**C++ Language (Data structure using c++)**

still confused: name visibility, pointers, dynamic memory, special member, polymorphism

*Introduction*

* Computable- able to solve/have a solution or algorithm
* Abstraction- how operating system works/control low-level details that user assumes
* Interface- logical perspective/outside (driver) -> physical perspective/inside (mechanic)
* Procedural abstraction- know function being called but not know how function work or calculated
* black box- someone already made solution and we use it by importing
* data items as binary -> data types to give meaning
* use abstraction to focus in on big picture
* abstract data type/data structure- how we view data and operations that are allowed without regard to how they will be implemented -> what data is representing not how will be constructed -> encapsulation and information hiding
* implementation independent -> many ways to implement but same goal/purpose
* find solution and decide whether good or bad to solve problem
* data items = objects = instance of class

*files*

* main.cpp
* source.cpp
  + the function and classes itself
* header.h
  + declaration of functions/calling

*beginning*

* **using namespace std**;
* **#include <iostream>**
* **#include <sstream>**
* **#include <string>**

*variable and types*

* alignas, alignof, and, and\_eq, asm, auto, bitand, bitor, bool, break, case, catch, char, char16\_t, char32\_t, class, compl, const, constexpr, const\_cast, continue, decltype, default, delete, do, double, dynamic\_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, noexcept, not, not\_eq, nullptr, operator, or, or\_eq, private, protected, public, register, reinterpret\_cast, return, short, signed, sizeof, static, static\_assert, static\_cast, struct, switch, template, this, thread\_local, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar\_t, while, xor, xor\_eq
  + int
  + string
    - char = element in a string/1 letter -> for ranges
  + float
    - double
  + bool
    - true
    - false
  + exponents using **pow()** from **#include <cmath>** library
* every variable must be declared with its type beforehand
  + int a, b, c;
    - int a;
    - int b;
    - int c;
  + int x = 0;
  + int x (0);
  + int x {0};
* can use **auto** to assign a type automatically. the type of variable will follow the type of value
  + int x = 0;
  + auto y = x; -> int y = x;
* **decltype**
  + int x = 0;
  + decltype(x) y; -> int y;
* converting types -> enclose the type with **()**
  + float f = 3.14;
    - int i = (int) f;
    - int i = int (f);
    - int i = f
    - int i;
      * i = f;
      * i = (int) f;
* return size in bytes
  + sizeof (char);
* decimal
  + cout << dec << variable << endl;
* octal
  + cout << oct << variable << endl;
* hexadecimal
  + cout << hex << variable << endl;

|  |  |  |  |
| --- | --- | --- | --- |
| **Escape code** | **Description** | |  |
| \n | newline | |  |
| \r | carriage return | |  |
| \t | tab | |  |
| \v | vertical tab | |  |
| \b | backspace | |  |
| \f | form feed (page feed) | |  |
| \a | alert (beep) | |  |
| \' | single quote (') | |  |
| \" | double quote (") | |  |
| \? | question mark (?) | |  |
| \\ | backslash (\) | |  |
|  | |
| **Group** | | **Type names\*** | | **Notes on size / precision** |
| Character types | | **char** | | Exactly one byte in size. At least 8 bits. |
| **char16\_t** | | Not smaller than char. At least 16 bits. |
| **char32\_t** | | Not smaller than char16\_t. At least 32 bits. |
| **wchar\_t** | | Can represent the largest supported character set. |
| Integer types (signed) | | **signed char** | | Same size as char. At least 8 bits. |
| *signed* **short** *int* | | Not smaller than char. At least 16 bits. |
| *signed* **int** | | Not smaller than short. At least 16 bits. |
| *signed* **long** *int* | | Not smaller than int. At least 32 bits. |
| *signed* **long long** *int* | | Not smaller than long. At least 64 bits. |
| Integer types (unsigned) | | **unsigned char** | | (same size as their signed counterparts) |
| **unsigned short** *int* | |
| **unsigned** *int* | |
| **unsigned long** *int* | |
| **unsigned long long** *int* | |
| Floating-point types | | **float** | |  |
| **double** | | Precision not less than float |
| **long double** | | Precision not less than double |
| Boolean type | | **bool** | |  |
| Void type | | **void** | | no storage |
| Null pointer | | **decltype(nullptr)** | |  |

