Chapter: 13

**Polymorphism, Exceptions, RTTI, Operator cast**

Virtual functions, Generic function & class, Exception, RTTI, Casting Operators

**13.1 Pointers To Derived Classes**

A *pointer* declared as a pointer to a base class can also be used to point to any class derived from that base. For example, assume two classes called ***base*** and ***derived***, where ***derived*** inherits ***base***. Then the following statements are correct:

**base** \*p; /\* base class pointer \*/

**base** base\_ob ; /\* *object* of type base \*/

**derived** derived\_ob ; /\* *object* of type derived \*/

p = &base\_ob ; /\* p points to base object : Normally p can.\*/

p = &derived\_ob ; /\* p points to derived object : Advanced pointing by p \*/

* A base pointer can point to an object of any class derived from that base without generating a type mismatch error.
* By a base pointer we can access only those members of the derived object that were inherited from the base. Because the base pointer has *knowledge only of the base* class, nothing about the *members added by the derived* class.
* The reverse is not true: A *pointer of the derived type* cannot be used to access an *object of the base* class. (A type cast can be used to *overcome this restriction*, but its use is not recommended practice.)
* Note (Be careful): *Pointer arithmetic is relative to the data type* the pointer is declared as pointing to. Thus, if you point a base pointer to a derived object and then increment that pointer, it will *not be pointing* to the next derived object. It will be *pointing to* (what it thinks is) the next base object. Be careful about this.
* Example 1: Following illustrates how a base class pointer can be used to access a derived class:

|  |  |
| --- | --- |
| **class** base{ **int** x;  **public**: **void** setx(**int** i){ x=i; }  **int** getx(){ **return** x; } };  **class** derived : **public** base { **int** y;  **public** : **void** sety(**int** i){ y=i; }  **int** gety(){ **return** y; } };  **int** **main**(){ **base** \*p; /\* pointer to base type \*/  **base** b\_ob ; /\* *object* of base \*/  **derived** d\_ob ; /\* *object* of derived \*/ | p = &b\_ob ; /\* p access base : point to base object \*/  p-> **setx**(10) ; /\* access base object \*/  **cout** << " Base object x: " << p-> getx () << '\n';  p = &d\_ob ; /\* p access derived : point to derived object \*/  p**->** **setx**(99) ; /\* access derived object \*/  d\_ob.**sety**(88) ; /\* can't use p to set y, so do it directly \*/  **cout** << " Derived object x: " << p**->** **getx**() << '\n';  **cout** << " Derived object y: " << d\_ob.**gety**() << '\n';  **return** 0; } |

* There is no value in using a base class pointer in the way shown in this example.

**13.2 Virtual Functions (VF)**

A virtual function is a *member function* that is declared within a base and redefined *by a derived*. To create a virtual function, ***precede the function's declaration*** with the keyword virtual. When a VF is redefined by a derived, the keyword ***virtual*** is not needed. A class that contains a VF is referred to as a polymorphic class.

* When a class containing a VF is inherited, the derived *redefines the VF* relative to the derived. VF implements the ***"one interface, multiple methods"*** philosophy that underlies polymorphism.
* The VF within the base class defines the ***form*** of the ***interface*** to that function.
* Each redefinition of the VF by a *derived* implements its operation as it *relates specifically to that* derived. I.e. the ***redefinition creates a*** specific method.
* A VF can be called just like any other *member function*. But interesting thing happens when a virtual function is called through a pointer- creates the run-time polymorphism.
* When a base pointer points to a *derived object that contains a VF* and that VF is called *through that pointer*, it is the type of that pointed object that determines which version of the VF will be executed at the time when the call occurs. And, this determination is made at run time. This process is the way that ***run-time polymorphism is achieved***.
* Therefore, if two or more different classes are derived from a base class that contains a VF, then when ***different objects are pointed to by a base pointer, different versions of the virtual function are executed***.
* Example 1: Following uses a VF. Here the type of the object being pointed to determines which version of an overridden virtual function will be executed when accessed via a base class pointer, and that this decision is made at run time.

|  |  |  |
| --- | --- | --- |
| **class** base{  **public**: **int** i;  base(**int** x) { i = x; }  **virtual** **void** func() { **cout**<< " *Using base version of func():* ";  **cout** << i << '\n'; }  }; | | |
| **class** derived1 : **public** base {  **public** : derived1(**int** x) : base(x){} */\* passing argument to base constructor and uses same definition. Using base's constructor \*/*  **void** func(){**cout**<< *" Using derived1's version of func (): "*;  **cout** << i\*i << '\n'; }  }; | **class** derived2 : **public** base {  **public** : derived2(**int** x) : base(x){} */\* passing argument to base constructor and uses same definition. Using base's constructor \*/*  **void** func(){**cout** << *" Using derived2 's version of func (): "*;  **cout** << i+i << '\n'; }  }; | |
| **int** **main**(){ **base** \*p;  **base** ob (10) ;  **derived1** d\_ob1 (10) ;  **derived2** d\_ob2 (10) ;  p = **&**ob; p **->** func(); /\* use base 's func() \*/  p = **&**d\_ob1 ; p **->** func(); /\* use derived1's func() \*/  p = **&**d\_ob2 ; p **->** func(); /\* use derived2 's func() \*/  **return** 0; } | | This program displays the following output:  ***Using base version of func( ): 10***  ***Using derived1's version of func( ): 100***  ***Using derived2's version of func( ): 20*** |

* Above program creates three classes.
* The ***base*** class defines the virtual function **func()**.
* "***base***" is then inherited by both ***derived1*** and ***derived2***. Each of these classes overrides ***func()*** with its individual implementation.
* Inside ***main()***, the ***base pointer*** ***p*** is declared along with objects of type ***base***, ***derived1***, and ***derived2***.

1. First, ***p*** is assigned the address of ***ob*** (***base type*** object). When ***func()*** is called by using ***p***, the ***base*** ***version*** of ***func()*** is used.
2. Next, ***p*** is assigned the address of ***d\_ob1***. In this time ***derived1*** version (the ***overridden*** version) of ***func()*** is executed when ***func()*** is called using ***p***. (the ***type of the object pointed to*** determines which VF will be called)
3. Finally, ***p*** is assigned the address of ***d\_ob2*** and ***func()*** is called again by using ***p***. This time, it is the ***overridden*** version of ***func()*** defined inside ***derived2*** is executed.

* Virtual functions are hierarchical in order of inheritance. Further, ***when a derived class does not override a virtual function, the function defined within its base class is used***. For example consider the previous example with the modified "***derived2***"

|  |  |  |
| --- | --- | --- |
| **class** derived2 : **public** base  { **public** :  **derived2** (**int** x) : base(x){}  */\* derived2 does not override func( ) \*/*  }; | **int** **main**() {. . . . .  p = **&**d\_ob2 ;  p **->** func();  */\* use base's func( ) \*/*  **return** 0;  } | This program displays the following output:  ***Using base version of func( ): 10***  ***Using derived1's version of func( ): 100***  ***Using base version of func( ): 10*** |

* In this version, ***derived2*** does not override ***func()***. When ***p*** is assigned ***d\_ob2*** and ***func()*** is called, ***base***'s version is used because it is ***next up in the class hierarchy***. In general, *when a derived class does not override a VF, the base class's version is used*.
* A VF can respond to ***random events that occur at run time***. Consider Example 1, following modified ***main()*** selects between ***d\_ob1*** and ***d\_ob2*** based upon the value returned by the ***standard random number generator*** ***rand()***.
* Remember that the version of ***func()*** executed is resolved at ***run time***. (Which is impossible at ***compile time***.)

**int** **main**(){ **base** \*p;

**derived1** d\_ob1 (10);

**derived2** d\_ob2 (10);

**int** i,j;

**for**(i=0; i<10; i++){ j = **rand**();

**if**((j%2) ) p = **&**d\_ob1 ; /\* if odd use d\_ob1 \*/

**else** p = **&**d\_ob2 ; /\* if even use d\_ob2 \*/

p **->** func(); } /\* call appropriate function \*/

**return** 0; }

Note

1. Redefinition of a VF inside a derived class and function overloading are different process (although they look similar).

* An ***overloaded function must differ in type and/or number of parameters***, while *a redefined VF must have precisely the same* ***type*** *and number of* ***parameters*** *and the same* ***return type***. *(changing either the number or type of parameters of redefined VF destroys its virtual nature and makes it an "overloaded function")*
* Virtual functions must be class members. This is not the case for overloaded functions.
* *Destructor* functions can be virtual, *constructors* cannot.
* Because of the difference between ***overloaded functions*** and ***redefined VF***, the term "overriding" is used to describe VF redefinition.

**13.3 Abstract class and Pure Virtual function (PVF)**

Sometimes when a VF is declared in the base class there is no meaningful operation for it to perform. Because often *a base simply supplies a core set of member functions and variables* to which the *derived class supplies the remainder*. In this case we use pure virtual functions (PVF).

