# Virtual Functions

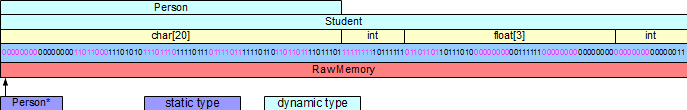
* Polymorphism means “many forms”, and it occurs when we have many classes that are related to each other by inheritance.
* Polymorphism refers to the multiplicity of meanings attached to a single identifier. Polymorphic stands for 'of many forms'
* A polymorphic language selects an operation on an object based on the type associated with the object.

#### Types

Raw memory stores information in the form of bit strings. These bit strings represent variables, objects, addresses, instructions, constants, etc. Without knowing what a bit string represents, the compiler cannot interpret the bit string. By associating a type with a region of memory, we tell the compiler how to interpret the bit string in that region of memory.

**C++ Pointers**

* C++ implements a polymorphic object using pointer syntax.
* The pointer type identifies the static type of the inheritance hierarchy to which the object belongs.
* This static time is known at compile time.
* The pointer holds the address of the polymorphic object.
* To dereference the object’s address, the compiler needs to know its dynamic type.
* The Dynamic type is the reference type of the object.
* Initially, we specify the dynamic type at object creation time through the constructor that we invoke.



* In the following example, we instantiate a Person\* object by dynamically allocating memory once for a Person type and done for a Student Type. By implementing different behaviors for different types in the same hierarchy, we enable different execution paths in show() for different dynamic types.

| **void show(const Person\*);  // Polymorphic Objects  Person jane("Jane"); float g[] = {54.6f, 67.7f, 89.6f}; Student john("John", 1234, g, 3);  Person\* pJane = &jane; Person\* pJohn = &john;  // possibly different behaviors show(pJohn); show(pJane);** |
| --- |

#### Function Bindings

* The compiler binds a function call to a function definition using an object’s type. The object’s type determines the member function to call in the inheritance hierarchy.
* The binding of a member function can take either of two forms:

1. **Early Binding:** Based on the object’s static type
2. **Dynamic DIspatch:** Based on the object’s dynamic type.

**Early Binding**

* Compile-time time polymorphism
* The compiler or linker directly associates an address to the function call

**Dynamic Dispatch**

* C++ Provides the keyword **virtual** for dynamic dispatching. If this keyword is present, the compiler inserts code for the bind the call to the most derived version of the member function based on the dynamic type.
* Refer to example on course site

**Overriding Dynamic DIspatch**

* To override dynamic dispatch with early binding

| void show(const Person& p) {  p.Person::display(std::cout); } |
| --- |

**Documentation**

Some programmers include the qualifier virtual in derived class declarations as a form of documentation. This improves readability but has no syntactic effect.

We can identify a member function as virtual even if no derived class exists. This clarifies the intent of the original developer for subsequent developers of the hierarchy.

**Polymorphic Objects**

* A polymorphic object is an object that can change its dynamic type throughout its lifetime.
* Its static type identifies the hierarchy of types to which the object belongs.
* Its dynamic type identifies the rule for interpreting the bit string in the region of memory currently allocated for the object.

We specify the static type of a polymorphic object through:

* A pointer declaration
* A receive-by-address parameter
* A receive-by-reference parameter

| // Polymorphic Objects - Static Type #include <iostream> #include "Student.h" void show(const Person\* p) { // ... }  void show(const Person& p) { // ... }  int main() {  Person\* p = nullptr;  // ... } |
| --- |

**Example:**

| // Polymorphic Objects - Dynamic Type // dyanmicType.cpp #include <iostream> #include "Student.h"  void show(const Person& p) {  p.display(std::cout);  std::cout << std::endl; }  int main() {  Person\* p = nullptr;  p = new Person("Jane Doe");  show(\*p);  delete p;  float g[] = {89.4f, 67.8f, 45.5f};  p = new Student("Harry", 1234, g, 3);  show(\*p);  delete p; } |
| --- |

In the **main()** function:

* **p** initially points to nothing (holds the null address). The object's dynamic type is undefined.
* After the first allocation, **p** points to a Person type (dynamic type).
* After the second allocation, **p** points to a Student type (the new dynamic type).
* The static and dynamic types are related to one another through the hierarchy.

Note that we only need one show() function to display both dynamic types.

p holds the address a polymorphic object throughout its lifetime. That address may change with deallocations and fresh allocations of memory. The dynamic type may be of any type in the Person hierarchy.

show() is a polymorphic function. Its parameter receives an unmodifiable reference to any type in the Person hierarchy.

#### Summary

* Polymorphism refers to the multiplicity of logic associated with the same name.
* Static type is the type of the object's hierarchy and is available at compile-time
* Dynamic type is the type of the object referenced and may change with different calls to the same function
* Early binding of a call to a member function's definition occurs at compile-time
* The keyword virtual on a member function's declaration specifies dynamic dispatch
* A polymorphic object's pointer type identifies the object's static type
* A polymorphic object's constructor identifies the object's dynamic type
* Declare a base class destructor virtual even if there are no derived classes