = mean.getVal() - (kappa.getVal()-1.)*mean.getVal()*10.; max =
appa.getVal()-1.)*mean.getVal()*10.; if (always_positive && min<0.) min = 0.; } RooRealVar main_var(key, ""
in_var.setError((kappa.getVal()-1.)*mean.getVal()); RooLognormal Lognormal(Form("%s_lnn",key.Data()), "", m
appa.getVal()==1.) main_var.setConstant(true); cout << ">>> BuildLognormalConstaint: " << key << ": mu = "
ppa.getVal() << " [" << min << ", " << max << "]" << endl; wspace->import(Lognormal); } void BuildLognormal
uble mean, double kappa, double min = 0., double max = 0., bool always_positive = true) { RooRealVar mean_c
oRealVar kappa_const(Form("%s_kappa",key.Data()), "", kappa); BuildLognormalConstaint(wspace, key, mean_con
ways_positive); } void PrepareGlobalVariables(RooWorkspace* wspace, RooWorkspace* wspace_base = 0) { cout <
dl; if (wspace_base!=0) { cout << ">>> import from existing workspace." << endl; wspace->import(*wspace_bas
mport(*wspace_base->var("BF_bd")); wspace->import(*wspace_base->var("Mass")); wspace->import(*wspace_base->
mport(*wspace_base->var("BDT")); wspace->import(*wspace_base->var("Tau")); wspace->import(*wspace_base->var
at("GlobalCat")); wspace->import(*wspace_base->cat("SpCat")); wspace->import(*wspace_base->cat("SelCat"));
pace->import(*wspace_base->var("Weight")); wspace->import(*wspace_base->var("dblmu_corr_scale")); wspace->i
df("one_over_BRBR_gau")); wspace->import(*wspace_base->pdf("fs_over_fu_gau")); wspace->import(*wspace_base-
mport(*wspace_base->var("TransTau_bs")); wspace->import(*wspace_base->function("EffTau_bs")); wspace->impor
mport(*wspace_base->pdf("syst_semi_shape_gau")); wspace->import(*wspace_base->pdf("syst_h2mu_shape_gau"));
constraints." << endl; vector<TString> keys; vector<double> values; keys.push_back("fsfu:val"); keys.push_b
ys.push_back("fsfu_S13:val"); keys.push_back("fsfu_S13:err"); keys.push_back("BF_BuToKpsiK:val"); keys.push
ys.push_back("BF_JpsiToMuMu:val"); keys.push_back("BF_JpsiToMuMu:err"); keys.push_back("BF_BsToMuMu:val");
ys.push_back("EffTau_BsToMuMu:val"); keys.push_back("EffTau_BdToMuMu:val"); enum { _fsfu, _fsfu_err, _fsfu_
F_bu2jpsik_err, _BF_jpsi2mumu, _BF_jpsi2mumu_err, _BF_bs2mumu, _BF_bd2mumu, _EffTau_bs2mumu, _EffTau_bd2mum
ternal_parameters.tex",keys,values); RooRealVar BF_bs("BF_bs", "", values[_BF_bs2mumu], 0., 1e-8); RooRealV
, 1e-8); wspace->import(BF_bs); wspace->import(BF_bd); double one_over_BRBR_val = 1./ (values[_BF_bu2jpsik]
e_over_BRBR_err = one_over_BRBR_val * sqrt(pow(values[_BF_bu2jpsik_err]/values[_BF_bu2jpsik],2) + pow(value
); BuildGaussianConstaint(wspace, "one_over_BRBR", one_over_BRBR_val, one_over_BRBR_err); BuildGaussianCons
lues[_fsfu_err]); BuildGaussianConstaint(wspace, "fs_over_fu_S13", values[_fsfu_S13], values[_fsfu_S13_err]
ss_bound[1]); RooRealVar ReducedMassRes("ReducedMassRes", "", ReducedMassRes_bound[0], ReducedMassRes_bound
T_bound[1]); RooRealVar Tau("Tau", "", Tau_bound[0], Tau_bound[1]); RooRealVar TauRes("TauRes", "", TauRes_
obalCat("GlobalCat", ""); GlobalCat.defineType("undefined",-1); for (auto& cat: CatMan.cats) GlobalCat.defi
Cat("SpCat", ""); ... i++) SpCat.defineType(Form("sp%d",i),i);
oCategory SelCat("SelCat", ""); SelCat.defineType("failed",0); SelCat.defineType("passed",1); wspace->impor
pace->import(BDT); wspace->import(Tau); wspace->import(TauRes); wspace->import(GlobalCat); wspace->import(S
mport(Weight); RooRealVar dblmu_corr_scale("dblmu_corr_scale", "", 1.0, 0., 10.0); wspace->import(dblmu_cor
irCat.defineType("seagull",0); PairCat.defineType("cowboy",1); wspace->import(PairCat); RooRealVar TransTau
```

It is possible to write a code without conventional formatting…but no one can really understand it!

BASIC CODE FORMATTING

- Unlike some other languages (e.g. python), C/C++ does not enforce the formatting, given the white spaces are basically ignored.
- However there are still some general recommendations for how to produce the most readable code.

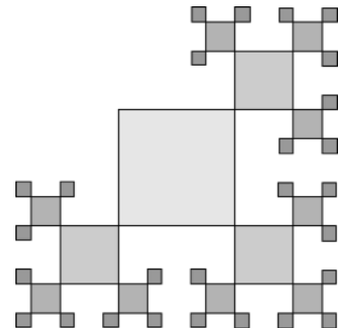
- **Function braces** — there are two acceptable styles, e.g.