*strings*

* **#include <string>**
* must declare its type beforehand: **string**
  + string title = “This is a string”;
  + string title (“This is a string”);
  + string title {“This is a string”};
* can use same variable to print out a different value using **endl** or **“/n”**
  + cout << title << endl
* double quotes = strings

*character sequence*

* single quotes = single character within a string/char
* char = element in a string/1 letter
* char name **[**x**];**
  + store up to x element of type char
  + if with char, then must use element to have a storage
* type name **[**element**]** **= “**string**”;**
  + element can be left out
* at end of array, \0, indicate end of string
* can be used as variable for cin and cout
* to print string to c-string
  + name.**c\_str()**

*constant*

|  |  |
| --- | --- |
| **Suffix** | **Type modifier** |
| u *or* U | unsigned |
| l *or* L | long |
| ll *or* LL | long long |

|  |  |
| --- | --- |
| **Suffix** | **Type** |
| f *or* F | float |
| l *or* L | long double |

|  |  |
| --- | --- |
| **Escape code** | **Description** |
| \n | newline |
| \r | carriage return |
| \t | tab |
| \v | vertical tab |
| \b | backspace |
| \f | form feed (page feed) |
| \a | alert (beep) |
| \' | single quote (') |
| \" | double quote (") |
| \? | question mark (?) |
| \\ | backslash (\) |

*bit-wise operators*

|  |  |  |
| --- | --- | --- |
| **operator** | **asm equivalent** | **description** |
| & | AND | Bitwise AND |
| | | OR | Bitwise inclusive OR |
| ^ | XOR | Bitwise exclusive OR |
| ~ | NOT | Unary complement (bit inversion) |
| << | SHL | Shift bits left |
| >> | SHR | Shift bits right |

*basic input/output*

|  |  |
| --- | --- |
| **stream** | **description** |
| cin | standard input stream |
| cout | standard output stream |
| cerr | standard error (output) stream |
| clog | standard logging (output) stream |

* cout = screen/prints out
  + using <<
  + to add line breaks
    - \n
    - cout << “string” << endl
    - cout << “string ” << “string”
* cin = keyboard/user types in
  + using >>
  + examples
    - int age;
    - cin >> age;
  + example to get entire line for a string using getline
    - string name;
    - **getline** (cin, name);
* converting string to integer using **stringstream**
  + string name = “123”;
  + int age;
  + stringstream(name) >> age;

*statement*

* all statement must end with a period, use **;**
* compound statement using **{}** 
  + **{** statement1**;** statement2**;** statement3**; }**
* jump statement use with if statements
  + **break;**
    - leaves the loop
  + **continue;**
    - skips a specific part of the iteration
  + **goto** label
    - jump to another point in program
    - label made with **:**
  + **switch (**expression**)** statement

*if and else*

* conditions are using **<**, **>, !=, ==**, **<=**, **>=**, **&&** (and), **||** (or), **!** (not)
  + true=1
  + false=0
* **if (**condition**)**
* when there are more than 1 statement, use {}
* **else if** **(**condition**)**
  + is an elif

*loops and iteration*

* types of loops -> when there are more than 1 statement use {}
  + **while (**expression**)** statement
    - iteration not known
    - must initialize first -> int x=0;
    - expression evaluated before statement
  + **for (**initialize**;** condition**;** increment**)** statement
    - iteration known
  + **for (**declaration **:** range**)** statement
    - only used with for loops
    - declaration
      * char c
      * auto c
  + **do** statement **while** **(**condition**);**
    - condition evaluated after execution of statement
    - statement will execute once even if condition not met
    - basically: if while condition True, repeat back to do statement. if not, exits loop
* increments
  + x = x + 1;
  + x += 1;
  + ++x;

*commenting*

* **//** = line comments
* **/\* \*/** = docstring/block comment
* **#** = directives read and interpreted by preprocessor

