PART 3

Virtual Function and Exception

C++ / Python Programming

Session #2

@Institut d'Astrophysique de Paris

OUTLINE

1. Virtual Function
2. Abstract Classes
3. Const Modifier
4. Inline Function
5. Exception
6. Error handling with Exception

Virtual Functions

- virtual function is member function with polymorphic behavior
- to make member function virtual, add keyword **virtual** to function declaration
- once function made virtual, it will *automatically* be virtual in all derived classes, regardless of whether **virtual** keyword is used in derived classes
- example:

```
class Base {
public:
    virtual void func(); // virtual function
    // ...
};
```

Virtual Functions

- once function made virtual, it will *automatically* be virtual in all derived classes, regardless of whether **virtual** keyword is used in derived classes
- therefore, not necessary to repeat **virtual** qualifier in derived classes (and perhaps preferable not to do so)
- virtual function must be defined in class where first declared unless pure virtual function (to be discussed shortly)
- derived class inherits definition of each virtual function from its base class, but may override each virtual function with new definition
- function in derived class with same name and same set of argument types as virtual function in base class overrides base class version of virtual function

Virtual Function Example:

```
#include <iostream>
#include <string>
class Person {
public:
  Person(const std::string& family, const std::string& given) : family_(family), given_(given) {}
  virtual void print() const {std::cout << "person: " << family_ << ’,’ << given_ << ’\n’;}
protected:
  std::string family_;          // family name
  std::string given_;           // given name
};

class Student : public Person {
public:

  Student(const std::string& family, const std::string& given, const std::string& id) : Person(family, given), id_(id) {}

  void print() const { std::cout << "student: " << family_ << ’,’ << given_ << ’,’ << id_ << ’\n’;

} private:

  std::string id_;              // student ID
};
```

Virtual Function Example:

```
void processPerson(const Person& p) {
  p.print();                    // polymorphic function call
  // ...
}

int  main() {
   Person p("Ritchie", "Barry");
   Student s("Allen", "John", "12345678");
   processPerson(p);                    // invokes Person::print
   processPerson(s);                    // invokes Student::print
}
```

Constructor, Destructor and Virtual Functions

- except in very rare cases, destructors in class hierarchy need to be virtual
- otherwise, invoking destructor through base-class pointer/reference would only destroy base-class part of object, leaving remainder of derived-class object untouched
- normally, bad idea to call virtual function inside constructor or destructor
- dynamic type of object changes during construction and changes again during destruction
- final overrider of virtual function will change depending where in hierarchy virtual function call is made
- when constructor/destructor being executed, object is of exactly that type, never type derived from it
- although semantics of virtual function calls during construction and destruction well defined, easy to write code where actual overrider not what expected (and might even be pure virtual)

Example: Non-Virtual Destructor (Problematic)

```
class Base {
 public:
   Base() {}
   ~Base() {} // non-virtual destructor
    // ...
};

class Derived : public Base {
 public:
    Derived() : buffer_(new char[10000]) {}
    ~Derived() {delete[] buffer_;}
    // ...
 private:
    char* buffer_;
};

void process(Base* bp) {
   // ...
 delete bp; // always invokes only Base::~Base
}

int main() {
   process(new Base);
   process(new Derived);   // leaks memory
}
```

Example: Virtual Destructor (corrected)

```
class Base {
 public:
   Base() {}
   virtual ~Base() {}    //virtual destructor
    // ...
};

class Derived : public Base {
 public:
    Derived() : buffer_(new char[10000]) {}
    ~Derived() {delete[] buffer_;}
    // ...
 private:
    char* buffer_;
};

void process(Base* bp) {
   // ...
 delete bp;          // invokes destructor polymorphically
}

int main() {
   process(new Base);
   process(new Derived);
}
```

Pure Virtual Functions

- sometimes desirable to require derived class to override virtual function
- pure virtual function: virtual function that must be overridden in every derived class
- to declare virtual function as pure, add "= 0" at end of declaration
- example:

```
class Widget {
 public:
   virtual void doStuff() = 0;    // pure virtual
   // ...
};
```

- pure virtual function can still be defined, although likely only useful in case of virtual destructor

Abstract Classes

- class with one or more pure virtual functions called abstract class
- cannot directly instantiate objects of abstract class (can only use them as base class objects)
- class that derives from abstract class need not override all of its pure virtual methods
- most commonly, abstract classes have no state (i.e., data members) and used to provide interfaces, which can be inherited by other classes
- if class has no pure virtual functions and abstract class is desired, can make destructor pure virtual (but must provide definition of destructor since invoked by derived classes)