```
void func() {
    // Your code here
}
```

"Google C++ style", as it is more compact, can show more code on the screen, etc.

```
void func()
{
    // Your code here
}
```

Easier to detect brace pairs, and they are always indented at the same level.



BASIC CODE FORMATTING (II)

- **Indented statements** — statements within curly braces should start one tab (or fixed spaces) in from the opening braces.

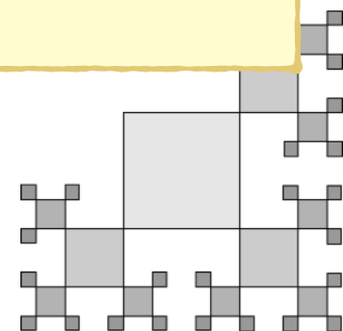
```
void func()
{
    int num(0);
    std::cout << “func() start\n”;
}
```

- **Limited line length** — lines should not be too long, as 80 characters are the typical maximum length. Extra indent for the subsequent line(s).

```
std::cout << "This is a really, "
    "really, really, really, "
    "really, really, really, "
    "really, really, really, "
    "really, long line.”
```

- If a line is split with an operator, place the operator at the beginning of the next line.

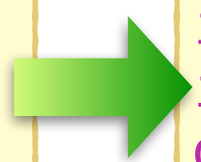
```
double answer = 1.2 * 3.4
    + 5. * 6. * 7.
    - 8. / 9.;
```



BASIC CODE FORMATTING (III)

- Use whitespace wisely — "align" the values and so on. e.g.

```
int index = 0;
int count_trees = 23;
double init_value = 9.8;
```

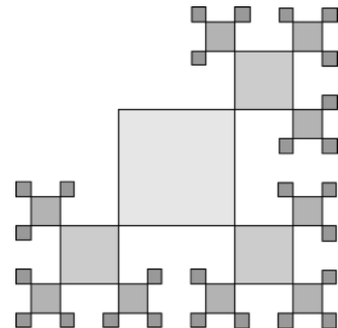


```
int    index       = 0;
int    count_trees = 23;
double init_value  = 9.8;
```

- C/C++ allows you to pick up your own preferred formatting; however if you are working on someone else's code, it is better to preserve the original style — consistency is important!

- Some modern IDEs or code editors can help you format your code (e.g. an IDE may automatically indent the statements inside a function, etc).

  - It is recommend to follow the automatic formatting.

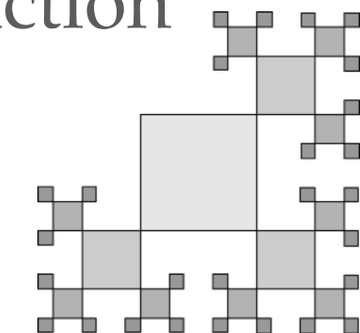


FORWARD DECLARATIONS

- A **forward declaration** tells the compiler about the existence of an identifier, before actually defining the identifier. e.g.