* A PVF has no definition relative to the base class. Only the function's prototype is included. To make a PVF, use this general form:

***virtual type func\_name(parameter\_list =0;***

* The key part of this declaration is the setting of the ***function equal to 0***. This tells the compiler that *no body exists for this function relative to the base class*.
* When a *virtual* *function* is made *pure*, it forces any derived class to override it. If a derived class does not, a compile-time error results.
* Abstract class: When a class contains at least one PVF, it is referred to as an abstract class. It is an incomplete type, and ***no objects of that class can be created***. Thus, abstract classes exist only to be inherited. They are neither intended nor able to stand alone.
* You can still create a pointer to an abstract class, since it is through the use of *base class pointers* that ***run-time polymorphism*** is achieved.
* It is also permissible to have a reference to an abstract class.
* When a VF is inherited, so is its *virtual nature*. I.e. when a derived inherits a VF from a base and then *that derived* is used as a *base* for yet another derived, the VF can be *overridden by the final derived* class (as well as the first derived). For example, if base **B** contains a VF called **f()**, and **D1** inherits **B** and **D2** inherits **D1**, both **D1** and **D2** can override **f()** relative to their respective classes.
* Example 1: This program creates a base called ***area*** that holds two dimensions of a figure. It also declares a VF called ***getarea()*** that, when overridden by derived classes, returns the area of the ***type of figure defined by the derived***.

|  |  |
| --- | --- |
| **class** area{ **double** dim1 , dim2 ; /\* dimensions of figure \*/  **public** : **void** setarea(**double** d1, **double** d2){ dim1 = d1; dim2 = d2;}  **void** getdim(**double** &d1 , **double** &d2) { d1 = dim1 ; d2 = dim2 ; }  **virtual** **double** getarea(){ **cout** << "*You must override this function* \n";  **return** 0.0; }  }; | |
| **class** rectangle : **public** area {  **public** :  **double** getarea(){**double** d1, d2;  getdim(d1, d2);  **return** d1\*d2; }  }; | **class** triangle : **public** area {  **public** :  **double** getarea() { **double** d1, d2;  getdim(d1, d2);  **return** 0.5\*d1\*d2;}  }; |
| **int** **main**(){ **area** \*p;  **rectangle** r;  **triangle** t;  r.setarea(3.3, 4.5) ;  t.setarea(4.0, 5.0) ;  p = &r; **cout** << " *Rectangle has area :* " << p-> getarea() << '\n';  p = &t; **cout** << " *Triangle has area :* " << p-> getarea() << '\n';  **return** 0; } | |

* In this case, the declaration of ***getarea()*** inside the base determines the nature of the interface. The actual implementation is left to the classes that inherit it. In this example, the area of a *triangle* and a *rectangle* are computed.
* Here the definition of getarea() inside area is just a placeholder and performs no real function. Because area is not linked to any specific type of figure, there is no meaningful definition that can be given to getarea() inside area.
* ***getarea()*** must be overridden by a derived class in order to be useful.

**class** area{ . . . . . . same as previous . . . . .

**virtual** **double** getarea() = 0; /\* pure virtual function \*/ };

* Example 2: Following program illustrates how a function's *virtual nature* is preserved when it is *inherited*:

|  |  |  |
| --- | --- | --- |
| class base {  public : virtual void func(){  cout << *" Base version of func ( )\n"*; } }; | class derived1 : public base {  public : void func(){  cout <<*"derived1's version of func()\n"*; }  }; | class derived2 : public derived1{ /\* *derived2 inherits derived1* \*/  public : void func(){  cout << *"derived2's version of func()\n"*; } }; |
| **int** **main**() { **base** \*p;  **base** ob;  **derived1** d\_ob1 ;  **derived2** d\_ob2 ;  p = **&**ob; p **->** func(); /\* use base's func() \*/  p = **&**d\_ob1 ; p **->** func(); /\* use derived1's func() \*/  p = **&**d\_ob2 ; p **->** func(); /\* use derived2's func() \*/  **return** 0; } | | |

* The VF ***func()*** is first inherited by ***derived1***, which overrides it relative to itself. Next, ***derived2*** inherits ***derived1***. In ***derived2***, ***func()*** is again overridden.
* Since VFs are hierarchical, if ***derived2*** did not override ***func()***, when ***d\_ob2*** was accessed, ***derived1***'s ***func()*** would have been used.
* If neither ***derived1*** nor ***derived2*** had overridden ***func()***, ***base's func( )*** would have been used.

**13.4 Polymorphism: Early binding & Late binding**

Polymorphism: Polymorphism is the process by which a common interface is applied to two or more similar (but technically different) situations, thus implementing the "one interface, multiple methods" philosophy. In polymorphism a single, well-defined interface is used to access a number of *different but related* actions, and artificial complexity is removed.

* There are two terms that are often linked to OOP in general and to C++ specifically. They are early binding and late binding.
* Early binding: Early binding essentially refers to those events that can be known at compile time. Specifically, it refers to those function calls that can be *resolved during compilation*. Early bound entities include:

|  |  |  |  |
| --- | --- | --- | --- |
| 1. "Normal" functions, | 1. Overloaded functions, | 1. Non-virtual member | 1. Friend functions. |

* When these types of functions are compiled, *all address information* necessary to call them is known at *compile time*.
* Calls to ***functions bound at compile time*** are the *fastest types of function calls*. Main disadvantage is lack of flexibility.
* Late binding: Late binding refers to events that must occur at ***run time***. A late bound function call is one in which the address of the function to be called is not known until the program runs.
* In C++, a virtual function is a late bound object. When a VF is accessed via a base class pointer, the program must determine at run time what ***type of object*** is being pointed to and then select which version of the ***overridden function*** to **execute**.
* Advantage: Flexibility at run time. Disadvantage: is that there Slower than ***early binding***.
* Example 1: Here is a program that illustrates "***one interface, multiple methods***." It defines an abstract list class for integer values.
* The interface to the list is defined by the PVFs ***store()*** and ***retrieve()***. To *store a value*, call ***store()***. To *retrieve a value*, call ***retrieve()***.
* The base list does not define any ***default methods*** for these actions. Instead, each derived defines exactly ***what type of list*** will be maintained.
* In the program, two types of lists are implemented: a queue and a stack. Although the two lists operate completely differently, each is accessed using the ***same interface***.

|  |  |
| --- | --- |
| #include<iostream >  #include<cstdlib >  **using namespace std;**  **class** list{ **public**: **list** \*head ; /\* pointer to start of list \*/  **list** \*tail ; /\* pointer to end of list \*/  **list** \*next ; /\* pointer to next item \*/  **int** num ; /\* value to be stored \*/  list(){ head = tail = next = NULL ; }  **virtual** **void** store(**int** i) = 0; /\* PVF \*/  **virtual** **int** retrieve() = 0; /\* PVF \*/  }; | |
| */\* Create a queue - type list.\*/*  **class** queue : **public** list {  **public** : **void** store(**int** i);  **int** retrieve();  };  **void** queue :: store(**int** i){ **list** \*item;  item = **new** queue ;  **if**(!item ){ **cout** << *" Allocation error .\n"*;  **exit**(1); }  item **->** num = i;  */\* put on end of list \*/*  **if**(tail) tail **->** next = item ;  tail = item ;  item **->** next = **NULL** ;  **if**(!head) head = tail ; }  **int** queue :: retrieve(){ **int** i;  **list** \*p;  **if**(!head){ **cout** << *" List empty.\n "*;  **return** 0; }  */\* remove from start of list \*/*  i = head **->** num;  p = head ;  head = head **->** next ;  **delete** p;  **return** i; } | */\* Create a stack - type list.\*/*  **class** stack : **public** list {  **public** : **void** store(**int** i);  **int** retrieve();  };  **void** stack :: store(**int** i){ **list** \*item ;  item = **new** stack ;  **if**(!item){ **cout** << *" Allocation error.\n"*;  **exit**(1); }  item **->** num = i;  */\* put on front of list for stack - like operation \*/*  **if**(head) item **->** next = head ;  head = item ;  **if**(!tail) tail = head ; }  **int** stack :: retrieve(){ **int** i;  **list** \*p;  **if**(!head){ **cout** << *" List empty .\n"*;  **return** 0; }  */\* remove from start of list \*/*  i = head **->** num;  p = head ;  head = head **->** next ;  **delete** p;  **return** i; } |
| **int** **main**() { **list** \*p;  */\* demonstrate queue \*/*  **queue** q\_ob ;  p = **&**q\_ob; /\* point to queue \*/  p**->** store(1);  p**->** store(2);  p**->** store(3);  **cout** << *" Queue : "*;  **cout** << p**->** retrieve();  **cout** << p**->** retrieve();  **cout** << p**->** retrieve();  **cout** << '\n'; | */\* demonstrate stack \*/*  **stack** s\_ob;  p = **&**s\_ob; /\* point to stack \*/  p**->** store(1);  p**->** store(2);  p**->** store(3);  **cout** << *" Stack : "*;  **cout** << p-> retrieve();  **cout** << p-> retrieve();  **cout** << p-> retrieve();  **cout** << '\n';  **return** 0;} |