*function*

* format
  + type name **()** **{**statements return**}**
    - return also use ;
  + function call is a statement
* types
  + int
  + string
    - use of **const** for constant -> const string a
      * it is like passing a value of larger element -> forbidden to modify
* main is a special name. first function called when program runs
  + return value is -> **return 0** -> to terminate program
* **void** = a type for function with no type
  + printing a message
* use **&** in parameter after the type to pass the variable itself not the copy
  + void duplicate (int&a)
* **inline** function -> inform compiler inline expansion preferred over usual function
  + inline string concatenate (const string& a, const string& b)
* default values in function -> set a value in parameter
* declaring functions -> so that you don’t have to write non-main functions before the main functions
  + type name **(**type variable**);**
* recursively is property that function have to be called by themselves. useful for sorting elements, calculating factorial numbers

#include <iostream>

using namespace std;

long factorial (long a)

{

if (a > 1)

return (a \* factorial (a-1));

else

return 1;

}

int main ()

{

long number = 9;

cout << number << "! = " << factorial (number);

return 0;

}

*overloaded function*

* two function can have same name, but have different type of function and parameter to make it two different function
* function templates
  + name <type-arguments> (function-argument)
* non-type template argument

*name visibility*

* global scope = outside any block
* block scope = inside a block
  + local variables are variables only for inside scope

*namespaces*

* namespaces -> group named entities that otherwise would have global scope into narrower scope
  + **namespace** identifier {type variable;}
    - **namespace** identifer {type variable() {return;}}
  + can be accessed through
    - identifier::variable
* **using** = introduce name into current declarative region so that we don’t need to qualify the name.
  + **using** identifier::variable
  + can also be used to introduce entire namespace
    - **using namespace** identifier
* namespaces can be renamed
  + **namespace** new\_name = current\_name;
* static storage = storage for global or namespace allocated for entire duration of program. different with storage for local variable
  + static storage = global variable. that are not explicitly initialized are automatically initialized to zero
  + automatic storage = local variable. that are not explicitly initialized are left uninitialized -> random value

*arrays*

* array is a series of elements of same type placed in adjacent memory location -> indexes
  + must be declared just like variables:
    - type name **[**element**];**
  + element is the total variable
  + all arrays outside a function always initialized, if not specified -> default to 0
* initializing arrays
  + arrays in local variable are left uninitialized -> elements are not set to particular index, undetermined
  + but can be initialized to specific value when declared:
    - type name **[**element**] =** **{**value**};**
    - type name **[**element**] {**value**};** 
      * value is the values of the variables, using commas
      * element can be left out -> compiler will calculate itself
  + if there are more elements than values, then the value is set to default 0
* accessing value of an array
  + to store a value 1 into an array
    - name **[**index**]** = 1**;**
    - index can also be a character/string of the value
  + to output a value to the variable x
    - x = name **[**index**];**
* multidimensional array
  + arrays of arrays
  + type name **[**element**][**element**];**
    - [vertical][horizontal] or [y][x]
* passing array to a function
  + int arr[x]
  + array as parameter
    - void function(int arr[])
  + array as argument
    - function(arr)
* library arrays
  + **#include <array>**
  + type name [element] = {value};
  + **array<**type**,** element**>** name **{**value**};**
* vectors -> function can also apply to string
  + dynamically allocated array to store element -> access using indexes
  + **#include <vector>**
  + appends item to the far end of vector
    - name.**push\_back(**item**)**
  + deletes last item from far end from vector
    - name.**pop\_back()**
  + insert item at index i
    - name.**insert(**i, item**)**
  + erase element from index i
    - name.**erase(**i**)**
  + change in capacity to amount
    - name.**reserve(**amount**)**

*unordered sets*

* **#include <unordered\_set>**
* immutable, can be insert or removed
* **set<**type**>** name = **{**values**}**
* add item to set
  + name.**insert(**item**)**
* remove item to set
  + name.**erase(**item**)**
* remove all element from set
  + name.**clear()**

