MODULE C3: C/C++ BASIC ELEMENTS II

INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

VARIABLE INITIALIZATION

- Recap: in programs the computer memory are accessed with an **object**, while a named object is the **variable**.
- C/C++ does not initialize most variables to a given value (e.g. zero) automatically. When a variable is assigned a memory location by the compiler, the default value of that variable is **whatever (*garbage*) value happens to already be stored in that memory location**!
  - A variable does not been set to a known value is called an **uninitialized variable**.
  - Lack of initialization is a performance optimization/consideration inherited from C. Useless initialization was a kind of waste computing power in old days.
  - You are recommended to initialize your variables since the cost of doing so is nearly nothing nowadays, however in some cases the initialization can be still omitted for optimization purposes.

UNINITIALIZED VARIABLE, WHAT IF

- Using the uninitialized variables can lead to unexpected results, e.g.

```
int main()
{
    int x;
    std::cout << "The value of x = " << x << "\n";
    return 0;
}
```

you are encouraged to try this, and see what would happen!

- The computer simply assigns unused memory to x. But what value will the program above print? The answer is *"who knows!"*, and the answer may (*or may not*) change every time you run the program.
- Some compilers initialize the variables to some preset value when you enable a debug or the optimizing mode (for g++, you can try to attach **-g** or **-O** and see if the "random" behavior changes or not).
- Some compilers may attempt to detect if a variable is being used without being initialized as well.

UNDEFINED BEHAVIOR

- Using uninitialized variables is one of the common mistakes, and is very challenging to debug (*because the program may still run fine by chance!*) — this is why "initialize your variables" is recommended nowadays.
- **Undefined behavior** is the result of the program whose behavior is not well defined by the language itself. Using uninitialized variables is just one of those cases!
  - Your program produces different results every time it is run (*not by design*).
  - Your program consistently produces the same but incorrect result.
  - Your program behaves inconsistently (*sometimes okay, sometimes not*).
  - Your program crashes, either immediately or later.
  - Your program works on some compilers/machines but not the others.
  - Your program produce different results when you change some other (*seemingly*) unrelated code.

The good practice is to avoid any situation that result in undefined behavior!

IDENTIFIERS

- The name of a variable (or a function, a type) is called an **identifier**. Naming identifiers should follow the given regulations:
  - The identifier can not be a **keyword**.
  - The identifier can only be composed of letters (lower or upper case), numbers, and the underscore.
  - The identifier must begin with a letter (lower or upper case) or an underscore.
  - C/C++ is case sensitive, and thus distinguishes between lower and upper case letters.
- However there is a recommended convention: **variable and function names** are better to begin with a **lowercase letter**, e.g.

```
int value; ← good
int Value; ← not good
int VALUE; ← not good
int VaLuE; ← not good
```

- Identifier names that start with a capital letter are typically used for user-defined types (such as classes).

C/C++ KEYWORDS

- C++ reserves a set of 84 **keywords**, which have special meanings within the language:

(*) added after C++11

alignas (*)	char32_t (*)	enum	namespace	return	try
alignof (*)	class	explicit	new	short	typedef
and	compl	export	noexcept (*)	signed	typeid
and_eq	const	extern	not	sizeof	typename
asm	constexpr (*)	FALSE	not_eq	static	union
auto	const_cast	float	nullptr (*)	static_assert	unsigned
bitand	continue	for	operator	static_cast	using
bitor	decltype (*)	friend	or	struct	virtual
bool	default	goto	or_eq	switch	void
break	delete	if	private	template	volatile
case	do	inline	protected	this	wchar_t
catch	double	int	public	thread_local	while
char	dynamic_cast	long	register	throw	xor
char16_t (*)	else	mutable	reinterpret_cast	true	xor_eq

- In C++11 override and final are special identifiers too, but are not reserved.