Abstract Class Example

```cpp
#include <cmath>

class Shape {
public:
  virtual bool isPolygon() const = 0;
  virtual float area() const = 0;

  virtual~Shape() {};
};


class Rectangle : public Shape {
public:
  Rectangle(float w, float h) : w_(w), h_(h) {}
  bool isPolygon() const override {return true;}
  float area() const override {return w_ * h_;}
private:
  float w_;          // width of rectangle
  float h_;          // height of rectangle
};
```

```cpp
class Circle : public Shape {

public:
  Circle(float r) : r_(r) {}
  float area() const override {
    return M_PI * r_ * r_;
  }
  bool isPolygon() const override {
    return false;
  }

private:
  float r_;          // radius of circle

};
```

Pure Virtual Destructor Example

```
class Abstract {
public:
    virtual~Abstract() = 0;                    // pure virtual destructor
    // ... (no other virtual functions)
};

inline Abstract::~Abstract()
  { /* possibly empty */ }
```

Exercise (1/2)

To have a polymorph behavior a class need at least a virtual method. The address of this method is stored in an array of virtual methods. Download the code from Exercises/download/VirtualMethods.

1. Test this code, then add the virtual keyword to the destructor and test the code again. What result displays the program ? Why ? Now add the keyword virtual to the affiche_info(), perimeter() and surface() methods. Has the object size changed ? Why ?
2. Now add a method void bidule() at your class without implemented it. Check your code isn't wrong. Add the keyword virtual. Why is the link edition wrong ?

Exercise (2/2)

- Define a pure abstract class called shape with only 3 public methods :

  virtual void print_info() const=0;

  virtual float perimetre() const=0;

  virtual float surface() const=0;

- Why must you add const=0 ?
- Write a circle class like rectangle class. A circle is charaterized by a center and a radius. Rectangle and circle classes inherit of shape
- Implement for each class perimetre and surface functions.

Const

- const qualifier specifies that object has value that is constant (i.e., cannot be changed)

```
const int x = 2; //x can not be changed (read as "x is an integer which is constant")
int const x = 2; //read as "x as constant integer"
x = 5; // Error: X is constant
```

- pointer to constant integer:

```
const int *ptr2const;  « or »   int const *ptr2const;
* ptr2const = 0;              // Error! Cannot modify the "pointee" data
 ptr2const = NULL;            // OK: modifies the pointer
```

- constant pointer to integer:

```
int *const constPtr;
 * constPtr = 0;              // OK: modifies the "pointee" data
constPtr = NULL;              // Error! Cannot modify the pointer
```

- constant pointer to constant integer:

```
 const int * constPtr2const
 * constPtr2const = 0;             // Error! Cannot modify the "pointee" data
constPtr2const = NULL;             // Error! Cannot modify the pointer
```

Example: Const with pointer

Program without const

```
#include <iostream>
int main(){
  int i = 10;
  int j = 20;
  int *ptr = &i;
  std::cout<< "ptr: "<< *ptr << std::endl;
  *ptr = 100;
  std::cout<< "ptr: "<< *ptr << std::endl;
  ptr = &j; // valid
  std::cout<< "ptr: "<< *ptr << std::endl;
 }
```

Program with const

```
#include <iostream>
int main(){
  int i = 10;
  int j = 20;
  const int *ptr = &i; // ptr is pointer to constant
  std::cout<< "ptr: "<< *ptr << std::endl;
  *ptr = 100; // error: object pointed cannot be modified using the pointer ptr
  std::cout<< "ptr: "<< *ptr << std::endl;
  ptr = &j; // valid
  std::cout<< "ptr: "<< *ptr << std::endl;
  const int *const ptr = &i; // constant pointer to constant integer
  std::cout<< "ptr: "<< *ptr << std::endl;
  //ptr = &j;     // error
 //*ptr = 100;   // error
}
```

Example: const with function

Function without const

- int A::func (int random_arg)
  can be represent function like:
  int A_func (A* this, int random_arg)
- In driver code:
  For first representation:
  A a;
  a.func(4);
  will internally correspond to something like
  A a;
  A_func(&a, 4).

Function with const

int A::func(int random_arg) const
can then be understood as a declaration with a const this pointer:
int A_func(const A* this, int random_arg).