* Example 2: To see why run-time polymorphism is so powerful, try using this ***main()*** instead of previous example:
* This ***main()*** illustrates how random events that occur at run time can be easily handled by using VFs and ***run-time polymorphism***.
* The program executes a ***for*** loop running from ***0*** to ***9***. Each iteration through the loop, you are asked to choose into which type of list- stack or the queue-you want to put a value. According to your answer, the base pointer ***p*** is set to point to the correct object and the current value of ***i*** is stored.
* Once the loop is finished, another loop begins that prompts you to indicate you to indicate from which list to remove a value. Once again, it is your response that determines which list is selected.

|  |  |
| --- | --- |
| **int** **main**(){ **list** \*p;  **queue** q\_ob ;  **stack** s\_ob ;  **char** ch;  **int** i;  **for**(i=0; i <10; i++) {  **cout** << *" Stack or Queue ? (S/Q): "*;  **cin** >> ch;  ch = **tolower**(ch);  **if**(ch == 'q') p = **&**q\_ob ;  **else** p = **&**s\_ob ;  p **->** store(i); } | **cout** << *" Enter T to terminate \n"*;  **for**(;;){ **cout** << *" Remove from Stack or Queue ? (S/Q): "*;  **cin** >> ch;  ch = **tolower**(ch);  **if**(ch == 't') **break** ;  **if**(ch == 'q') p = **&**q\_ob ;  **else** p = **&**s\_ob ;  **cout** << p**->** retrieve() << '\n'; }  **cout** << '\n';  **return** 0;} |

**Difference between Stack and Queue Data Structures :**

|  |  |
| --- | --- |
| Stack: A stack is a *linear data structure* in which elements can be *inserted* and *deleted* only from one side of the list, called the top.   * A stack follows the LIFO ***(Last In First Out)*** principle, i.e., the *element inserted at the last is the first element to come out*. * The *insertion of an element* into stack is called push operation, and *deletion of an element* from the stack is called pop operation. * In stack we always keep track of the last element present in the list with a *pointer* called top. | Queue: A queue is a *linear data structure* in which elements can be *inserted only from one side* of the list called rear, and the elements can be *deleted only from the other side* called the front.   * The queue follows the FIFO ***(First In First Out)*** principle, i.e. the *element inserted at first in the list, is the first element to be removed from the list*. * The *insertion of an element* in a queue is called an enqueue operation and the *deletion of an element* is called a dequeue operation. * In queue we always maintain two pointers, one pointing to the element which was ***inserted at the first and still present in the list*** with the front *pointer* and the second pointer pointing to ***the element inserted at the last*** with the rear *pointer*. |
| C:\Users\User\Downloads\out\geek-stack-1.tif | C:\Users\User\Downloads\out\geek-queue-1.tif |

**13. 5 Generic-Functions & Generic-Classes (GnF & GnC)**

Generic functions and classes (reusable code): We create *generic functions* & *classes* using templates. In a *generic function or class*, the type of data that operated upon is *specified as a parameter*. This allows you to use one function or class with several different types of data without *specific explicit-code* for each different data type.

* A GnF defines a general set of operations that will be applied to various types of data. A GnF has the type of data that ***it will operate upon*** passed to it as a parameter.
* A GnF is the data-independent-code which defines the *nature of the algorithm*. The compiler automatically generates the correct code for the type of data during function execution. By a Gnf the function can ***automatically overload itself***.
* It helps a lot because *many algorithms are logically the same* no matter what type of data is being operated upon. For example, the Quicksort algorithm is applicable for both integers and floats. It is just that the *type of the data* being sorted is different.
* template: A GnF is created using the keyword template. In C++ the keyword template is used to create a ***template (or framework)*** that describes what a function will do, leaving it to the *compiler* to fill in the details as needed. The general form of a template is :

**template <**class *Ttype***> ret\_type func\_name(**parameter list**){** /\* body of function \*/ **}**

* Here ***Ttype*** is a *placeholder name* for a data type used by the function. It can be used within the function definition. The compiler will automatically replace this placeholder with an actual data type during function execution.
* ***Class*** is used to specify a generic type in a template declaration. It is traditional; you can also use the keyword typename.
* Template function: A ***generic function / GnF*** (that is, a function definition preceded by a ***template statement***) is also called a ***template function***.
* Generated function: When the compiler creates a specific version of this function, it is said to have created a ***generated function***.
* Instantiating a function: The act of generating a function is referred to as ***instantiating*** it. Put differently, a *generated function* is a specific instance of a *template function*.
* Generic-Classes (GnC): When you define GnC you create a ***class that defines all algorithms*** used by that class, but the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.
* GnC are useful when a class contains generalizable logic (i.e when data types varies). By using a GnC, you can create a class that will maintain a *queue*, a *linked* *list*, and so on for any type of data.
* The compiler will *automatically generate* the correct type of object based upon the type you specify when the object is created.
* Member functions of a GnC are, themselves, automatically GnF. They need not be explicitly specified as such using template.
* The general form of a GnC declaration is:

**template** <**class** **Ttype** > **class** ***class\_name*** **{ . . .**

**. . . };**

* Here ***Ttype*** is the ***placeholder type name*** that will be specified when a class is instantiated.
* If necessary, you can define more than one generic data type by using a *comma-separated list*.
* Once you have created a GnC, you create a specific instance of that class by using the following general form:

***class\_name <type> ob;***

* Here ***type*** is the ***type name*** of the data that the class will be operating upon.
* One point to remember(?): in the case of GnC we create the object of that generic class using

***class\_name<type> obj\_name ;***

instead of ordinary ***" class\_name obj\_name ;"***. And we can ***access/define*** any *function/member* of that generic class "outside" of it by using:

***template <class Ttype > class\_name<type> :: member(parametr){}***

Here the key point is that ***" class\_name<type>"*** considered the class name instead of ordinary ***" class\_name"*** to define an object of its type or accessing any member outside of it.

* When you create a GnF, you are, in essence, allowing the compiler to generate as many different versions of that function as necessary *to handle the various ways that your program calls that function*.
* Example 1: The following program creates a ***GnF / Function template*** that swaps the values of the two variables it is called with. (Because the general process of exchanging two values is independent of the type of the variables)

|  |  |
| --- | --- |
| **template**<**class** X> **void** swapargs(X &a, X &b){ **X** temp;  temp = a;  a = b;  b= temp ; }  **int** **main**( ){ **int** i=10 , j =20;  **float** x=10 , y =23.3;  **cout** << *" Original i, j: "* << i << ' ' << j << **endl** ;  **cout** << *" Original x, y: "* << x << ' ' << y << **endl** ; | swapargs(i, j); /\* swap integers \*/  swapargs(x, y); /\* swap floats \*/  **cout** << *" Swapped i, j: "* << i << ' ' << j << **endl** ;  **cout** << *" Swapped x, y: "* << x << ' ' << y << **endl** ;    **return** 0; } |

* The keyword ***template*** is used to define a generic function. The line:

***template<class X> void swapargs(X &a, X &b)***

tells the compiler two things: that a template is being created and that a generic definition is beginning.

* Here ***X*** is a generic type that is used as a placeholder.
* After the template portion, function ***swapargs()*** is declared, using ***X*** as the ***data type of the values*** that will be swapped.
* In ***main()***, the ***swapargs()*** function is called using two different types of data: ***integers*** and ***floats***. Because ***swapargs()*** is a generic function, the compiler automatically creates two versions of ***swapargs()***-
* one that will exchange ***integer*** values and
* one that will exchange ***floating-point*** values.
* The ***template portion*** of a GnF definition does not have to be on the same line as the ***function's name***. For example,

**template** <class X>

**void** swapargs(**X** &a, **X** &b) { **X** temp; temp=a; a=b; b=temp; }

* No other statements can occur between the ***template statement*** and the start of the GnF definition. For example, the following fragment will not compile:

**template** <class X>

**int** i; /\* this line causes error \*/

**void** swapargs (**X** &a, **X** &b) { **X** temp ; temp = a; a = b; b= temp ; }

* Instead of using the keyword class, we can use the keyword typename to specify a ***generic type*** in a template definition. Eg:

**template**<typename X> **void** swapargs(**X** &a, **X** &b){ **X** temp; temp = a; a = b; b= temp; }

* The typename keyword can also be used to *specify an unknown type* within a template.
* To define more than one generic data-type with the template statement, use a ***comma-separated list***. For example:

**template**<**class** type1, **class** type2>

**void** myfunc(**type1** x, **type2** y){ **cout**<< x <<' '<< y << **endl**; }

**int** **main**(){ **myfunc**(10 , "hi");

**myfunc** (0.23 , 10L);

**return** 0; }

* The placeholder types ***type1*** and ***type2*** are replaced by the compiler with the data types ***int*** and ***char \**** and ***double*** and ***long***, respectively, when the ***compiler generates*** the specific instances ***(or specific object)*** of ***myfunc()***.
* GnF are similar to overloaded functions except that they are more restrictive.
* For overloaded function different actions can be performed within the body of each function.
* But a GnF must perform the same general action for all versions.
* For example, the following overloaded functions cannot be replaced by a Gnf because they do not do the same thing:

**void** outdata(**int** i){ **cout** << i; }

**void** outdata(**double** d){ **cout** << **setprecision**(10) << **setfill** ('#');

**cout** << d;

**cout** << **setprecision**(6) << **setfill** (' '); }

* Example 2 (overloading GnF / template): Generally a template function overloads itself as needed. But we can ***explicitly overload*** one, too. If you overload a GnF, that overloaded function (our version) overrides (or "hides") the GnF relative to that specific version. For example, consider this version of ***Example 1***:

**template** <**class** X> **void** swapargs(**X** &a, **X** &b) { **X** temp ; temp = a; a = b; b= temp ; }

**void** swapargs (**int** a, **int** b) { **cout** << *" this is inside swapargs (****int*** *,****int*** *)\n"*; } /\* This overrides the GnF swapargs().\*/

**int** **main**( ){ **int** i=10, j =20;

**float** x=10, y =23.3;

**cout** << *" Original i, j: "* << i << ' ' << j << **endl** ; **cout** << *" Original x, y: "* << x << ' ' << y << **endl** ;

swapargs(i, j); /\* calls overloaded swapargs(), because of matched int arguments \*/

swapargs (x, y); /\* swap floats \*/

**cout** << *" Swapped i, j: "* << i << ' '<< j << **endl** ; **cout** << *" Swapped x, y: "* << x << ' ' << y << **endl** ;

**return** 0; }

* When ***swapargs(i,j)*** is called, it invokes the *explicitly overloaded version* of ***swapargs()*** defined in the program (because of ***int*** values). Thus, the compiler does not generate this version of the generic ***swapargs()*** function because the GnF is overridden by the explicit overloading.
* Manual overloading of a template, as shown in this example, allows you to ***tailor a version*** of a GnF to accommodate a special situation.
* In general, if you need to have different versions of a function for different data types, you should use overloaded functions rather than templates.
* Example 3: This program creates a very simple generic singly linked list class. It then demonstrates the class by creating a linked list that stores *characters*.

|  |  |
| --- | --- |
| **template** <**class** data\_t > class list { **data\_t** data ;  list \*next ;  **public** :  **list** ( **data\_t** d);  **void** add(**list** \*node){  node **->** next = **this**;  next = 0; }  **list** \*getnext(){ **return** next ; }  **data\_t** getdata(){ **return** data ; }  };  */\* definition of member function 'list' \*/*  **template** <class data\_t > list <data\_t >:: list ( **data\_t** d) { data = d;  next = 0;} | **int** **main**(){ **list<char>** start ('a');  **list<char>** \*p, \* last ;  **int** i;  */\* build a list \*/*  last = **&**start ;  **for** (i=1; i <26; i++){ p = ***new*** **list <char >(** 'a' + i);  p**->**add ( last );  last = p; }  /\* follow the list \*/  p = **&**start ;  **while**(p) { **cout** << p-**>** getdata();  p = p**->** getnext();}  **return** 0;} |

* The actual data-type stored by the list is ***generic*** in the class declaration. Here objects and pointers are created inside ***main()*** that specify that the ***data-type*** of the list will be char.
* ***Setting data type in object declaration of a generic class-type:*** The desired data type is passed inside the angle brackets in the following declaration:

**list< char >** start('a') ;

* By simply changing the data-type specified "**inside < >**" when list objects are created, you can change the type of data stored by the list. For example, you could create another object that stores integers by using:

**list< int >** int\_start(1) ;

* Use list to store data types that you create: For example, if you want to store address information, use following structure:

**struct** addr { **char** name[40];

**char** street[40];

**char** city[30];

**char** state[3];

**char** zip[12]; }

Then, to use list to generate objects that will store objects of type addr, use: **list< addr >** obj( structvar );

(assuming that ***structvar*** contains a valid ***addr*** structure)

* A template class can have more than one ***generic data type***. Simply *declare all the data types required by the class* in a ***comma-separated list*** within the *template specification*.
* Example 4: the following short example creates a class that uses two generic data types:

|  |  |
| --- | --- |
| **template** <**class** Type\_1 , **class** Type\_2> **class** myclass{ Type1 i;  Type2 j;  **public** : myclass( **Type1** a, **Type2** b) { i = a; j = b; }  **void** show() { cout << i << ' ' << j << '\n'; }  }; | **int** **main**(){ **myclass< int, double >** ob1 (10 , 0.23) ;  **myclass<char , char \*>** ob2('X', *" This is a test "*);  ob1.show(); /\* show int , double \*/  ob2.show(); /\* show char , char \*\*/  **return** 0; } |
| This program produces the following output: ***10 0.23***  ***X This is a test*** | |

* The program declares two types of objects. ***ob1*** uses ***integer*** and ***double*** data. ***ob2*** uses a ***character*** and a ***character*** ***pointer***.
* For both cases, the compiler automatically generates the appropriate data and functions for each object.

Note

1. C++ provides a library that is built upon ***template classes***. This library is usually referred to as the Standard Template Library, or STL for short.
2. STL provides generic versions of the most *commonly used algorithms and data structures*.

**13.6 EXCEPTION HANDLING**

Exception handling (resilient code): Exception handling is the subsystem of C++ that allows us to handle errors that occur at run time in a *structured and controlled way*. By exception handling, your program can *automatically* invoke an *error handling routine* when an error occurs.

* Exception handling is C++'s *built-in error handling mechanism*. Mostly used to manage and respond to run-time errors. C++ exception handling is built upon three keywords: try, catch, and throw.
* Generally the program statements that you want to monitor for exceptions are contained in a try block.
* If an ***exception (i.e., an error) occurs*** within the try block, it is thrown using throw.
* The exception is caught, using catch, and processed.
* General Form and **try-catch** blocks: The general form of try and catch are:

**try**{ /\* try block \*/ }

**catch**(type1 arg){ /\* catch block \*/ }

**catch**(type2 arg){ /\* catch block \*/ }

**catch**(type3 arg){ /\* catch block \*/ }

. . .

**catch**(typeN arg){ /\* catch block \*/ }

* try: The try block must contain the portion of your program that you want to *monitor for errors*. This can be a few statements within one function or all-codes by enclosing the ***main()*** function within a try block ***(which causes the entire program to be monitored).***
* Any statement that ***throws an exception*** must have been executed from within a try block.
* A function called from within a try block can also ***throw an exception***.
* catch: Any exception must be caught by a catch statement that immediately follows the try statement that throws the exception. Catch statement processes the exception.
* ***Any type of data*** can be caught by catch. Class types are frequently used as exceptions.
* There can be more than one catch associated with a try. The catch that is used is ***determined*** ***by*** the type of the ***exception***. I.e, if the data type specified by a catch ***matches the data type of the exception***, that catch is executed (and all others are bypassed).
* When an exception is caught, arg will receive its value. If you don't need access to the exception itself, specify only type in the catch clause-arg ***is optional***.
* General form of the **throw**: The general form of the throw statement is: ***throw exception ;***
* ***throw*** must be executed either from within the try block proper or from any function that the code ***within the block calls*** (directly or indirectly).
* ***exception*** is the value thrown. If you throw an exception for which there is no applicable catch statement, an abnormal program termination might occur.
* In standard C++, throwing an unhandled exception causes the standard library function ***terminate()*** to be invoked. By default, ***terminate()*** calls ***abort()*** to stop your program.
* You can specify your own ***termination handler*** by referring to your compiler's library reference for details.
* Catch all exceptions with ellipsis ". . ." : To *catch all exceptions* instead of just a certain type, use following form of catch:

**catch(...){ /\* process all exceptions \*/ }**

Here the ellipsis matches any type of data. ***[ ". . ." called ellipsis. It indicates an intentional omission of a word/whole-line/text-section without altering original meaning.]***

|  |  |
| --- | --- |
| * Appling restrictions to exceptions: | * We can restrict the *type of exceptions* that a function can throw back to its caller. * We can control what *type of exceptions* a function can throw outside of itself. * We can also prevent a function from throwing any exceptions whatsoever. |

* To apply these restrictions, you must add a throw ***clause*** to the function definition. The general form is:

**ret\_type** func\_name(***arg\_list***) throw(***type\_list***)**{** /\* exceptions \*/ **}**

* Here only those data types contained in the *comma-separated* type-list may be thrown by the function.
* When a function attempts to throw a disallowed exception the standard library function ***unexpected()*** is called, this causes the ***terminate()*** function to be called, which causes abnormal program termination.
* ***For own termination handler:*** need to refer to compiler's documentation for directions on how this can be accomplished.
* If you don't want a function to be able to throw any exceptions, use an empty list.
* Rethrowing exceptions: To rethrow an expression from within an exception handler: call ***throw***, by itself with no exception. This causes the current exception to be passed on to an outer ***try/catch*** sequence.
* Example 1: (Execution process of Exception Handling): Following shows the way C++ exception handling operates:

**int** **main**(){**cout** << " ***start \n***";

**try**{ /\* start a try block \*/

**cout** << " ***Inside try block \n***";

**throw** 10; /\* throw an error \*/

**cout** << " ***This will not execute*** "; */\* not execute, control transferred to "catch" due to "throw 10"\*/*

}

**catch**(**int** i){ /\* beginning catch block: catch an error \*/

**cout** << " ***Caught One ! Number is:*** ";

**cout** << i << "\n"; }

**cout** << "end ";

**return** 0; }

|  |  |
| --- | --- |
| This program displays the output: | ***start***  ***Inside try block***  ***Caught One! Number is: 10***  ***end*** |

|  |  |
| --- | --- |
| * There is: | * a ***try*** block containing three statements and * a ***catch(int i)*** statement that processes an integer exception. |

* Within the ***try*** block, only two of the three statements will execute: the first ***cout*** statement and the ***throw***. Once an exception has been thrown, ***control passes to the*** ***catch*** expression and the ***try*** block is terminated. The ***cout*** statement following the ***throw*** will *never execute*.
* i.e. ***catch*** is not called, rather, ***program execution is transferred to it***. (The stack is automatically reset as needed to accomplish this.)
* After the ***catch*** statement executes, ***program control*** continues with the statements following the ***catch***.
* Often, however, a ***catch block*** ***will end with*** a call to ***exit()***, ***abort()***, or some other function that causes *program termination* because exception handling is frequently used ***to handle catastrophic errors***.
* The type of the exception must match the type specified in a ***catch*** statement. Considering Example 1, following won't work .

**catch**(**double** i){ */\* '****catch'*** *is double type: won't work for an int exception \*/*

**cout** << " ***Caught One ! Number is:*** ";

**cout** << i << "\n"; }

* This program produces the following output because the integer ***exception*** will not be caught by a double ***catch*** statement.

|  |  |
| --- | --- |
| This program displays the output: | ***start***  ***Inside try block***  Abnormal program termination |

* Example 2: An exception can be thrown ***from a statement that is outside*** the ***try*** ***block*** as long as the ***statement is within a function that is called from within the try block***. For example, this is a valid program:

|  |  |
| --- | --- |
| **void** Xtest(**int** test) { **cout** << " Inside Xtest , test is: " << test << "\n";  **if**(test) **throw** test ; } | |
| **int** **main**(){ **cout** << " start \n";  **try**{ */\* throwing by the function Xtest : calling function within try block \*/*  Xtest(0);  Xtest(1);  Xtest(2);} */\* it is also an exception but never thrown or executed \*/*  **catch**(**int** i{ **cout** << " Caught One ! Number is: ";  **cout** << i << "\n"; }  **cout** << "end ";  **return** 0; } | output: ***start***  ***Inside try block***  ***Inside Xtest, test is: 0***  ***Inside Xtest, test is: 1***  ***Caught One! Number is: 1***  ***end*** |

* ***Xtest(2)*** is also ***exception*** but never thrown because of control transferred to "***catch***" after throwing ***1*** as ***exception***.
* Example 3: [To avoid "***error skipping***" as ***Xtest(2)*** in Example 2] A ***try*** ***block*** can be localized to a function. In this case, each time the function is entered, the exception handling relative to that function is reset. For example:

|  |  |
| --- | --- |
| **void** Xhandler(**int** test){ **try** { **if**( test )  **throw** test ; }  **catch**(**int** i){ **cout** << " Caught One ! Ex. #: " << i << '\n'; }  } | |
| **int** **main**(){ **cout** << " start \n";  Xhandler (1);  Xhandler (2);  Xhandler (0);  Xhandler (3);  **cout** << "end ";  **return** 0;} | output: ***start***  ***Caught One! Ex. #: 1***  ***Caught One! Ex. #: 2***  ***Caught One! Ex. #: 3***  ***end*** |

* ***try block*** is not inside ***main()***, instead ***try-catch*** blockscontaining function ***Xhandler()*** is called from ***main()***.
* As you can see, three exceptions are thrown. After each exception, the function returns. When the function is called again, the ***exception handling is reset***.
* Example 4: More than one catch associated with a try. Each catch must catch a different type of exception (two or more catch with same data-type returns error). For example, consider Example 3 with the following ***Xhandler()*** [catches both integers and strings]:

**void** Xhandler(**int** test){ **try** { **if**(test) **throw** test ;

**else** **throw** "value is zero" }

**catch**(**int** i){ **cout** << " Caught One ! Ex. #: " << i << '\n'; }

**catch**(**char** \*str){ */\* \*str is used to print ' value is zero ' \*/*

**cout** << " Caught a string :";

**cout** << str << '\n';}

}

* In general, ***catch*** expressions are checked in the order in which they occur in a program. Only a matching statement is executed. All other ***catch blocks*** are ignored.
* Example 5: Following catches all exceptions [with ellipsis . . .] using ***catch(...)***:

|  |  |  |
| --- | --- | --- |
| **void** Xhandler (**int** test ) { **try** { **if**( test ==0) **throw** test ; */\* throw int \*/*  **if**( test ==1) **throw** 'a'; */\* throw char \*/*  **if**( test ==2) **throw** 123.23; */\* throw double \*/*  }  **catch** (**...**){ */\* catch all exceptions \*/*  **cout** << " Caught One !\n"; }  } | **int** **main**(){ **cout** << " start \n";  Xhandler(0);  Xhandler(1);  Xhandler(2);  **cout** << "end ";  **return** 0; } | output:  start  Caught One!  Caught One!  Caught One!  end |

* All three ***throws*** were caught using the one ***catch*** statement.
* Example 6: Use ***catch(...)*** as the last catch of a cluster of catches [as last catch block for miscellaneous errors ]. In this capacity it provides a useful default or "catch all" statement. For example, this slightly different version of the preceding program explicitly catches integer exceptions but relies upon ***catch(...)*** to catch all others:

|  |  |  |
| --- | --- | --- |
| **void** Xhandler (**int** test ) { **try** { **if**( test ==0) **throw** test ; */\* throw int \*/*  **if**( test ==1) **throw** 'a'; */\* throw char \*/*  **if**( test ==2) **throw** 123.23; */\* throw double \*/*  }  **catch** (**int i**){ */\* catch an int exception \*/*  **cout** << " Caught" << i << '\n'; }  **catch** (**...**){ */\* all other exceptions \*/*  **cout** << " Caught One !\n"; }  } | **int** **main**(){ **cout** << " start \n";  Xhandler(0);  Xhandler(1);  Xhandler(2);  **cout** << "end ";  **return** 0; } | output:  start  Caught 0  Caught One!  Caught One!  end |

* By catching all exceptions, you prevent an unhandled exception from causing an ***abnormal program termination***.
* Example 7: ***ret\_type func\_name(arg\_list) throw(type\_list){ /\* exceptions \*/ }*** to restrict the types of exceptions that can be thrown from a function:

|  |  |
| --- | --- |
| **void** Xhandler(**int** test ) **throw**(**int**, **char**, **double**) {  **if**( test ==0) **throw** test ; /\* throw int \*/  **if**( test ==1) **throw** 'a'; /\* throw char \*/  **if**( test ==2) **throw** 123.23; /\* throw double \*/ } | **int** **main**(){ **cout**<< "start \n";  **try**{ Xhandler(0); } */\* 1 and 2 also \*/*  **catch**(**int** i) { **cout** << " Caught int \n";}  **catch** ( **char** c) { **cout** << " Caught char \n"; }  **catch** ( **double** d) { **cout** << " Caught double \n"; }  **cout** << "end ";  **return** 0; } |

* In this program, the function ***Xhandler()*** can throw only ***integer***, ***character***, and ***double*** exceptions. If it attempts to throw any other type of exception, an abnormal program termination will occur. (That is, ***unexpected()*** will be called.) To see an example of this, remove ***int*** from the list and retry the program.
* A function can only be restricted in what types of exceptions it throws back to the ***try block*** that called it. That is, ***a try block within a function can thrown any type of exception so long as it is caught within that function***.
* The restriction applies only when ***throwing an exception*** out ***of the function***.
* Example 8: The following change to ***Xhandler()*** prevents it from throwing any exceptions:

**void** Xhandler(**int** test ) throw(){ **if**( test ==0) **throw** test ;

**if**( test ==0) **throw** 'a';

**if**( test ==2) **throw** 123.23; }

* The above statements no longer work . Instead , they will cause an ***abnormal program termination*** .
* Example 9: The reason for rethrow an exception is to allow ***multiple handlers*** access to the exception. For example, perhaps one exception handler manages one aspect of an exception and a second handler copes with another.
* An ***exception*** can only be ***rethrown*** from within a ***catch*** ***block*** (or from any function called from within that block).
* When you rethrow an exception, it will not be recaught by the ***same*** ***catch*** ***statement***. It will propagate to an ***outer*** ***catch*** statement.
* The following program illustrates rethrowing an exception. It rethrows a ***char \**** exception.

|  |  |  |
| --- | --- | --- |
| **void** Xhandler() {  **try** { **throw** " hello "; } /\* throw char \* \*/  **catch**( **const** char \*) { /\* catch char \* \*/  **cout** << " char \* inside Xhandler \n";  **throw** ; /\* rethrow char \* \*/ }  } | **int** **main**(){ **cout** << " start \n";  **try** { Xhandler (); }  **catch** ( const char \*) {  **cout** << "char \* inside main \n"; }  **cout** << "end ";  **return** 0; } | output:  start  char \* inside Xhandler  char \* inside main  end |

**13.7 Handling *exceptions* thrown by *new***

Behavior of new as specified by Standard C++: In early C++, new returned null on failure. In later version new caused an exception on failure. Finally, it was decided that a new failure will generate an exception by default, but that a null pointer could be returned instead, as an option.