*pointers*

* variables are access through names, instead of location
* reference operator/address of operators = points to the address/location of variable -> use **&**
  + name **= &**variable**;**
  + assign address/location not content to name -> location is the indexes
* deference operator = value pointed to the address -> use **\***
  + name = **\***variable**;**
  + assigns the content of the address it was pointed to previously
  + name **=** variable**[**index**];**
  + name **= \*(**variable **+** x**);**
* declaring pointers
  + since pointers can refer to value it points to, different property when points to a char then when int or float = once deference, type needs to be known -> declaration of pointer include type
  + type **\*** name**;**
    - type is the type of the data pointer points to
* pointers and arrays
* pointer initialization
* pointer initialization
* pointer arithmetic’s
  + adding one to pointer, pointer made to point to the following element of same type = size in byte of type it points to is added to pointer
  + \*(p++)
    - increment pointer, but dereference incremented
  + \*(++p)
    - increment pointer and dereference incremented
  + ++(\*p)
    - dereference and increment
  + (\*p)++
    - dereference and post increment value
  + ++ has higher priority over \*, but if used postfix, value is the one that was incremented
* pointers and string literal
  + to put string/char into pointers -> use **const** in front of type
* pointers to pointers
  + pointers that points to pointers -> points to data or other pointers -> use **\*** for level of indirection
* void pointers
  + points to a value with no type
  + data pointed by void cannot be directly deference -> must transform to some other pointer type that points to concrete data type before being deference
* invalid pointers and null pointers
  + void pointers = point to somewhere without type
  + null pointers = point to nowhere
  + need to point nowhere, not just invalid:
    - **= 0;**
    - **= nullptr;**
    - **= NULL;**
* pointers to function
  + passing a function as an argument to another function
    - **(\***name**)**

*dynamic memory*

* new and new[]
  + **new** type **[**element**];**
  + when memory allocation fails, instead of terminating program, pointer return is null pointer = nullptr
    - **new (nothrow)** type **[**element**];**
* delete and delete[]
  + **delete[]** name**;**

*data structure*

* group of data element grouped together under one name
* members = data elements -> can have different type and length
  + **struct** name **{**type name; type name; ...**}** object1, object2, ...**;**
    - object can be accessed by:
      * object.name
  + **struct** product **{**type name; type name; ...**}** ;
    - product apple, banana**;**
* pointers to structure
  + structure\_name variable;
  + structure\_name \* variable1
    - variable1 = &variable
  + arrow operator (->)
    - used exclusive with pointers to objects that have members. to access member of an object directly from address
      * variable1 -> name\_inside\_structure
      * (\*variable1).name\_inside\_structure

|  |  |  |
| --- | --- | --- |
| **Expression** | **What is evaluated** | **Equivalent** |
| a.b | Member b of object a |  |
| a->b | Member b of object pointed to by a | (\*a).b |
| \*a.b | Value pointed to by member b of object a | \*(a.b) |

* nesting structure
  + element of structure is another structure

*other data type*

* type aliases -> typedef / using
  + a different name which a type can be identified
  + **typedef** type name
  + **using** name = type
* union
  + allow one portion of memory to be access as different data types
  + similar structure to data structure, except use **union** and object only one
* anonymous union
  + when unions are member of class/structure, can be declare with no name
    - no need name or object
* enumerated type
  + type defined with set of custom identifiers as possible values
    - **enum** name **{**value, value1, ...**}** object**;**
      * values are converted to integer
        + value has value 0, value 1 has value 1, and so on
        + if value = 1, then value1 is 2, and so on
* enumerated type with enum class
  + can create enum type that does not convert to integer or has type integer but the enum type itself
    - **enum class** name {value, value1, ...} object**;**

*classes*

* format
  + **class** name **{**access: member; access: member1; ... {return;}**}** object**;**
    - member = type name (parameter)
    - return is under one member: indented or same line
    - to call when there is no object:
      * name object\_name;
    - access:
      * private = accessible only within other member of same class
        + by default private
      * protected = accessible from other member of same class
      * public = accessible from anywhere
    - call using object name
    - to access private, use namespaces
      * name::name (constructor) {private\_member}
* constructor
  + name::name **(**parameter**) {**instances;**}**
* overloading constructor
  + to call default no need to use ()
* uniform initialization
  + name object = value;
  + name object {value, value1, ...}
* member initialization in constructor
  + name::name (constructor) : private\_members { }
* pointer to classes
  + class\_name \* name;
  + member of object can be access using (->)
* classes can be defined using **struct** or **union**
  + will have public access as default
* overloading operators

|  |
| --- |
| **Overloadable operators** |
| + - \* / = < > += -= \*= /= << >>  <<= >>= == != <= >= ++ -- % & ^ ! |  ~ &= ^= |= && || %= [] () , ->\* -> new  delete new[] delete[] |