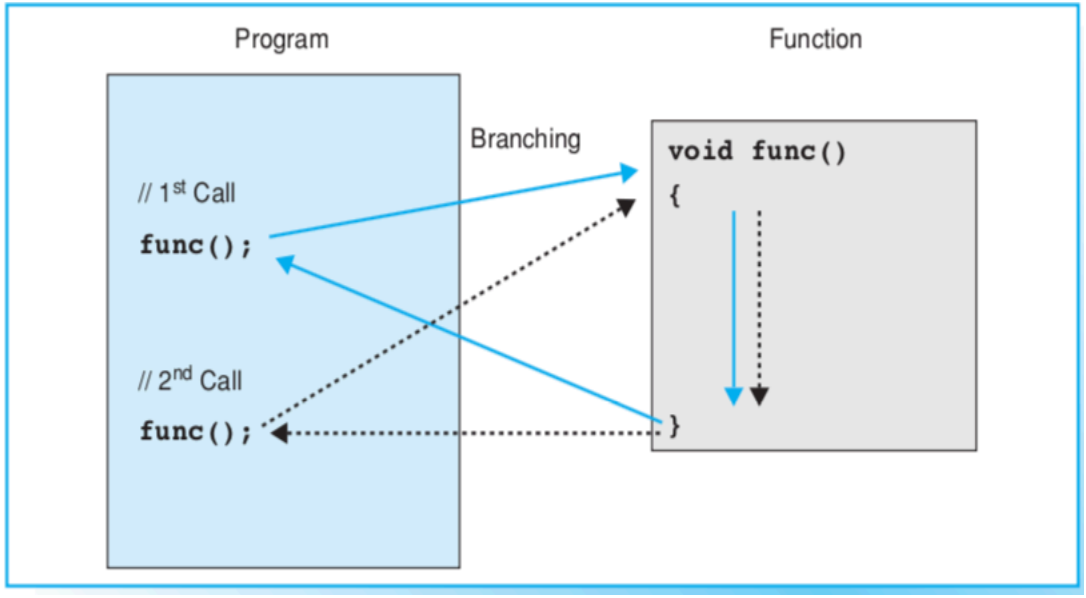
```
Class polygon {
private:
void calculate_area() { /* we calculate area_ */ }
public:
double area() const { return area_; }
......
}
```

Inline Functions

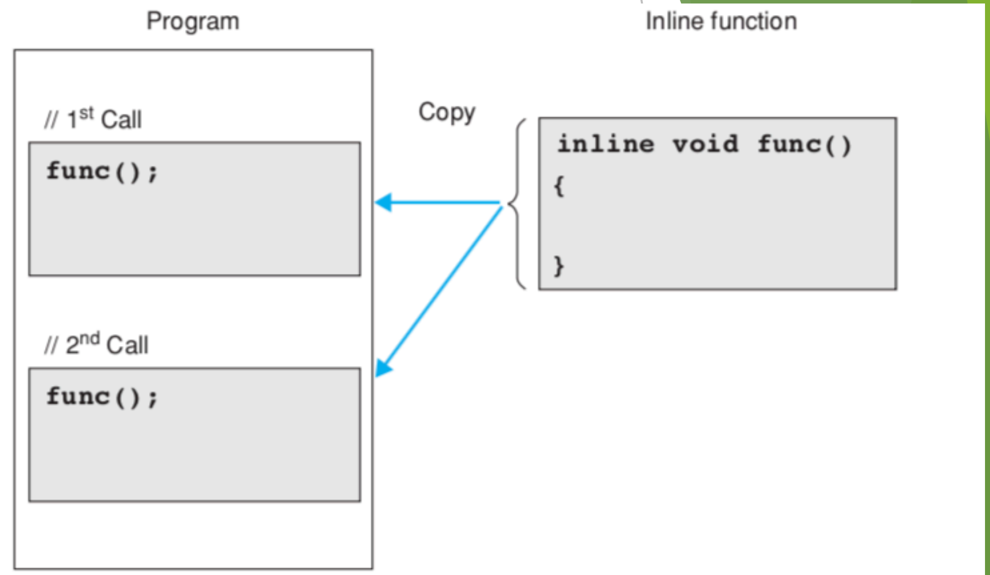
- Function for which compiler copies code from function definition directly into code of calling function rather than creating separate set of instructions in memory
- no need to transfer control to separate piece of code and back again to caller, eliminating performance overhead of function call
- inline typically used for very short functions (where overhead of calling function is large relative to cost of executing code within function itself)
- Example:

```
inline bool isEven (int x)
{
  return x % 2 == 0;
}
```

Inline



Function without inline



Function with inline

To define inline function, inline keyword is used in the function header

Inlining of a function

- Inlining of **isEven** function transforms code fragment 1 into code fragment 2
- Code fragment 1:

```
inline bool isEven(int x) {
  return x % 2 == 0;
}

void myFunction() {
  int i = 3;
  bool result = isEven(i);
}
```