* Allocation exceptions with ***new*** and ***xalloc*** or ***bad\_alloc***: In Standard C++, when an ***allocation request cannot be honored***, ***new*** throws a ***bad\_alloc*** (***xalloc*** in older versions) exception. If you don't catch this exception, your program will be terminated.
* It is good for short programs but in real applications you ***must catch this exception*** and process it in some rational manner.
* To have access to this exception, you must include the header **<new>** in your program.
* Returning old fashioned null In Standard C++: It is also possible to have new return null instead of throwing an exception when an allocation failure occurs. This form of new is : ***p\_var =new(nothrow) type ;***
  + Here ***p\_var*** is a pointer variable of type.
* The ***nothrow*** form of new works like the original version of new from years ago. Since it returns null on failure, it can be ***"dropped into"*** older code and *avoid exception handling*. Useful when compiling older code with a modern C++ compiler.
* It is also valuable when you are replacing calls to ***malloc()*** with ***new***.
* Example 1: Here is an example of ***new*** that uses a ***try/catch block*** to monitor for an allocation failure.

|  |  |
| --- | --- |
| #include <iostream>  #include <new>  using namespace std;  **int** **main**(){ **int** \*p;  **try**{ p = **new** **int**; } /\* allocate memory for int \*/  **catch** (bad\_alloc xa){ **cout** << " Allocation failure .\n";  **return** 1; } | **for**(\*p = 0; \*p < 10; (\*p)++) **cout** << \*p << " " ;  **delete** p; // free the memory  **return** 0; } |

* Here if an allocation failure occurs, it is caught by the ***catch*** statement.
* Example 2: Since the previous program is unlikely to fail under any normal circumstance, the following program demonstrates ***new's*** exception-throwing capability by forcing on allocation failure. It does this by allocating memory until it is exhausted.

**int** **main**(){ **double** \*p;

**do**{ **try**{ p = **new** **double**[100000]; } */\* this will eventually run out of memory \*/*

**catch**( bad\_alloc xa ){ **cout** << " Allocation failure .\n";

**return** 1;}

}**while** (p);

**return** 0;}

* Example 3: Following shows the use of ***new(nothrow)*** alternative. It reworks the Example 2 and forces an allocation failure.

**int** **main**(){ **double** \*p;

**do**{ p = **new(nothrow)** **double**[100000]; */\* this will eventually run out of memory \*/*

**if**(p) cout << "Allocation ok \n";

**else** cout << "Allocation error \n";

}**while** (p);

**return** 0;}

* When you use the ***nothrow*** approach, you must check the ***pointer returned by*** ***new*** after each allocation request.
* ***RTTI*** allows you to identify the type of an object during the execution of your program.
* The casting operators give you safer, more controlled ways to cast. As you will see, one of the casting operators, dynamic cast, relates directly to RTTI.

**13.8 RTTI (run-time type identification)**

* ***RTTI*** is not found in non-polymorphic languages such as C. In such languages, there is no need for *run-time type information* because the type of each object is known at *compile time* (i.e., when the program is written).
* In polymorphic languages such as C++, there can be situations in which the type of an object is unknown at compile time because the precise nature of that object is not determined until the program is executed. For example: A ***base class pointer*** can be used to point to objects of the *base class* or to any object *derived* from that base. This determination must be made at run-time, using ***RRTI***.

|  |  |
| --- | --- |
| * C++ implements polymorphism through : | * The use of ***class hierarchies***, * ***Virtual functions***, and * ***Base class pointers***. |

* To obtain an object's type, use typeid and must include the header **<typeinfo>**. The most common form of typeid is:

***typeid(object)***

* Here object is the object whose type you will be obtaining.
* typeid returns a reference to an object of type type\_info that describes the type of object defined by object.

|  |  |
| --- | --- |
| * The type\_info class defines these public members: | * ***bool*** **operator**==( ***const*** type\_info &ob); * ***bool*** **operator**!=( ***const*** type\_info &ob); * ***bool*** **before**( ***const*** type\_info &ob); * ***const*** **char**\***name**(); |

* The overloaded **"=="** and **"!="** provide for the comparison of types.
* The ***before()*** function returns ***true*** if the invoking object is before the object *used as a parameter in collation order*. *(Internal use only. Its return value has nothing to do with inheritance or class hierarchies.)*
* The ***name()*** function *returns a pointer* to the *name of the type*.
* ***The most important use of*** typeid ***is:*** its application through a pointer of a polymorphic base. Using typeid you can determine at *run-time* the ***type of the object*** that is being pointed to by a base pointer.
* typeid will automatically return the ***type of the actual object being pointed to***, which can be a base object or a derived object from that base. [Note: A base pointer can point to a ***base object*** or any ***derived object*** from that ***base***.]

|  |  |
| --- | --- |
| * The same applies to references: | * When typeid is applied to a ***reference to an object*** of a polymorphic class, it will return the type of the object actually being referred to, which can be of a derived type. * When typeid is applied to a non-polymorphic class, the base type of the pointer or reference is obtained. |

* ***Another form of*** typeid***:*** This form of typeid takes a ***type name*** as its argument: **typeid(** type\_name **)**
* It is used to obtain a type\_info ***object*** that describes the specified type so that it can be used in a ***type comparison statement***.
* bad\_typeid ***exception***: Because typeid is commonly applied to a dereferenced pointer (i.e., one to which the **\*** operator has been applied), a special exception has been created to handle the situation in which the pointer being dereferenced is null. In this case, typeid throws a bad\_typeid exception.
* Example 1: The following program demonstrates typeid. It first obtains type information about one of C++'s built-in types, int. It then displays the types of objects pointed to by p, which is a pointer of type BaseClass.

|  |  |  |
| --- | --- | --- |
| #include <iostream>  #include <typeinfo>  using namespace std;  **class** BaseClass {  **virtual** **void** f(){ } */\* BaseClass polymorphic \*/*  /\* ... \*/ };  **class** Derived1 : **public** BaseClass{ /\* ... \*/ };  **class** Derived2 : **public** BaseClass{ /\* ... \*/ };  **int** **main**(){ **int** i;  **BaseClass** \*p, baseob;  **Derived1** ob1;  **Derived2** ob2; | */\* First, display type name of a built -in type. \*/*  **cout** << " Typeid of i is " << **typeid** (i). **name** () << **endl** ;  */\* Demonstrate typeid with polymorphic types. \*/*  p = &baseob ;  **cout** << "p is pointing to an object of type" << **typeid**(\*p).**name**() << **endl**;  p = &ob1;  **cout** <<"p is pointing to an object of type" << **typeid**(\*p).**name**() << **endl** ;  p = &ob2;  **cout** <<"p is pointing to an object of type" << **typeid**(\*p).**name**() << **endl** ;  **return** 0;} | |
| Output [may vary depending on compiler] : | | *Typeid of i is* ***int***  *p is pointing to an object of type* ***class BaseClass***  *p is pointing to an object of type* ***class Derived1***  *p is pointing to an object of type* ***class Derived2*** |

* when ***typeid*** is applied to a ***base pointer of a polymorphic type***, the type of object pointed to will be determined at run time, as the output produced by the program shows.
* Example 2 (***When objects are passed to functions by*** reference): In the following program, the function ***WhatType()*** declares a reference parameter to objects of type ***BaseClass***. This means that ***WhatType()*** can be passed ***references to objects*** of type ***BaseClass*** or any class derived from ***BaseClass***. When the ***typeid*** operator is applied to this parameter, it returns the actual type of the object being passed.

|  |  |
| --- | --- |
| **class** BaseClass {  **virtual** **void** f(){ } */\* BaseClass polymorphic \*/*  /\* ... \*/ };  **class** Derived1 : **public** BaseClass{ /\* ... \*/ };  **class** Derived2 : **public** BaseClass{ /\* ... \*/ };  */\* Demonstrate typeid with a reference parameter.\*/*  **void** WhatType( **BaseClass** &ob){  **cout** << "ob is referencing an object of type " << **typeid**(ob).**name**() << **endl** ; } | **int** **main**(){ **BaseClass** baseob ;  **Derived1** ob1;  **Derived2** ob2;  WhatType( baseob );  WhatType(ob1);  WhatType(ob2);  **return** 0;} |

Output: *ob is pointing to an object of type* ***class BaseClass***

*ob is pointing to an object of type* ***class Derived1***

*ob is pointing to an object of type* ***class Derived2***

* Example 3: Since the ***type\_info*** object returned by ***typeid*** overloads the ***==*** and ***!=*** operators, this too is easy to know whether the type of *one object matches that of another*. The following program demonstrates the use of these operators.

