Cpp 基礎知識

Polymorphism (分為**靜態**(編譯時期 early binding)、**動態**兩種(執行時期 late binding))

|  |
| --- |
| * 編譯時期多型(靜態多型)   - function overloading: functions that share the same name (call function with parameter number and type) (重載可以使用一個函式名稱來執行不同的實作)  - operator overloading 運算子多載 (其實同function overloading)   * 執行時期多型 (動態多型)   - 如何呼叫”不同物件”的”相同名稱的成員函數” 🡺 利用 inheritance 與 Polymorphism (這時就會扯到virtual function) |

Virtual Function (“執行時期”的多型)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| * Override methods in subclass * 一個based class的物件指標，可以用來指向其derived class而不會發生錯誤。 * 例如若基底類別是Foo1，而衍生類別是Foo2，則 下面這個指定是可以接受的：   Foo1 \*fptr;  Foo2 f2;  fptr = &f2;   * 多型與動態繫結的基礎從這開始，它們只有在使用指標或參考時才得以發揮它們的特性，然而由於fptr仍是Foo1類型的指標，它只能存取Foo1中有定義的成員，目前來說也只能操作Foo1中的成員。 * 注意將derived class型態的指標指向based class的物件基本是不可行的，derived class的指標並不能存取based class的成員。 * 虛擬函式是一種成員函式，它在基底類別中使用關鍵字"virtual"宣告（定義），並在衍生類別中重新定義虛擬函式，這將成員函式的操作決議推遲至執行時期再決定。 * 虛擬函式可以實現執行時期的「多型」，也就是「一個介面，多種函式」，一個含有虛擬函式的類別被稱為「多型的類別」（Polymorphic class），當一個基底類別型態的指標指向一個含有虛擬函式的衍生類別，您就可以使用這個指標來存取衍生類別中的虛擬函式，下面這個例子是個簡單的示範：   **class** Foo1 {  **public**:   **virtual** **void** show() { // 虛擬函式   cout << "Foo1's show" << endl;   }  };  **class** Foo2 : **public** Foo1 {  **public**:   **virtual** **void** show() { // 虛擬函式   cout << "Foo2's show" << endl;   }  };   **void** showFooByPtr(Foo1 \*foo) {foo->show();} **void** showFooByRef(Foo1 &foo) {foo.show();}  **int** main() {   Foo1 f1;   Foo2 f2;    // 動態繫結   showFooByPtr(&f1);   showFooByPtr(&f2);    // 動態繫結   showFooByRef(f1);   showFooByRef(f2);   // 靜態繫結   f1.show();   f2.show();  }  執行結果：  Foo1's show Foo2's show  Foo1's show Foo2's show  Foo1's show Foo2's show  Pure virtual function   * C++提供一種語法定義Pure virtual function，指明某個函式只是提供一個介面，要求繼承的子類別必須重新定義該函式，定義純虛擬函式除了使用關鍵字"virtual"之外，要在函式定義之後緊跟著'= 0'，例如：   class Some { public:     virtual void someFunction() = 0; };   * 一個類別中如果含有純虛擬函式，則該類別為一（Abstract class），該類別只能被繼承，而不能用來直接生成實例，如果試圖使用一個抽象類別來生成實例，則會發生編譯錯誤。   Ex:  class ConcreteCircle { public:     void radius(double radius) { \_radius = radius; }     double radius() { return \_radius; }     **void render() { cout << "畫一個半徑 " << \_radius << " 的實心圓" << endl; }** private:     double \_radius; };    class HollowCircle { public:    void radius(double radius) { \_radius = radius; }     double radius() { return \_radius; }     **void render() { cout << "畫一個半徑 " << \_radius << " 的空心圓" << endl; }** private:     double \_radius; };   * 顯然的，這兩個類別除了render()方法的實作內容不同之外，其它的定義是一樣的，而且這兩個類別所定義的顯然都是「圓」的一種類型，您可以定義一個 抽象的AbstractCircle類別，將ConcreteCircle與HollowCircle中相同的行為與定義提取至抽象類別中：   class AbstractCircle {  public:  void radius(double radius) { \_radius = radius; }  double radius() { return \_radius; }  virtual void render() = 0; // 宣告虛擬函式  protected:  double \_radius;  };   * 注意到在類別宣告了虛擬函式render()，所以AbstractCircle是個抽象類別，它只能被繼承，繼承了AbstractCircle的類別 必須實作render()函式，接著您可以讓ConcreteCircle與HollowCircle類別繼承AbstractCircle方法並實作 render()函式：   class HollowCircle : public AbstractCircle {  public:  void render() { cout << "畫一個半徑 " << \_radius << " 的空心圓" << endl; }  };  class ConcreteCircle : public AbstractCircle {  public:  void render() { cout << "畫一個半徑 " << \_radius << " 的實心圓" << endl; }  };   * 由於共同的定義被提取至AbstractCircle類別中，並於衍生類別中繼承了下來，所以在ConcreteCircle與HollowCircle 中無需重覆定義，只要定義個別對render()的處理方式就行了，而由於ConcreteCircle與HollowCircle都是 AbstractCircle的子類別，因而可以使用AbstractCircle上所定義的虛擬操作介面，來操作子類別實例上的方法，如下所示：   void render(AbstractCircle &circle) {  circle.render();  }  int main() {  ConcreteCircle concrete;  concrete.radius(10.0);  render(concrete);    HollowCircle hollow;  hollow.radius(20.0);  render(hollow);    return 0;  }  Vtable/Vpointer   * 虛擬函數的運作直接和 VPTR 和 VTABLE 有關。 * 當類別的宣告中含有一個以上的虛擬函數時，編譯器就會為這個類別產生這兩樣東西：VTABLE 裡面存的是這個類別中所有的虛擬函數之位址，VPTR則是一個指向VTABLE的指標。 * 編譯器處理虛擬函數的方法是在每一個類別附加一個隱藏的data member (VPTR) 在此一data member中存放一個指向記載這些虛擬函數記憶體位址的一個特殊矩陣(VTABLE); 一個類別只有一個VTABLE，該 table 基本上就是編譯器在編譯該程式時產生的一個簡單的靜態 (static) 矩陣。 * 基底類別和衍生類別都各自擁有自己的VTABLE. * 類別的建構函數會令VPTR 指向VTABLE，當系統執行到程式在呼叫一個虛擬函數時， 系統會讓程式尋找該一物件object中所存放的vpointer並導引到相對應的 VTABLE 去取得該虛擬函數程式碼所在的位址並執行此一程式碼，這就是系統執行遇到虛擬函數時的處理運作過程。 * 如果衍生類別重新定義了某一虛擬函數, 則會在其VTABLE中更新該新虛擬函數之位址, 否則仍然存放該虛擬函數在基底類別的VTABLE中相同的位址. 如果衍生類別定義了新的虛擬函數, 則會在其VTABLE中加入該新虛擬函數之位址. * 程式執行到一個衍生物件並開始建構時, 它的基底物件會先被建構起來並產生其VTABLE. 如果衍生類別有重新定義某個虛擬函數, 則derived object建構函數會在該衍生物件的 VTABLE中更新被重新定義虛擬函數之位址. 這就是你不應該在建構函數中呼叫虛擬函數的原因: 衍生物件中被重定義的虛擬函數之位址可能還沒有被放入/更改到衍生物件之VTABLE中. 你可能執行到舊的虛擬函數. * C++的原則是 VPTR 會指向設定它的建構函數所屬類別的 VTABLE 。所以當衍生類別在呼叫基底類別的建構函數時，基底類別所設定的 VPTR 當然是指向基底類別的 VTABLE ，等到開始執行衍生類別的建構函數，這時衍生類別的 VPTR 就會指向衍生類別的 VTABLE 了。      * The display function() of A will be executed. WHY? * , display function of B should be called?! * 原因在於early binding / late binding. * 當compile時，compiler keeps ready all the things.   再看一次code  **a = &b;** // what compiler does, is of class A, so function in A is called  **a->display();**   * When early binding is done, compiler doesn’t care about the address pointer is holding. * If you want to give preference to address in pointer, use “virtual” in based class.  |  |  | | --- | --- | |  |  |  * The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.   以下材料來自(<https://www.youtube.com/watch?v=Eaz0P_gJ9FE&t=605s>)   * Anywhere in the inheritance tree, 只要有virtual, we need a vtable. * A type, a vtable. (same type share the same vtable) * Only the virtual functions are put in the vtables  |  | | --- | | struct Animal {  virtual void makeSound() { cout << "Animalll" << endl; }  virtual void walk(){}  void sleep() {}  };  struct Cow: public Animal {  void makeSound() { cout << "mooo" << endl; }  };  struct Pig: public Animal {  void makeSound() { cout << "oink" << endl; }  void walk() {}  };  struct Donkey: public Animal {  void makeSound() { cout << "hee haw" << endl; }  };  int main() {  srand(time(NULL));  Animal\* animal;  switch(rand() % 3) {  case 0:  animal = new Cow;  break;  case 1:  animal = new Pig;  break;  case 2:  animal = new Donkey;  break;  }  animal->makeSound();  delete animal;  } |  |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | Animal:   * (v) makeSound * (v) walk * sleep | Cow:   * (v) makeSound | Pig:   * (v) makeSound * (v) walk | Donkey   * (v) makeSound | | vtable | makeSound:  walk: | makeSound:  walk: | makeSound:  walk: | makeSound:  walk: |  * Ex: if you create an object from Donkey:   The first 4 bytes is the pointer to the v-table.  makeSound() 🡺 go to   * Ex: if several Donkey objects are created, they refer to the same vtable |