- Code fragment 2:

```
void myFunction() {
   int i = 3;
   bool result = (i % 2 == 0);

}
```

Exceptions

- language mechanism for handling exceptional (i.e., abnormal) situations
- situations perhaps best thought of as case when code could not do what it was asked to do and usually (but not always) corresponds to error condition, error handling
- exceptions propagate information from point where error *detected* to point where error *handled*
- code that encounters error that it is unable to handle throws exception
- code that wants to handle error catches exception and performs processing necessary to handle error
- exceptions provide convenient way in which to *separate error detection from error handling*

Traditional Error handling

- if any error occurs, terminate program
- pass error code back from function (via return value, reference parameter, or global object) and have caller check error code

Example:

```
#include <iostream>
bool func3() {
  bool success = false;
  // ...
  return success;
}

bool func2() {
  if (!func3()) {
    return false;
  }
  // ...
  return true;
}

bool func1() {
  if (!func2()) {
    return false;
  }
  // ...
  return true;
}

int main() {
  if (!func1()) {
     std::cout << "failed\n";
     return 1;
  }
  // ...
}
```

Error Handling with Exceptions

- provide convenient way in which to separate error detection from error handling
- when error condition detected, signalled by throwing exception (with throw statement)
- thrown exception caught by handler (in catch clause of try statement), which takes appropriate action to handle error condition associated with exception

```
#include <iostream>
using namespace std;
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main () {
   int x = 50;
   int y = 0;
   double z = 0;
   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
      cerr << msg << endl;
   }
   return 0;
}
```

Exceptions

- exceptions are objects
- type of object used to indicate kind of error
- value of object used to provide details about particular occurrence of error
- exception object can have any type (built-in or class type)
- for convenience, standard library provides some basic exception types
- all exception classes in standard library derived (directly or indirectly) from std::exception class
- exception object is propagated from one part of code to another by throwing and catching
- exception processing disrupts normal control flow

Standard Exception Classes

- Exception Classes Derived from exception Class

Type	Description
logic_error	faulty logic in program
runtime_error	error caused by circumstances beyond scope of program
bad_typeid	invalid operand for typeid operator
bad_cast	invalid expression for dynamic_cast
bad_weak_ptr	bad weak_ptr given
bad_function_call	function has no target
bad_alloc	storage allocation failure
bad_exception	use of invalid exception type in certain context

Throwing Exceptions

- throwing exception accomplished by **throw** statement
- throwing exception transfers control to handler
- object is passed
- type of object determines which handlers can catch it
- handlers specified with **catch** clause of **try** block
- for example

```
throw "OMG!";
```

can be caught by handler of **const char*** type, as in:

```
try {
// ...
}
catch (const char* p) {
// handle character string exceptions here
}
```

Catch Exceptions

- exception can be caught by **catch** clause of **try-catch** block (code that might throw exception placed in **try** block & code to handle exception placed in **catch** block)
- **try-catch** block can have multiple **catch** clauses & **catch** clauses checked for match in order specified and only first match used
- **catch** (...) can be used to catch any exception
- example:

```
try {
// code that might throw exception
}
catch (const std::logic_error& e) {
// handle logic_error exception
}
catch (const std::runtime_error& e) {
// handle runtime_error exception
}
catch (...) {
// handle other exception types
}
```

Rethrowing Exceptions

- caught exception can be rethrown by **throw** statement with no operand
- example:

```
try {
// code that may throw exception
}
catch (...) {
throw;          // rethrow caught exception
}
```

Example:

```
void handle_exception() {
   try { throw; }
   catch (const exception_1& e) {
      log_error("exception_1 occurred");
      // ...
   }
   catch (const exception_2& e) {
      log_error("exception_2 occurred");
      // ...
   }
 // ...
}

void func() {
   try {operation();}
   catch (...) {handle_exception();}
   // ...
   try {another_operation();}
   catch (...) {handle_exception();}
 }
```

allows reuse of exception handling code

References

1. A tower of C ++ - Bjarne Stroustrup
2. Effective Modern C ++ - Scott Meyers
3. Thinking in C ++ - Bruce Eckel
4. Websites:
   - https://en.cppreference.com/w/
   - http://www.cplusplus.com/
   - https://www.tutorialspoint.com/cplusplus/
   - https://www.onlinegdb.com/online_c++_compiler
   - https://www.geeksforgeeks.org