MULTI-WORD IDENTIFIERS

- If the variable or function name is multi-word, there are two common conventions: **words separated by underscores** ("snake_case") or **intercapped**.
- Avoid naming your identifiers starting with an underscore, as these are typically reserved for OS or library use.
- The identifiers should make **clear about the meanings**; avoid abbreviations if possible.
- A good practice is to make the length of an identifier name proportional to the range of the variable, ie. very brief variable such as **i, j**, better to be used only locally.

```
int count;        ← mixed, can be ambiguous
int pcount;       ← bad, what is the p?
int people_count; ← good, snake_case
int peopleCount;  ← good, intercapped
int index;        ← mixed, can be ambiguous
int page_index;   ← good, snake_case
int pageIndex;    ← good, intercapped
int _index;       ← bad, reserved for OS/lib
int data;         ← bad, too generic
int x,y;          ← mixed, only good locally
```

LITERALS & OPERATORS

```
double pi = 3.1415927;
double R = 2.5;
double area_of_pie = pi*R*R/4.0;
std::cout << "Area of a quarter pie is " << area_of_pie;
```

- In the piece of code above, those **3.1415927**, **2.5**, **4.0**, and **“Area of a quarter pie is ”** are **literals** (or literal constants). They have fixed values and inserted directly into the source code.