```
#include <iostream>

double add(double x, double y); ← forward declaration of add()
                                  as a function prototype
int main()
{
    std::cout << add(2.3,4.5) << "\n";
}

double add(double x, double y) ← actual definition of add(),
{                                includes the function body
    return x+y;
}
```

- The declaration statement is called a **function prototype**, consists of the function's return type, name, parameters, but no function body.



FORWARD DECLARATIONS (II)

- If the **declaration** is ***NOT*** provided, and the function definition is placed after the caller, it does not even compile!

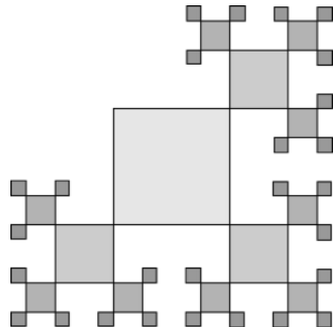
missing_proto.cc

```
#include <iostream>

int main()
{
    std::cout << add(2.3,4.5)
              << "\n";
}

double add(double x, double y)
{
    return x+y;
}
```

```
$ g++ missing_proto.cc
missing_proto.cc:5:18: error:
use of undeclared identifier 'add'
    std::cout << add(2.3,4.5)
                 ^
1 error generated.
```

- The name of parameters of the prototype can be omitted in principle, although preserving the names is a better practice.

```
double add(double, double);
```



FORGETTING THE FUNCTION BODY

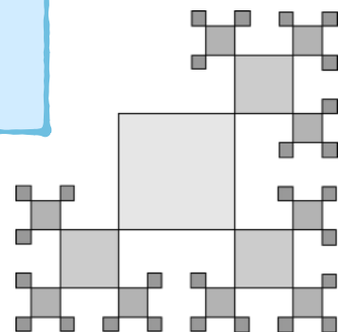
- If the **function body** is ***NOT*** provided, and a forward declaration is properly given, then the code can be compiled, but will fail at linking stage (produce final executable).

missing_body.cc

```
#include <iostream>
double add(double x, double y);
int main()
{
    std::cout << add(2.3,4.5) << "\n";
}
```

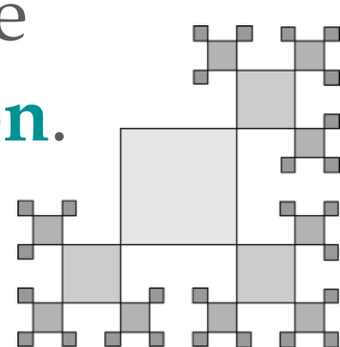
```
$ g++ missing_body.cc
Undefined symbols for architecture x86_64:
  "add(double, double)", referenced from:
      _main in missing_body-22a30b.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1
```

The above code still can be compiled by adding -c to the g++ commands.



DECLARATIONS VS. DEFINITIONS

- A **declaration** is a statement that satisfy the compiler about the existence of an identifier and its type information.
- Compiler itself does not need a complete **definition**, which actually implements (functions/types) or instantiates (variables) the identifier, to work.
  - However the program does require the definition of the code to satisfy the linker, either linked in the same executable, or loaded as a dynamic linking library.
  - A definition is needed to satisfy the linker. If you use an identifier without providing a definition, the linker will error.
- In our earlier example, if one puts the add() definition before main(), in this case the **definition also serves as a declaration**.



DECLARATIONS VS. DEFINITIONS (II)

- There is a so-called **"one definition rule"**, or ODR, in C/C++:
  - Within a given file, a function, object, type, or template should have exactly one definition.
  - Within a given program, an object or normal function should have exactly one definition, *note a program can have multiple files!*
  - Types, templates, and inline functions and variables can have identical definitions in different files.

Remark #1: in C++, overloading a function is possible (ie. same function name with different parameters), in this case they are considered as distinct functions.

Remark #2: there should be only one definition as ODR, but multiple declarations are allowed, although adding just one declaration already works!

WORK WITH MULTIPLE SOURCE FILES

- Remember we have a "broken-link" code given earlier:

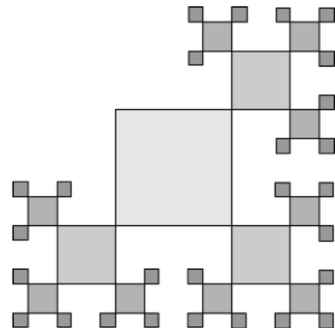
missing_body.cc

```
#include <iostream>

double add(double x, double y);

int main()
{
    std::cout << add(2.3,4.5) << "\n";
}
```

- One can actually resolve the linker problem with the definition of add() function included in a separated file, e.g.

func_add.cc

```
double add(double x, double y)
{
    return x+y;
}
```



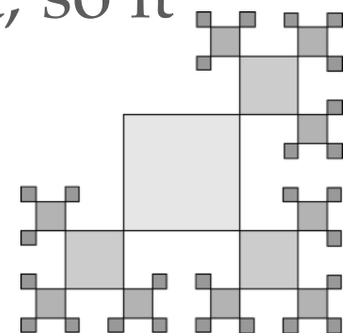
WORK WITH MULTIPLE SOURCE FILES (II)

- One can either compile the second file, func_add.cc, together with the main code, or compile it separately and link together afterwards.