map/unordered\_map/set/unordered\_set

|  |
| --- |
| * **map** is built by “**Red-Black Tree**” * **unordered\_map** is built by “**hash**” * RB tree is a balanced tree, good for find(), insert(), delete() (O(logn)) |

Copy control (與dynamic memory management 相關)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| shallow copy 只複制其 array 開頭 (該說是動態指標之 address 較為適宜)，對同一份資料進行修改，故執行完時會影響原本之資料；而 deep copy 為本身再複制一份相同之 array 出來，所有運算和結果存在副本裡面，不影響原本之資料。  以 class 設計而言，只要該 class 有用到**動態記憶體配置**，幾個部份幾乎是必定重改寫：operator = 、constructor 、destructor，  這部分就來談談這些底層的觀念。   * Constructor: control what happen when objects of the class type are created. (若沒寫，compiler 會設default constructor) * Copy constructor (若沒寫，compiler 會設一 synthesized copy constructor) * Assignment operator (若沒寫，compiler會設一synthesized assignment operator) * Destructor (compiler always 有一個你看不到的版本)   注: “copy constructor”, “assignment operator”, “destructor”, 三者合稱c++的 big three，有其一必有其二  \*\*\*\* assign / copy 是不一樣的概念 \*\*\*\*  Type b;  Type a = b; // 同義於 Type a(b); 都是copy constructor  Type b;  Type c;  c = b; // assignment   1. Copy constructor   Take a single parameter that is a **const reference** to an object of the class type  Class Fraction {  Fraction(const Fraction& f); // copy constructor  }  來看看synthesized copy constructor長怎樣，舉個例子  class HasPtr {  private:  std::string \*pstring;  int i;  }  簡單的說，如果用預設的，在成員函數有pointer時會產生問題   |  |  | | --- | --- | | synthesized: | Correct copy constructor | | HasPtr(const HasPtr& obj){  Pstring = obj.pstring;  i = obj.i;  }  此為**shallow copy**, 2 pointer 會同時指向同object | HasPtr(const HasPtr& obj){  Pstring = new string();  \*pstring = \*obj.pstring;  //(上兩行可以合併: pstring = new string(\*obj.pstring));  i = obj.i;  }  此為**deep copy** |  1. assignment operator   接著來看看operation overloading  Class sales\_item{  Sales\_item& operator=(const sales\_item&);  }  (語法可以理解成: return\_type member\_function= passing\_type)  Sales\_item a, b;  a = b; 🡨 equals to **a.operator=(b)**  來看看synthesized Assignment Operator長怎樣，舉個例子  class HasPtr {  private:  std::string \*pstring;  int i;  }   |  |  | | --- | --- | | synthesized: | Correct | | HasPtr& HasPtr::operator=(const HasPtr& obj){  if(this != &obj){  pstring = obj.pstring;  i = obj.i;  }  return \*this;  } | HasPtr& HasPtr::operator=(const HasPtr& obj){  if(this != &obj){  delete pstring;  pstring = new string(\*obj.pstring);  i = obj.i;  }  return \*this;  } |  1. Destructor   Use to clean up memory  Class sales\_item{  Public:  ~sales\_item(){}  }   * Compiler always synthesizes a destructor for us. * You need to write down ur own destructor when you want to release resources acquired in the constructor or during the lifetime of object. * Compiler will run ur desctructor then the synthesized destructor.   總結一下大三元:   * 如果一個class需要用到destructor, 通常也需要用到assignment operator, and copy constructor. 稱為三位一體法則(rule of three)。 |