- Those *, /, << are **operators**. Operators are typically expressed by a symbol or pair of symbols in C/C++ language:
  - Standard math operators are obvious: addition (+), subtraction (-), multiplication (*), and division (/); assignment (=) is an operator, too.
  - Some operators use more than one symbol, such as the equality operator (==). There are also a number of operators in words (e.g. **new** and **delete**).

EXPRESSIONS

- An **expression** is a combination of literals, variables, operators, and explicit function calls that produce a single output value.
- When an expression is executed, each of the terms in the expression is evaluated until a single value remains (called **evaluation**). That single value is the *result of the expression*.

`2`	**2** is a literal that evaluates to value 2
`"Hello world!"`	**"Hello world!"** is a literal that evaluates to text "Hello world!"
`x`	**x** is a variable that evaluates to the value of x
`2 + 3`	**2 + 3** uses operator + to evaluate to value 5
`y = sin(3.14)`	**3.14** passed to function **sin()**, get evaluated, and pass the value to y
`std::cout << x`	**x** evaluates to the value of x, which is then printed to the console

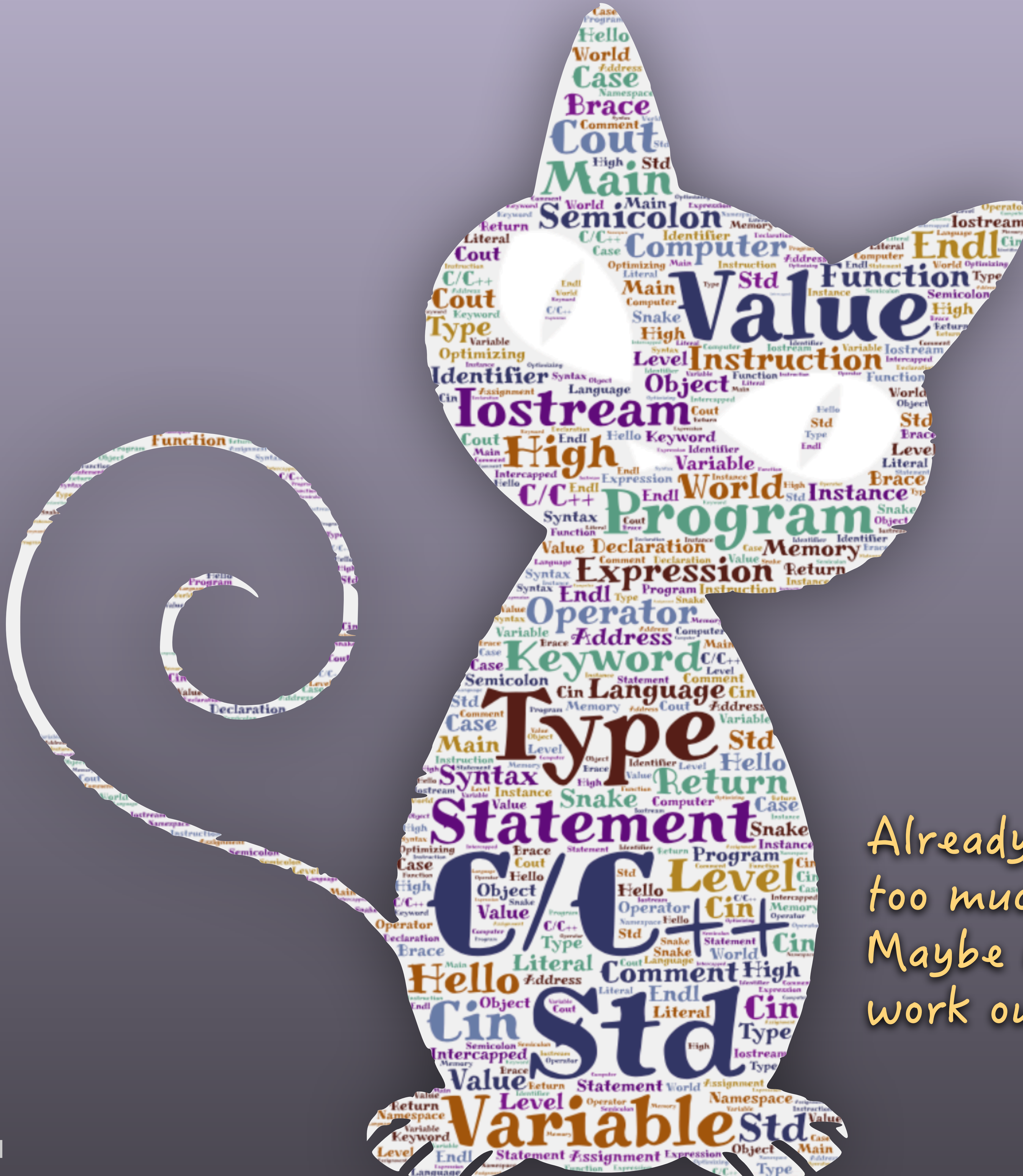
- Literals and variables evaluate to their own values. Function calls evaluate the value return from the functions. Operators combine multiple values together to produce a new value.