* + type **operator** operator\_sign (parameter) {body}
    - using this, can call function implicitly using expression or explicitly using operator name

|  |  |  |  |
| --- | --- | --- | --- |
| **Expression** | **Operator** | **Member function** | **Non-member function** |
| @a | + - \* & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A,int) |
| a@b | + - \* / % ^ & | < > == != <= >= << >> && || , | A::operator@(B) | operator@(A,B) |
| a@b | = += -= \*= /= %= ^= &= |= <<= >>= [] | A::operator@(B) | - |
| a(b,c...) | () | A::operator()(B,C...) | - |
| a->b | -> | A::operator->() | - |
| (TYPE) a | TYPE | A::operator TYPE() | - |

* keyword **this** = use within class member function to refer object itself
  + can check if parameter pass to member function is object itself
* static member = class variable because only one common variable for object of that same class
  + use **static** before type
  + initialized outside of class, can be referred as member of object of class or by class name
    - static\_type class\_name::n=0;
* const member function = when object of class as a const object
  + **const** class\_name name
  + member of const object can only be called if member also const
    - type name const {return}
* class template
  + **template <class** T**>**
  + **class** name **{};**
  + template specialization
    - **template <>**
    - **class** name **<**type**>** {};

*special member*

* implicitly define as member of class
* default constructor -> C::C();
  + object of class declared but not initialized with any argument
* destructor -> C::~C();
  + takes no argument and return nothing
* copy constructor -> C::C (const C&);
  + object passed a named object of its own type as argument -> construct copy
* copy assignment -> C& operator= (const C&);
  + can be copied on assignment operation
* move constructor -> C::C (C&&);
  + transferred not copy
* move assignment -> C& operator= (constC&&);

|  |  |  |
| --- | --- | --- |
| **Member function** | **implicitly defined:** | **default definition:** |
| [Default constructor](http://www.cplusplus.com/doc/tutorial/classes2/#default_constructor) | if no other constructors | does nothing |
| [Destructor](http://www.cplusplus.com/doc/tutorial/classes2/#destructor) | if no destructor | does nothing |
| [Copy constructor](http://www.cplusplus.com/doc/tutorial/classes2/#copy_constructor) | if no move constructor and no move assignment | copies all members |
| [Copy assignment](http://www.cplusplus.com/doc/tutorial/classes2/#copy_assignment) | if no move constructor and no move assignment | copies all members |
| [Move constructor](http://www.cplusplus.com/doc/tutorial/classes2/#move) | if no destructor, no copy constructor and no copy nor move assignment | moves all members |
| [Move assignment](http://www.cplusplus.com/doc/tutorial/classes2/#move) | if no destructor, no copy constructor and no copy nor move assignment | moves all members |

*friendship and inheritance*

* friend function -> use keyword **friend**
  + another function can access the private member of a class if declare with friend within the class
    - **friend** name;
* friend classes
  + another class can access the private member of a class
    - **friend class** name;
* inheritance between classes
  + base and derived class
    - **class** derived\_name**:** **public** base\_name {statements};
      * can also use public, protected, private
        + class = private
        + struct = public
  + inherits all except constructor, destructor, friend, private member, assignment operator member
* inheritance from base class
  + calling base constructor
    - specific:
      * derived\_name (parameters) : base\_name (parameter) {statement}
    - default:
      * derived\_name (parameters)
* multiple inheritance
  + same with inheriting one but just add more separated by commas

*polymorphism*

* pointers to base class
* virtual member = member function that can be redefined in derived class while preserving calling properties through reference
  + function to become virtual using keyword **virtual** before writing type
  + inherits a virtual function is polymorphic class
    - **virtual** type name () {return;}
* abstract base class -> only use as base class so can have virtual member without definition
  + virtual type name () =0;
  + can be used to create pointers