```
$ g++ missing_body.cc func_add.cc ← direct produce a.out from 2 source files
```

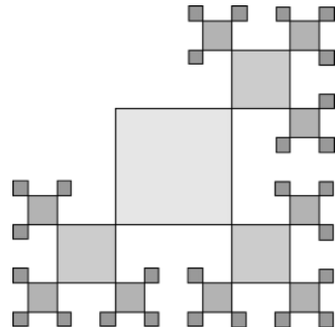
```
$ g++ -c missing_body.cc func_add.cc ← compile the sources
$ g++ -o run missing_body.o func_add.o ← produce executable "run" from
                                         2 object files
```

- One should be able to compile the source files in any order (*e.g. no knowledge about other sources*).
  - It is a bad idea to add `#include "func_add.cc"` in your code as well, as the compiler will be treated the contents in the same file.
- Eventually we will have to work with multiple source files a lot, so it is good to understand how to compile a multiple-file project!



NAMING COLLISIONS & NAMESPACE

- C++ requires that all identifiers be non-ambiguous. If two identical identifiers are introduced into the same program, the compiler or linker will produce a **naming collision** (or naming conflict) error:
  - Two definitions are introduced into the same file ⇒ compiler error;
  - Two definitions are introduced into separate files ⇒ linker error;
- The chance of name collisions increases as the scale of program, and **namespace** is one of the mechanisms for avoiding naming collisions.
  - The namespace provides a **namespace scope** to the names declared inside, and won't be mistaken for identical names in other scopes.
  - And we are already using it, just to remember the **std** of the std::cout, std::cin, and std::endl!



GLOBAL NAMESPACE

- Any name that is not defined inside a class, function, or a namespace is considered to be part of an implicitly defined **global namespace** (or the global scope).
- All of the functionalities in the standard library are kept in the std namespace (std = standard); **std::cout is just cout in the std namespace**.
- You have to tell the compiler that the identifier lives inside the namespace when accessing it.

There is no conflict of the two "cout" and can be compiled. Although such a confusing coding is still better to be avoid.

```
#include <iostream>
void cout(int x) ← in global scope
{
    std::cout << x << "\n";
}          ↑ in std scope
int main()
{
    cout(42);
}
```

ACCESSING NAMESPACE

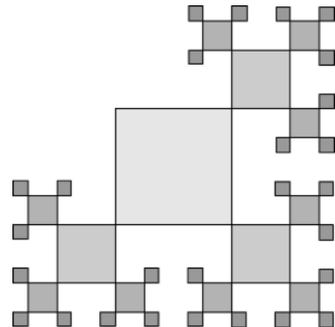
- The most straightforward way to access the identifier under a namespace is through by the **scope resolution operator ::**, as we already did before:

```
#include <iostream>
int main()
{
    std::cout << "Hello, World!" << std::endl;
}
```

- Another way is to use a **using** directive statement, e.g.

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, World!" << endl; ← std:: is omitted.
}
```

Remark: this is not very recommended since it gives up the merit of namespace...



USER-DEFINED NAMESPACES

- We can define our own namespaces via the **namespace** keyword as **user-defined namespaces**. e.g.

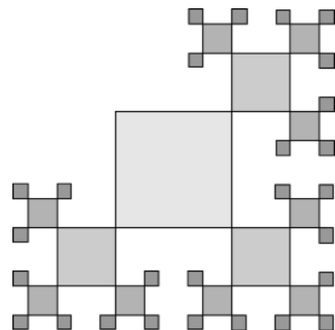
main.cc

```cpp
#include <iostream>

namespace foo {
    void print() {
        std::cout << "Foooo!\n";
    }
}

namespace goo {
    void print() {
        std::cout << "Goooo!\n";
    }
}

int main()
{
    foo::print(); ← call to print() under namespace foo
    goo::print(); ← call to print() under namespace goo
}
```

No compiling error!

```
$ g++ main.cc
$ ./a.out
Foooo!
Goooo!
```

There is no conflicts between the two print(), as they are placed under two different namespace.



USER-DEFINED NAMESPACES (II)

- The scope resolution operator :: can also be used without any preceding namespace. In such a case, the identifier in the global namespace will be accessed.
- Blocks with the same namespace name in multiple locations (across multiple files, or multiple places in the same file) is allowed, and will be considered as part of the same namespace.