EXPRESSION STATEMENTS

- Expressions are always evaluated as part of statements, and cannot be compiled by themselves. Try to compile the expression **x = 5**, your compiler would complain (*missing semicolon*).
- Any expression can be converted into an equivalent statement (=**expression statement**) by adding a **semicolon** afterwards. When the statement is executed, the expression will be evaluated (and the result of the expression is discarded, compiler may raise a warning).

```
int x;
x = 5; ← an expression statement
x - 2; ← an expression statement (but useless, as the value discarded)
int y ((x=6)*7); ← expression (x=6)*7 set to y initialization
std::cout << "x,y = " << x << "," << y << “\n";
```

What's your evaluation of x,y at the end?

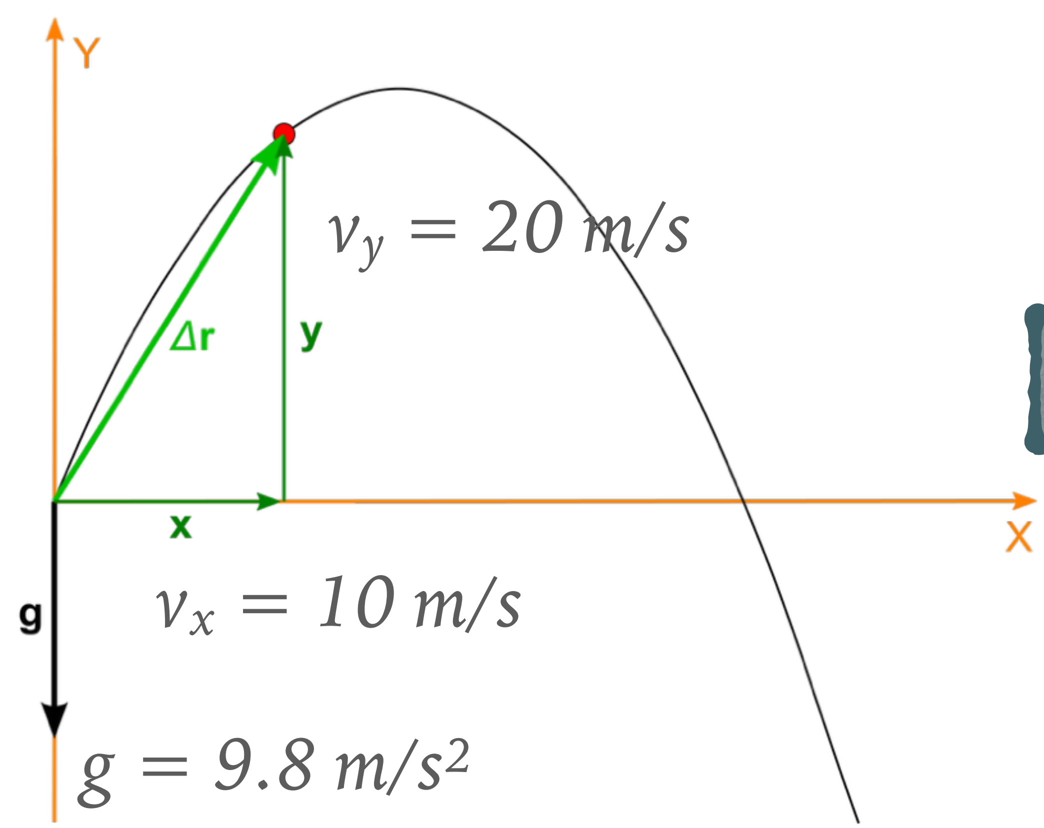


Already have
too much terminology?
Maybe it is good to
work out a program now!

PUT THINGS ALL TOGETHER

- We have introduced a lot of terminology and concepts up to now. Let's integrating the knowledge into our first "meaningful" program — or, by adding a little bit of physics taste?

- Suppose we would like to design a program to calculate the **trajectory of a projectile motion** at a given time, e.g.



keyboard input as time t

↓

Print location (x,y) on screen

$$\begin{cases} x = v_x t \\ y = v_y t - \dfrac{1}{2} g t^2 \end{cases}$$

CONSTRUCT YOUR PROGRAM STEP-BY-STEP

- Although the problem to be solve is still very simple and straightforward, but it is still nice to practice the construction of the program in steps.
  - It is not a good idea to write the whole program all at once, as it is very easy to be overwhelmed by a lot of errors at the first compile.
  - A better strategy is to start with a skeleton, and add one piece at a time, make sure it compiles, and test it. Only move to the next piece when the previous piece of code works properly.
- So let's start with the following "doing nothing" code:

```
int main()
{
    return 0;
}
```

trajectory.cc

Please consider to typing the program line-by-line (not copy-and-paste), if you are beginners.

CONSTRUCT YOUR PROGRAM STEP-BY-STEP (II)

- The plan is to have the input time from the users, in order to do so we need to include **iostream** header.
- Then we can initiate a double **target_time**, and obtain its value from **std:cin**.
- In order to ensure if the target_time stores exactly the number we input, let's first print out the value directly with **std::cout**.

```
$ g++ -std=c++11 trajectory.cc
$ ./a.out
Please input time t: 2.34
Time t = 2.34  ← output = input, OK!
```

```
#include <iostream>

int main()
{
    double t(0.);
    std::cout << "Please input time t: ";
    std::cin >> t;
    std::cout << "Time t = " << t << “\n";
   return 0;
}
```

trajectory.cc

CONSTRUCT YOUR PROGRAM STEP-BY-STEP (III)

- Once we obtain the target_time, it is mostly straightforward to add the calculation for the position (x,y) according to the equations given earlier:

```
$ g++ -std=c++11 trajectory.cc
$ ./a.out
Please input time t: 2.34
Time t = 2.34
Position x = 23.4
Position y = 19.9696
```