|  |  |
| --- | --- |
| **class** X { **virtual** **void** f(){} };  **class** Y { **virtual** **void** f(){} }; | **int** **main**(){ **X** x1, x2;  **Y** y1;  **if**(typeid(x1) == typeid(x2)) **cout** << "x1 and x2 are same types \n";  **else** **cout** << "x1 and x2 are different types \n";  **if**( typeid (x1) != typeid (y1)) **cout** << "x1 and y1 are different types \n";  **else** **cout** << "x1 and y1 are same types \n";  **return** 0;} |
| output  ***x1 and x2 are same types***  ***x1 and y1 are different types*** |

* Example 4: The ***typeid*** operator can be applied to ***template classes***. For example, consider the following program. It creates a hierarchy of template classes that store a value.
* The VF ***get\_val()*** returns a value that is defined by each class.

|  |  |  |
| --- | --- | --- |
| * ***Num*** for value of the number itself | * ***Square*** for square of the number | * ***Sqr\_root*** for square root of the number. |

* Objects derived from ***Num*** are generated by ***generator()*** function. The ***typeid*** determines the type of the generated object .

|  |  |  |
| --- | --- | --- |
| #include <iostream>  #include <typeinfo>  #include <cmath>  #include <cstdlib>  using namespace std;  **template** <**class** T> **class** Num{  **public** :  **T** x;  Num (**T** i) { x = i; }  **virtual T** get\_val () { **return** x; }  }; | **template** <**class** T> **class** Square : **public** Num <**T**> {  **public** :  Square(**T** i) : Num <**T**>(i){ }  **T** get\_val(){ **return** (**this ->** x)\*(**this ->**x);  */\* Edited: main book 'return x\*x;'* \*/ }  }; | |
| **template** <**class** T> **class** Sqr\_root : **public** Num <**T**> {  **public** :  Sqr\_root(**T** i) : Num <**T**>(i){ }  **T** get\_val(){ **return** **sqrt**((**double**) **this ->** x);  */\* Edited: main book sqrt( (double) x);' \*/* }  }; | |
| */\* A Random selection factory for objects derived from Num : for run-time selection.\*/*  **Num** <**double**> \*generator(){  **switch**( **rand**() % 2){  **case** 0: **return** **new** **Square** <**double**> ( **rand**() % 100) ;  **case** 1: **return** **new** **Sqr\_root** <**double**> ( **rand**() % 100) ; }  **return** **NULL** ; } | | |
| int main(){  **Num** <**double**> ob1(10), \*p1;  **Square** <**double**> ob2(100.0) ;  **Sqr\_root** <**double**> ob3(999.2) ;  **int** i;  **cout** << **typeid**(ob1).**name**() << **endl** ;  **cout** << **typeid**(ob2).**name**() << **endl** ;  **cout** << **typeid**(ob3).**name**() << **endl** ;  **if**( **typeid**(ob2) == **typeid**(**Square** <**double**>)) **cout** << "is Square <**double**>\n";  p1 = **&**ob2 ;  **if**( **typeid**(\*p1) != **typeid**(ob1)) **cout** << " Value is: " << p1 **->** get\_val();  **cout** << "\n\n";  **cout** << "Now , generate some Objects .\n";  **for** (i=0; i <10; i++){  p1 = generator(); */\* get next object \*/*  **if**(**typeid**(\*p1) == **typeid**(Square <**double**>)) **cout** << "Square object :";  **if**(**typeid** (\*p1) == **typeid**(Sqr\_root <**double**>)) **cout** << "Sqr\_root object:";  **cout** << "Value is:" << p1 -> get\_val();  **cout** << **endl** ;  }  **return** 0;} | | output :  *class Num<double>*  *class Square<double>*  *class Sqr root<double>*  *is Square<double>*  *Value is: 10000*  *Now, generate some Objects.*  *Sqr root object: Value is: 8.18535*  *Square object: Value is: 0*  *Sqr root object: Value is: 4.89898*  *Square object: Value is: 3364*  *Square object: Value is: 4096*  *Sqr root object: Value is: 6.7082*  *Sqr root object: Value is: 5.19615*  *Sqr root object: Value is: 9.53939*  *Sqr root object: Value is: 6.48074*  *Sqr root object: Value is: 6* |

* The example shown in main book ***will not compile*** . Reason: Consider a template class Derived with a template base class:

**template** <**typename** T> **class** Base { **public**: **int** d; };

**template** <**typename** T> **class** Derived : **public** Base<**T**>{ **void** f(){**this->**d = 0;} };

* ***this*** has type ***Derived<T>***, a type which depends on ***T***. So ***this*** has a ***dependent type***. So ***this->d*** makes ***d*** a ***dependent name***. Dependent names are looked-up in the context of the template definition as *non-dependent* names and in the context of instantiation.
* Without ***this->***, the name ***d*** would only be ***looked-up as a non-dependent name, and not be found***.
* Another solution is to declare ***d*** in the *template definition* itself:

**template** <**typename** T> **class** Derived : **public** Base<**T**> { **using** Base::d;

**void** f(){ d = 0; } };

|  |  |
| --- | --- |
| Note: | ***RTTI*** is not common in every program. However, when you are working with polymorphic types, it allows you to know what type of object is being operated upon in any given situation. |

**13.9 C++ casting operators**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C++ has four new casting operators. They are: | 1. ***dynamic\_cast*** | 1. ***const\_cast*** | 1. ***reinterpret\_cast*** | 1. ***static\_cast*** |

**13.9.1 dynamic\_cast:**

* The dynamic cast is related to ***RTTI***, it performs a run-time cast that verifies the *validity of a cast*. Dynamic cast can be used to cast *one type of pointer into another* or ***one type of reference into another***. The general form of dynamic cast is:

***dynamic\_cast<***target\_type***>(expr)***

* Here ***target\_type*** specifies the ***target type of the cast*** and ***expr*** is the ***expression being cast into the new type***.
* The ***target\_type*** must be a pointer or reference type, and " ***expr*** " the expression being cast must *evaluate* to a pointer or reference.
* If the ***cast is invalid*** during the execution of ***dynamic\_cast***, the cast fails.
* The purpose of **dynamic\_cast** is to perform casts on polymorphic types: For example, given the two *polymorphic classes* **B** and **D**, with **D** ***derived from*** **B**.
* A **dynamic\_cast** can always cast a **D\*** ***pointer*** into a **B\*** ***pointer***. Because a ***base pointer*** can always ***point to a derived***.
* But a **dynamic\_cast** can cast a **B\*** ***pointer*** into a **D\*** ***pointer*** only if the *object being pointed to* actually is a **D** object.
* In general, ***dynamic\_cast*** will succeed if the pointer (or reference) being cast is a pointer (or reference) to either an *object of the target type* or an ***object derived from the target type***. Otherwise, the cast will fail.
* If the cast fails, ***dynamic\_cast*** evaluates to null if the cast involves pointers.
* If a ***dynamic\_cast*** on reference types fails, a ***bad\_cast*** ***exception*** is thrown.
* Example of Successful cast: Assume that ***Base*** is a polymorphic class and that ***Derived*** is derived from ***Base***.

**Base** \*bp, b\_ob;

**Derived** \*dp, d\_ob;

bp = &d\_ob ; /\* ***base pointer*** points to ***Derived object*** : ***ok*** \*/

dp = **dynamic\_cast** < **Derived** **\*** >(bp);

**if**(dp) **cout** << " Cast OK ";

* Here the cast from the base pointer ***bp*** to the derived pointer ***dp*** works because ***bp*** is actually pointing to a Derived object. It displays "***Cast OK***".
* Example of Unsuccessful cast: In the next fragment, the cast fails because ***bp*** is ***pointing to a Base object*** and it is illegal to cast a *base object* into a *derived object*. It displays "***Cast Fails***".

bp = &b\_ob ; /\* ***base pointer*** points to ***Base object*** : ***Wrong*** \*/

dp = **dynamic\_cast** < **Derived \***>(bp);

**if** (!dp) **cout** << " Cast Fails ";

Notes:

The ***dynamic\_cast*** operator can sometimes be used instead of ***typeid***. Consider the previous example. The following fragment will assign ***dp*** the ***address of the object pointed to*** by ***bp*** ***iff*** the *object is really a Derived object*.

**Base** \*bp;

**Derived** \*dp;

// ...

**if(** **typeid**(\*bp) == **typeid**(Derived) **)** dp = (Derived \*) bp;

* Here a C-style cast is used to perform the cast. It's safe because the if checks the ***legality of the cast*** using typeid before the cast actually occurs.
* The better way to accomplish this is to replace the typeid operators and if statement with this ***dynamic\_cast***:

*dp* ***= dynamic\_cast < Derived \*>*** *(bp);*

* After this statement executes **dp** will contain either a null or a pointer to an ***object of type*** ***Derived***. Because ***dynamic\_cast*** succeeds only if the object being cast is either
* already an object of the *target type* or
* an object *derived* from the *target type*,
* Since ***dynamic\_cast*** succeeds ***only if the cast is legal***, it can *simplify* the logic in certain situations.