*type conversion*

* when converting, automatically converts when already set new type (implicit), but sometimes need explicit
* implicit conversion automatically perform when value copied to a compatible type
  + numerical types, bool, pointer conversion
  + promotion = unspecific to more specific
* implicit conversion with class, by three member class
  + single-argument constructor = conversion from type to initialize object
  + assignment operator = conversion from type to assignment
  + type-cast operator = conversion to type
    - use operator keyword
      * **keyword** destination **()** **{return** destination**;}**
* keyword explicit
  + to prevent two functions from being called -> use keyword **explicit** to prevent affected constructor
* type casting
  + two main syntaxes
    - (new\_type) variable;
    - type (variable);
  + control conversion between classes
    - casting <new\_type> (expression)
      * casting:
        + dynamic\_cast = only used with pointer and reference to new class (or with void\*). ensure result of conversion points to valid object of destination pointer type. upcast and downcast
        + reinterpret\_cast = pointer to pointer, even unrelated classes. to/from integer
        + static\_cast = pointer to related classes, upcast and downcast.
        + const\_cast = manipulate constness of object pointed by pointer, set or removed.
* typeid = check type of expression
  + **#include <typeinfo>**
  + **typeid** (expression)

*exception*

* react to runtime error by transferring control to special function = handler
  + portion of code placed under exception inspection -> enclosed in try-block
    - **throw** keyword inside **try** block
    - exception handler declared with **catch (**parameter**)** immediately after try block
* exception specification
  + dynamic exception specification
    - **double** function\_name (type parameter) **throw** (throw\_type);
    - throws exception of some type other than the throw type -> call std::unexpected instead std::terminate
* standard exception
  + base class to declare object be thrown as exception
    - **#include <exception>**
    - std::exception
    - member function -> **what()**

*preprocessor directives*

* lines using **#**, only can be extended using **\**
* define and undefine
  + **#define** identifier replacement
    - replaces identifier with replacement
    - replacing strings using #, e.g
      * #define str(x) #x = “x”
      * ## => cout
  + **#undef** identifier
    - stops replacing
* conditional inclusion = allow include/discard part of code of program if certain condition met -> **#ifdef, #ifndef, #if, #endif, #else, #elif**
  + #ifdef = section of program compiled if macro that is specified as parameter has been defined
  + #ifndef/#endif = compiled if identifier has not been defined
  + #if/#else/#elif = specify condition be met to be compiled
    - end with #endif
* line control = line number and file name when error
  + **#line** x “filename”
* error directive = aborts compilation process, generating compilation error
  + **#error**
* source file inclusion =
  + **#include** <header>
  + **#include** “file”
* pragma directive = specify diverse option to compiler
  + **#pragma**
* predefined macro names

*input/output with files*

* declaring file
  + classes variable;
    - classes
      * ofstream = write on files -> default is ios::out -> combines
      * ifstream = read from files -> default is ios::in -> combines
      * fstream = write and read -> default is ios::in | ios::out -> override
        + **#include <fstream>**
  + class variable (“file\_name”, mode);
    - variable.is\_open()
* opening a file
  + variable**.open** (“file\_name”, mode);
    - mode -> separate using |
      * ios::in = open for input operator
      * ios::out = open for output operator
      * ios::binary = open in binary mode
      * ios::ate = set initial position at end of file, if not set then at beginning of file
      * ios::app = output operator performed at end of file. appends to file
      * ios::trunc = if file open for output operation and already exist, previous content deleted and replaced by new one
* closing a file
  + variable**.close()**
* text files = ios::binary not included in opening mode
  + writing -> file\_name <<
  + reading
    - reads whole file use while loop and getline
      * getline(variable, name)
* checking state flags = check state of stream and return bool
  + **bad()** = true if reading/writing fail
  + **fail()** = true like bad() but also if format error appear. e.g return character when trying to read integer
  + **eof()** = true if file open for reading has reach end
  + **good()** = not opposite of bad()
  + **clear()** = reset state flag
* get and put stream positioning
  + ifstream/istream -> keep internal get position with location of element to be read in next input
  + ofstream/ostream -> keep internal put position with location where next element has to be written
  + fstream/iostream -> keep get and put position
  + **streampos begin, end;**
    - **tellg()** and **tellp()** -> get current get position (tellg) and put position (tellp)
    - **seekg(**position**);** and **seekp(**position**);** -> change location of get and put position
      * or (offset, direction);
      * ios::beg = offset counted from beginning of stream
      * ios::cur = offset counted from current position
      * ios::end = offset counted from end of stream
* binary files
* buffer and synchronization

**others**

* exit(1) //terminates unsuccessfully
* exit(0) //terminates successfully
* goto(name)
  + name:
* pretty print
  + #include <iomanip>
  + setw(x)
  + x is the number of width