Dynamic memory

(小記:當初波士頓九月onsite，可以說全然是敗在這一塊，加油加油)

|  |
| --- |
| * new / delete   int\* p1 = new int;  \*p1 = 1001;  …  int\* p2 = new int(1001);  …  int\* p3 = new int(); 預設是0  …  delete p1;  delete p2;  delete p3;  ps: 記憶體非常容易產生問題，若忘記delete 還會有memory leak的問題。   * C++11的解法就是smart pointer * smart pointer: **shared\_ptr** and **unique\_ptr** * **share\_ptr**: multiple smart pointers can refer to the same object, when the last reference to the object gets destroyed, it also gets released. * **unique\_ptr**: only one smart pointer can refer to the object at a time.   用法  shared\_ptr<string> p1; // 指向type 為string的pointer p1  if(p1 && p1->empty()) { // chech p1是否為null以及指向的object是否為空  \*p1 = “hi”;  }  shared\_ptr<string> p2(new int(42)); // shared\_prt points to a int |

Function Templates

|  |
| --- |
| **template** <**class** T>  **void** swaps(T& a1,T& a2){  T tmp=a1;  a1=a2;  a2=tmp;  }  **int** **main**() {  **int** a=10, b=20;  **cout**<<'('<<a<<','<<b<<')'<<**endl**;  swaps(a,b);  **cout**<<'('<<a<<','<<b<<')'<<**endl**;  **return** 0;  }  //output:  (10,20)  (20,10) |

網上OOP面試題小整理

|  |
| --- |
| * What are the core concepts of OOP?   + Abstraction: is the process of concealing the complex logic by hiding its implementation. Whatever the functionality is needed, it can be called by using its method identifier to get the result.   + Encapsulation: Class is used to hide the fields that hold data and the methods to perform the functions.   + Inheritance:   + Polymorphism: process of defining the same method several times to utilize based on data types or arguments. (共兩種polymorphism: runtime polymorphism and compile time polymorphism.) * What is a Class in OOP?   + A Class is ideally called as a template of an object. An object of its class type will have the same properties as defined in the implementation of Class. An object will always be a specific instance of a class. * What is a constructor in OOP?   + A constructor is a method used to invoke the object creation process by initializing the state of the object. * What is the Destructor in OOP?   + A Destructor is a method which is invoked at the time of the object is destroyed. In C++, destructor needs to be called, it is not needed and it will be handled by garbage collection mechanism. * What is an Abstract class in OOP?   + An [abstract class](https://www.educba.com/java-interface-vs-abstract-class/) is used to define at least one abstract method but an object cannot be created from it.  Classes created using abstract classes are called derived classes. An abstract class will not contain implementation code in its base class. * What is multiple-inheritance in OOP?   + Multiple inheritances are the process of extending the behaviors of multiple classes by a single class. It creates an ambiguous situation at runtime to decide about which class’ behavior has to be executed. [C++ supports](https://www.educba.com/c-plus-plus-interview-questions/) multiple inheritances. The ambiguity and complex issues create a diamond problem which is defined as to extend which parent class upon extending multiple classes. * What is static and dynamic binding in OOP?   + Static binding (also called early binding) (happens at compile time)   + Dynamic binding (called late binding) (happens at run time)   + Example of static binding is method **overloading** and dynamic binding is method **overriding**.   + The process of binding for static, final and private methods will always be done at compile time whereas the overriding is done at runtime. The process of binding overloaded methods is called static whereas the binding of overridden methods is called dynamic. * What is operator overloading in OOP?   + Operator overloading is defined as different operators will have different mechanism based on the placement of operators and the arguments. * What is exception handling in OOP?   + Exception handling is defined as the process of handling the exceptions during the execution of program flow. The flow of execution should be altered based on the outcome of the exception. The general blocks of exception handling include try, catch and throw for the most of programming languages like C++, Java etc. In try block the code needs to be executed will be placed and catch block will handle the exception and throw block will return the type of exception and error if it can’t be handled. This is the safest way of handling applications to safeguard the flow of the working application * What are the benefits of OOPS techniques?   + The main benefits of the OOPS concepts are modularity, extensibility, simplicity, reusability, maintainability, modifiability etc. The complexity of the programming can be reduced and the coding structure can be made clear. The different complex functionalities can be decoupled using different classes and implementation methods around the application. The reusability feature provides minor changes in the code whenever needed, which provides the adaptability for the code changes or functionality changes. The objects of the different classes can be reused in different implementation classes to use its features completely. The maintenance process becomes easier if the code is maintained in an organised way. |

inline function

|  |
| --- |
| * A way to speed up the programs. * Normal function calls need to jump to another address and jump back when the function terminates. * “inline” let the compiler replaces the **function call** to the corresponding **function code**. |