**13.9.2 const\_cast, reinterpret\_cast and static\_cast**

General forms of other three casting operators are:

***const\_cast***< target\_type > (expr)

***reinterpret\_cast***< target\_type > (expr)

***static\_cast***< target\_type > (expr)

* Here ***target\_type*** specifies the target type of the cast and ***expr*** is the ***expression being cast*** into the *new type*.
* const\_cast: The ***const\_cast*** operator is used to explicitly override const ***and/or volatile*** in a *cast*. The most common use of ***const\_cast*** is to remove ***const***-ness.
* The ***target\_type*** must be the *same as* thesource type except for the alteration of its ***const*** or ***volatile*** attributes.
* static\_cast: The ***static\_cast*** operator performs a non-polymorphic cast. For example, it can be used to ***cast a base class pointer*** into a ***derived class pointer***.

|  |  |
| --- | --- |
| * It can also be used for any standard conversion. | * No run-time checks are performed. |

|  |  |
| --- | --- |
| * reinterpret\_cast: The ***reinterpret\_cast*** operator: | * Changes one ***pointer type*** into ***another*** (mainly different, ***pointer type***). * It can also change a *pointer* into an *integer* and an *integer* into a *pointer*. |

* A ***reinterpret\_cast*** should be used for casting ***inherently incompatible pointer types***.

Note: Only ***const\_cast*** can cast away ***const***-ness. That is, neither ***dynamic\_cast***, ***static\_cast***, nor ***reinterpret\_cast*** can alter the ***const***-ness of an object.

* Example 1: The following program demonstrates ***dynamic\_cast***:

|  |  |  |  |
| --- | --- | --- | --- |
| **class** Base{ **public** : **virtual** **void** f(){ **cout** << "Inside Base \n"; } };  **class** Derived : **public** Base{ **public** : **void** f() {  **cout** << " Inside Derived \n"; } };  **int** **main**() { **Base** \*bp , b\_ob ;  **Derived** \*dp , d\_ob ;  dp = **dynamic\_cast** < Derived \*> (& d\_ob );  **if**(dp) { **cout** << " Cast from *Derived \* to Derived* \* OK .\n"; dp **->**f(); }  **else** **cout** << " Error \n";  **cout** << **endl** ;  bp = **dynamic\_cast** < Base \*> (& d\_ob );  **if**(bp) { **cout** << " Cast from *Derived \* to Base* \* OK .\n"; bp **->**f(); }  **else** **cout** << " Error \n";  **cout** << **endl** ;  bp = **dynamic\_cast** < Base \*> (& b\_ob );  **if**(bp) { **cout** << " Cast from *Base \* to Base* \* OK .\n"; bp **->**f();}  **else** **cout** << " Error \n";  **cout** << **endl** ;  */\*following is not ok\*/*  dp = **dynamic\_cast** < Derived \*> (& b\_ob );  **if**(dp) **cout** << " Error \n";  **else** **cout** << " Cast from *Base\* to Derived\** not OK .\n";  **cout** << **endl** ; | | bp = &d\_ob ; */\* base pointer bp points to Derived object \*/*  dp = **dynamic\_cast** < Derived \*> (bp);  **if**(dp){**cout** << " Casting *bp to a Derived* \* OK .\n"  <<" because bp is really pointing \n"  <<"to a Derived object .\n";  dp **->**f();}  **else** **cout** << " Error \n";  **cout** << **endl** ;  bp = &b\_ob ; */\* bp points to Base object \*/*  dp = **dynamic\_cast** < Derived \*> (bp); */\** ***NOT OK:*** *\*/*  **if**(dp) **cout** << " Error \n";  **else** { **cout** << "Now casting *bp to a Derived* \*\n"  << "is not OK because bp is really \n"  <<" pointing to a Base object .\n"; }  **cout** << **endl** ;  dp = &d\_ob ; */\* dp points to Derived object \*/*  bp = **dynamic\_cast** < Base \*> (dp);  **if**(bp){ **cout** << " Casting *dp to a Base \** is OK .\n";  bp **->**f(); }  **else** **cout** << " Error \n";  **return** 0; } | |
| Output:  Cast from Derived \* to Derived \* OK.  Inside Derived  Cast from Derived \* to Base \* OK.  Inside Derived  Cast from Base \* to Base \* OK. | Inside Base  Cast from Base \* to Derived \* not OK.  Casting bp to a Derived \* OK.  because bp is really pointing  to a Derived object.  Inside Derived | | Now casting bp to a Derived \*  is not OK because bp is really  pointing to a Base object.  Casting dp to a Base \* is OK.  Inside Derived |

* Cast from ***base\**** to ***derived\**** means: derived pointer points to a base object via ***derived\**** cast.
* Example 2: The following example illustrates how a ***dynamic\_cast*** can be used to replace ***typeid***.

|  |  |  |
| --- | --- | --- |
| # include <iostream >  # include <typeinfo >  using namespace std;  **class** **Base** { **public** : **virtual** **void** f() {} };  **class** **Derived** : **public** Base { **public** : **void** derivedOnly(){ **cout** << "Is a Derived Object \n";} }; | | Output :  *Cast from Base to Derived failed.*  *Is a Derived Object*  *Cast from Base to Derived failed.*  *Is a Derived Object* |
| **int** **main**() { **Base** \*bp , b\_ob ;  **Derived** \*dp , d\_ob ;  */\* use typeid \*/*  bp = & b\_ob ;  **if**( **typeid** (\* bp) == **typeid** ( **Derived** )){dp = ( **Derived \***) bp;  dp -> derivedOnly( );}  **else** **cout** << " Cast from Base to Derived failed .\n";  bp = & d\_ob ;  **if**( **typeid** (\* bp) == **typeid** ( **Derived** )){dp = ( **Derived** **\***) bp;  dp -> derivedOnly( );}  **else** **cout** << "Error , cast should work !\n"; | */\* use dynamic\_cast \*/*  bp = &b\_ob ;  dp = **dynamic\_cast** < **Derived** \*> (bp);  **if**(dp) dp -> derivedOnly ();  **else** **cout** << " Cast from Base to Derived failed .\n";  bp = & d\_ob ;  dp = **dynamic\_cast** < **Derived \***> (bp);  **if**(dp) dp -> derivedOnly ();  **else** **cout** << "Error , cast should work !\n";  **return** 0; } | |

* The use of ***dynamic\_cast*** simplifies the logic required to cast a base pointer into a derived pointer.
* Example 3: The ***dynamic\_cast*** operator can also be used with template classes. For example, the following program reworks the template class from Example 4 in the preceding section so that it uses ***dynamic\_cast*** to determine the type of object returned by the ***generator()*** function.

/\* . . . . . . . same as Exampole 4, section 13.8 RTTI \*/

**int** **main**() { **Num** <**double**> ob1(10), \*p1;

**Square** <**double**> ob2(100.0), \*p2;

**Sqr\_root** <**double**> ob3(999.2), \*p3;

int i;

**cout** << " Generate some objects .\n";

**for**(i=0; i <10; i++) { p1 = generator();

p2 = **dynamic\_cast** < **Square**<**double**> \* > (p1);

**if**(p2) **cout** << " Square object : ";

p3 = **dynamic\_cast** < **Sqr\_root**<**double**> \* > (p1);

**if**(p3) **cout** << " Sqr\_root object : ";

**cout** << "Value is:" << p1 -> get\_val();

**cout** << **endl**; }

**return** 0; }

* Example 4: The following program demonstrates the use of ***reinterpret\_cast***.

**int** **main**() { **int** i;

**char** \*p = " *This is a string* ";

i = **reinterpret\_cast** <**int**> (p); /\* cast pointer to integer \*/

**cout** << i;

**return** 0;}

* Here ***reinterpret\_cast*** converts the pointer ***p*** into an ***integer***. This conversion represents a ***fundamental type change*** and is a good use of ***reinterpret\_cast***.
* Example 5 :The following program demonstrates const\_cast.

|  |  |
| --- | --- |
| **void** f( **const** **int** \*p) { **int** \*v;  v = **const\_cast** <**int** \*> (p); */\* cast away const - ness \*/*  \*v = 100; */\* now , modify object through v \*/* }  **int** **main**(){ **int** x = 99;  **cout** << "x before call : " << x << **endl** ;  f(&x);  **cout** << "x after call : " << x << **endl** ;  **return** 0; } | Output:  x before call: 99  x after call: 100 |

* As you can see, ***x*** was modified by ***f()*** even though the *parameter* to ***f()*** was specified as a ***const*** pointer.
* It must be stressed that the use of ***const\_cast*** to ***cast way*** const-ness is a potentially dangerous feature. Use it with care.
* Example 6: The ***static\_cast*** operator is essentially a substitute for the original cast operator. It simply performs a non-polymorphic cast. For example, the following casts a ***float*** into an ***int***.

**int** **main**() { **int** i; **float** f;

f = 199.22;

i = **static\_cast** <**int**> (f);

**cout** << i;

**return** 0;}