# Abstract Base Classes

* C++ supports the distinction between an interface and its implementations through abstract and concrete classes.
* An abstract class is a base class that defines an interface, while a concreate class is a derived class that implements that interface.
* The abstract class identifies the member functions that the class hierarchy exposes to its client and is the gateway to testing the derived classes in its own hierarchy.
* Each concrete class gives a specific meaning to the interface.

#### Pure Virtual Functions

* The principal component of an abstract class is a pure virtual member function.
* Pure refers to the lack of any implementation detail.
* A pure virtual function is a signature without a definition.
* The client code only requires access to the signature.

Declaration

| virtual Type identifier(parameters) = 0; |
| --- |

* The assignment to 0 identifies the virtual function as pure. A pure function must be a virtual member function.

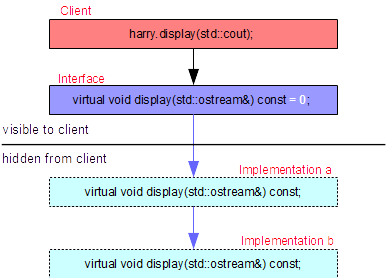
Example:

* We define the pure virtual function for the signature **display(std::ostream&) const** using

| virtual void display(std::ostream&) const = 0; |
| --- |

Implementations

* A pure virtual function typically has multiple definitions within its hierarchy.
* Each definition has the same signature as the pure virtual function but has a different meaning.
* That is, it provides the client with the implementation that suits a specific dynamic type.



#### 

#### 

#### Abstract Class

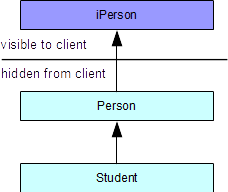
* An abstract class is a class that contains or inherits a pure virtual function. Because the class provides no implementation(s) for its pure virtual function(s), the compiler cannot instantiate the class.
* Any attempt to create an instance of an abstract bas class generates a compiler error.

Definition

* The definition of any abstract base class contains or inherits at least one pure virtual member function.
* The class definition contains the declaration of the pure virtual function.
* We call an abstract base class without any data members a pure interface.

Example

* Let us define an abstract base class named **iPerson** for our **Person** hierarchy and use this class to expose the hierarchy’s **display()** funcion to any client code.



* The **iPerson.h** header file defines our abstract class:

| // Abstract Base Class for the Person Hierarchy // iPerson.h  **#include <iostream> class iPerson { public:  virtual void display(std::ostream&) const = 0; };** |
| --- |

* We derive our **Person** hierarchy from this interface. The header file that defines our **Person** and **Student** class includes the header file that defines our abstract base class:

| // Late Binding // Student.h #include <iostream> #include "iPerson.h" const int NC = 30; const int NG = 20;  class Person : public iPerson {  char name[NC+1]; public:  Person();  Person(const char\*);  void display(std::ostream&) const; };  class Student : public Person {  int no;  float grade[NG];  int ng; public:  Student();  Student(int);  Student(const char\*, int, const float\*, int);  void display(std::ostream&) const; }; |
| --- |

* The class definitions of the **Person** and **Student** classes inform the compiler that each concrete class implements its own version of the **display()** member function.

#### 

#### Arrays of Pointers

* A systematic technique of accessing objects of different dynamic type withing the same hierarchy is through an array of pointers of their static type.
* The executable code dereferences each pointer at runtime based on its object’s dynamic type.

Person Example

* The following code demonstrates the use of an array of pointers to **Person** objects. The objects pointed to by the array elements may be of differing dynamic type, but are of the same static type. The CreatePerson() global function creates a **Person** object and returns its address.

Refer to examples on the course sites

**Client Code**

* The following code manages Person object through the array of pointers **p**. The output generated for the input provided is listed below:

| // Array of Pointers // array\_of\_pointers.cpp  #include <iostream> #include "iPerson.h"  const int NP = 5;  int main() {  iPerson\* p[NP];  for (int i = 0; i < NP; i++)  p[i] = nullptr;   int n = 0;  bool quit = false;  do {  iPerson\* ptemp = CreatePerson();  if (ptemp != nullptr) {  p[n++] = ptemp;  } else {  quit = true;  }  } while(n < NP && !quit);   for (int j = 0; j < n; j++) {  p[j]->display(std::cout);  std::cout << std::endl;  }   for (int j = 0; j < n; j++)  delete p[j]; } |
| --- |

**Interface**

* The interface to a Person object includes the prototype for a global function that creates teh object:

| // Abstract Base Class for the Person Hierarchy // iPerson.h #include <iostream> class iPerson { public:  virtual void display(std::ostream&) const = 0; }; iPerson\* CreatePerson(); |
| --- |

#### 

#### Concrete Class Definitions

* The concrete class definitions specify the various implementations of the iPerson interface:

| // Person and Student Concrete Classes // Student.h  #include "iPerson.h" const int NC = 30; const int NG = 20;  class Person : public iPerson {  char name[NC+1]; public:  Person();  Person(const char\*);  void display(std::ostream&) const; };  class Student : public Person {  int no;  float grade[NG];  int ng; public:  Student();  Student(int);  Student(const char\*, int, const float\*, int);  void display(std::ostream&) const; };  iPerson\* CreatePerson(); |
| --- |

#### Implementations

* Each concrete class that declares the display() member function in its definition defines its own verision of the **display()** function.

#### Unit Tests on an interface

* Good programming practice suggest that we code unit tests for an interface rather than a specific implementation. This practice requires that the interface does not change during the life cycle of the software. With a constant interface we can perform unit tests at every upgrade throughout an object’s lifecycle wihotut changing the test code.

Abstract Class

A pure virtual function is a