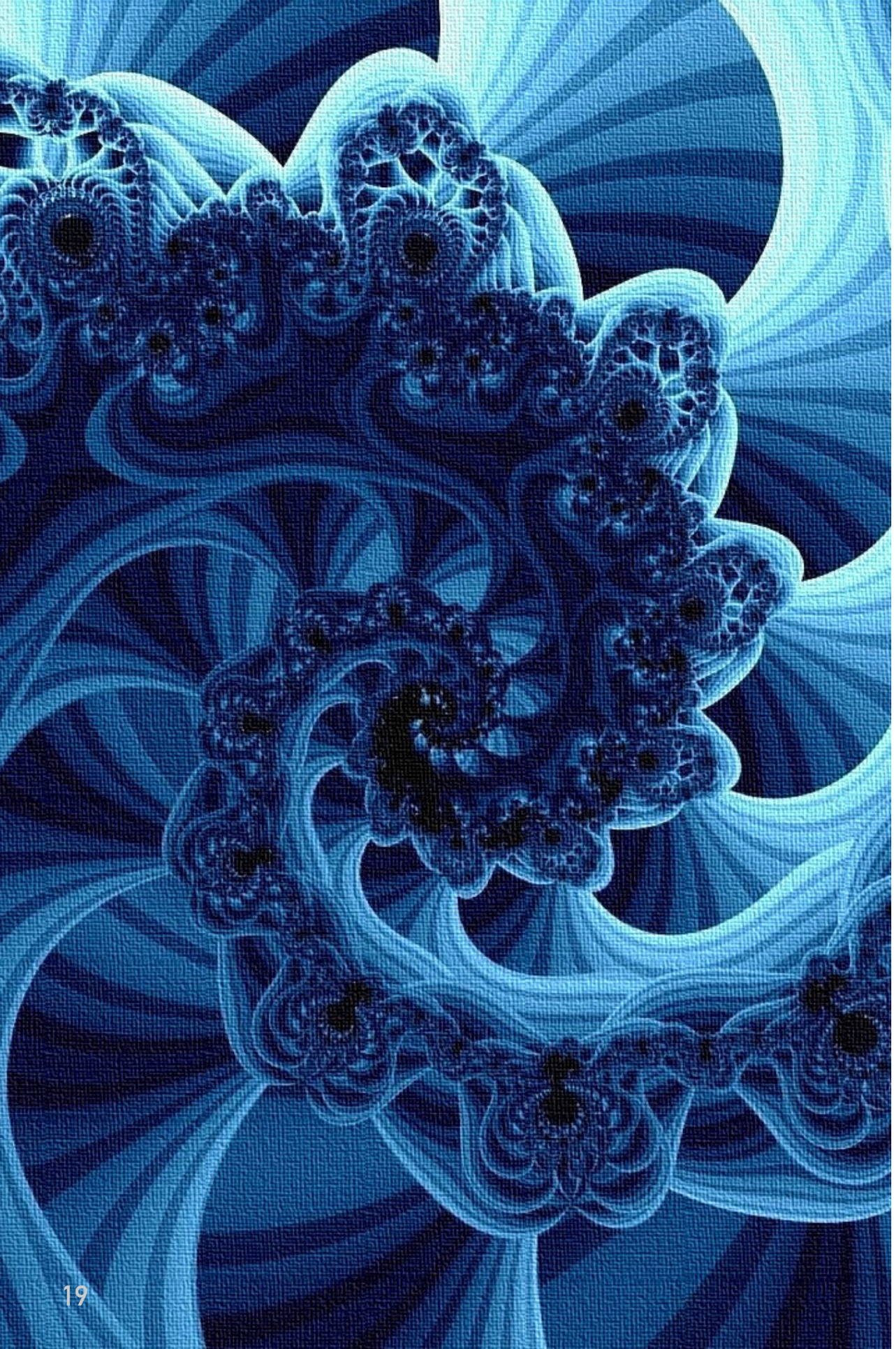
```
#include <iostream>

void print() {
    std::cout << "Global!\n";
}

namespace foo {
    void print() {
        std::cout << "Foooo!\n";
    }
}

namespace foo {
    void print_twice() {
        print();
        print();
    }
}

int main()
{
    ::print();
    foo::print_twice();
}
```

← this call to the print() in foo

← this call to global print()

```
Global!
Foooo!
Foooo!
```

terminal output



MODULE SUMMARY

- In this module we have discussed how to make your code “looks good” — e.g. how to use the whitespace and indentation wisely, and the standard issue about the namespace.
- We will continue to discuss the preprocessor and how to organize the codes with header files.