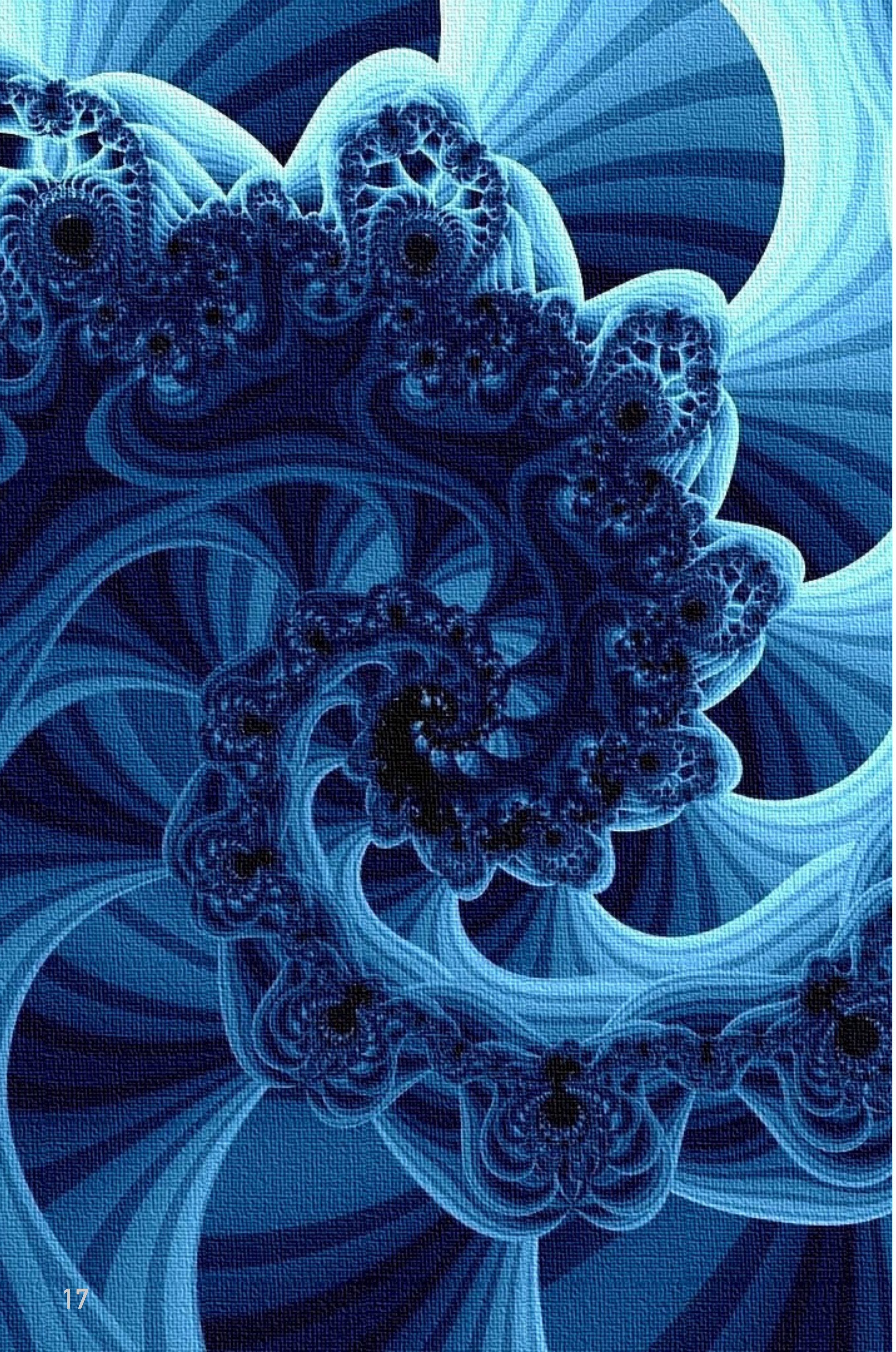
← still need to verify if the results are correct or not!

```
#include <iostream>

int main()
{
    double t(0.);
    std::cout << "Please input time t: ";
    std::cin >> t;
    std::cout << "Time t = " << t << "\n";

    double x(10.*t);
    double y(20.*t - 0.5*9.8*t*t);
    std::cout << "Position x = " << x << "\n";
    std::cout << "Position y = " << y << “\n";
    return 0;
}
```

trajectory.cc

TIPS TOWARD "GOOD" PROGRAMS

- **Surely the #1 goal of programming is to have a working program.**
- But generally it is not possible to know how the program should be best constructed, at the initial development phase.
  - It is pretty common when we try to make our programs work, things like error handling and comments might be skipped as a "short-cut", in particular there can be a lot of useless code leftover when we are testing different approaches.
  - Once your program is "working", the coding job is not really finished, unless the program is going to be used only once.
  - A good practice is to **cleanup** your code by removing/combining temporary/debugging/redundant codes, adding comments, handling error cases, formatting your code, and ensuring best practices are followed, ie. **maintainability** of the code is equally important as the performance.



MODULE SUMMARY

- In this module we have revisit the variables definition, restrictions to the variable naming (including keywords, etc), as well as C/C++ operators. Our very first introduction to the language itself is completed.
- For the next module, we will start to discuss the functions (*very important feature!*), how to arrange the arguments, and in particular the local "scope".