1. **-Modularity:** break the solution into several modules. Each module can be used in many different applications; well-defined module can have different context; help us easily to understand; more adaptable to change.

-**Objected-oriented concepts:** we can solve problem to build the class of objects. These objects have **encapsulation** (data and data operation), **inheritance** (classes can inherit properties from other classes), and **polymorphism** (determine right operations at execution time).

-**Coupling (loosely):**  measure of dependence among modules. 1. More adaptable to change 2. Easier to understand. 3. Good reusability 3. Increase cohesion.

-**Cohesion (highly):** each module should be well-defined task. 1. Well-named 2 less affected by change 3 easy to reuse 3 easy to maintain.

-**Abstraction:** Separate the purpose of a module from its implement.

-**Data abstraction**: specify what operations do with the collection of data. It built a wall around a program’s data structure. -Controlling the interaction between program and its data structure.

2. -Modularity (break to modules) ---> Abstraction (specify module) ---> Data Abstraction (specify operations do on data) ---> **ADT** (.h file) ---> **Data Structure** (. cpp file implement ADT)

**ADT: collection of data and a set of operations on the data. (.h file)**

**Data Structure: implement the ADT by using Data Structure with a programming language. (Using programing language to store a collection of data)**

**Information hiding: some details should be private and hide within a module And they cannot be change by other modules.**

1. **Public:** member (data and operations) in public can be access by anyone.

**Private:** member (data and operations) in private can be access by member class and friend class.

**-Constructor:** allocate memory for new instance **and** initialize object’s data to specific value.

**When use:** When we declare an instance of class, the constructor is invoked.

**-Destructor:** destroy an instance of class when the object no longer use.

**When use:** when we don’t need the instance of the class, invoke destructor.

1. Question?
2. **Implement of method of class(.cpp file):** syntax: 1.contain header file #include “PlainBox.h” 2. Method defined here are part of *PlainBox* class. *PlainBox::*

**How to Manipulate the private** variables of the class: We have **setItem()** and **getItem()** to access private data field.

1. Question?
2. **Template: it allow us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.**

Template<typename T>

PlainBox<typename>::PlainBoc(){

}

Header file 后面要加 #include “PlainBox.cpp”

Interlude 2 & chap. 2

1. **Local variable**: define and declared in functions allocated in runtime stack( automatic storage class)

**When allocate**: each time when the function is called, it will automatically create on run-time stack.

**When deallocated**: When the function end, the record is destroyed and frees the memory. At this time, function’s local variables and their values are no longer usable.

**Dynamically allocated variables: in heap( free store) by using *new* operator*.***

**When allocate: we allocate memory for variable on free store by using *new* operator.**

**When deallocate: when the variable is no longer needed, we should deallocate it. ( if we forget to deallocate the dynamically allocated variable that we no longer need, it will result in memory leak)**

1. **Memory leak: happen when dynamically allocated memory is not return to the free storage (forget deallocate) when it is no longer used or referenced.**

**Why** avoid: it will result in program consuming (wasting) more memory that it needs OR lead to runtime program crash when subsequent (next next…) memory allocations fail.

**How to avoid**: do ***delete***on any dynamically allocated memory once we finish using it*.* Apply *delete* operator to pointer that reference the dynamically allocated memory that I want to return to free store.

1. **Dangling pointer: is a pointer variable that that a *delete* have been issued on it or no longer reference to a valid object.**

**What cause:** pointer happen when an object is deleted or deallocted, without modify value of the pointer, so that the pointer still point to the memory location of deallocated memory.

**Why avoid:** when we dereference( pointer and data the pointer points) the dangling pointer, and the memory now contain different data, a corruption of unrelated data may result, bug difficult to find.

SomeType\* x = new SomeType;

SomeType\* y = x;

…

delete x;// delete must uses in dynamically allocated memory.

…

y->someMethod(…);//Nooooooooooooo!!!! Dangling pointer!

**How to avoid:** 1. **Set** pointer variable to ***nullptr*** when no longer use it*.*

2. **Test** whether the pointer variable contain ***nullptr*** when we call this method.

**?????**

1. Employee &fred ( **reference variable** defined by &)

Employee \*sally (**pointer variable** defined by \*)

**Difference:** reference variable must be bound to a specific object when declaration. Once bound, the reference variable cannot refer to a different object**.**

Employee chris;

Employee& empRef = chris;

**Visualize them in memory picture**:

SomeType x;

SomeType\* p = &x;

SomeType& r = x;

**Deference (\*ptr = 7.7):** means a pointer and accessing the data(**Value**) which it points to. )using the value!!!

Reference (ptr = & b): using the **address**.

**Chapter 5**

A recursive algorithm must exhibit each of the following characteristics:

1. The algorithm computes a solution to a general problem by combining the solutions of one or more identical, but "smaller" problems.
2. The algorithm invokes itself ("recurses") to compute the solutions to these problems.
3. There exist one or more *base cases* for which solutions are either known or can be directly computed without further recursion.
4. It is possible to prove that base cases will always be reached for any arbitrary invocation of the algorithm.
5. **Base case**: the base case of a function is the problem that we know the answer; it can be solved without any more recursion calls. And the base case can stop the recursion for a big problem

A grammar is a 4-tuple (N, T, Π, S) where:

* N is a set of *non-terminal symbols*. (That is, they are not elements of the language itself.)
* T is a set of *terminal symbols*. (These comprise the "alphabet" of the language. They are symbols that actually appear in strings belonging to the language.)
* Π is a set of *production rules* (also called *rewrite rules*) that describe all ways that symbols in N can be replaced by combinations of symbols drawn from the N and T sets. Along with the symbols in N, the production rules specify the structure of the language.
* S is an element of N called the *start symbol*. All valid strings in the language can be generated starting from S and using a series of the rewrite rules.

1 N = { <pdn>, <dd> }

1. T = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
2. Π = {  
   <pdn> → <dd>  
   <pdn> → <pdn> <dd>  
   <dd> → 0  
   <dd> → 1  
   <dd> → 2  
   <dd> → 3  
   <dd> → 4  
   <dd> → 5  
   <dd> → 6  
   <dd> → 7  
   <dd> → 8  
   <dd> → 9  
   }
3. S = <pdn>