Chapter-10 VIRTUAL FUNCTIONS

•Virtual functions are used to supprot run-tme polymorfism.
•Ploymorfism is supported in C++ in two ways.
•First, it is supported at compile time, through the use of overloaded operators and functions.
•Second it is supported at run time, through the use of virtual functions.

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. Assume **derived** class inherits **base** class. *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.
p=&derived_ob;// p points to derived object.
```

Example:

```
#include <iostream>
using namespace std;
class base {private: int x;
public:
void setx(int i) {x=i;}
int getx() {return x;}};
class derived :public base {private: int y;
public:
void sety(int i) {y=i;}
int gety() {return y;}};
int main() {
base *p,b_ob;
derived d_ob;
p=&b_ob;
p->setx(10);//access base object
cout<<”Base object   x:”<<p->getx()<<endl;
```

```
p=&d_ob;
p->setx(99);//access derived object
cout<<”Derived object x:”<<p->getx()<<endl;
cout<<” Derived object y:”<<d_ob.gety()<<endl;
return 0;
}
```

Base pointer can point to derived object but opposite is not true.

Introduction to Virtual Functions

- A virtual function is a member function that is declared within a base class and redifined by a derived class.
- To create a virtual function, precede the function's declaration with the keyword **virtual**.
- Virtual functions are used to supprot run-tme polymorfism.

Example:

```
#include <iostream>
using namespace std;
class base {
public: int i; base (int  x)  {i=x;}
virtual void func () {
cout<<”using base version of func():  ”<<i<<endl;}
};
class derived1 :public base {
public:  derived1 (int  x) :base(x)  {}
void func(){cout<<”using derived1's version of func():”;
cout<<i*i<<endl;}
};
class derived2 :public base {
public:  derived2 (int  x) :base(x)  {}
void func(){cout<<”using derived2's version of func():”;
cout<<i+i<<endl;}
 };
```

```
int main() {base *p,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
```

- The redefinition of virtual function inside a derived class might seem somewhat similar to function overloading. However, two precess are distinctly different.
- An overloaded function must differ in type and/or number of parameters, while a redefined virtual function must have precisely the same type and number of parameters and the same return type. (If you change either the number or type of parameters when redefining a virtual function, it simply becomes an overloaded function and its virtual nature is lost.)
- Virtual functions must be class members.
- While destructor functions can be virtual, constructors cannot.
- The term overriding is used to describe virtual function redefinition. When a derived class does not override a virtual function, the function defined within its base class is used.

Example:

```
#include <iostream>
using namespace std;
class base {
public: int i; base (int  x)  {i=x;}
virtual void func () {
cout<<”using base version of func():  ”<<i<<endl;}
};
class derived1 :public base {
public:
derived1 (int  x) :base(x)  {}
void func(){cout<<”using derived1's version of func():”;
cout<<i*i<<endl;}
};
class derived2 :public base {
public:
derived2 (int  x) :base(x)  {}
 };
```

```
int main() {base *p,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 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
```

Example: (Virtual functions are randomly selected at runtime)

```
#include <iostream>
#include <cstdlib>
using namespace std;
class base {
public: int i; base (int  x)  {i=x;}
virtual void func () {
cout<<”using base version of func():  ”<<i<<endl;}
};
class derived1 :public base {
public:  derived1 (int  x) :base(x)  {}
void func(){cout<<”using derived1's version of func():”;
cout<<i*i<<endl;}
};
class derived2 :public base {
public:  derived2 (int  x) :base(x)  {}
void func(){cout<<”using derived2's version of func():”;
cout<<i+i<<endl;}
 };
```

```
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;
}
```

Example: (Use virtual function to define interface)

```
#include <iostream>
using namespace std;
class area {double dim1,dim2;//dimension of figure
public:
void setdim (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”<<endl; return 0.0;}
};
class rectangale :public area {
public:  getarea(){ double d1,d2;
getdim(d1,d2);
return d1*d2;}    };
class triangale :public area {
public:  getarea(){ double d1,d2;
getdim(d1,d2);
return 0.5*d1*d2;}    };
```

```
int main() {
area *p;
rectangle r;
triangle t;

r.setdim(10,5);
t.setdim(4.0,5.0);

p=&r;
cout<<"Rectangle has area: "<<p->getarea()<<endl;

p=&t;
cout<<"Triangle has area: "<<p->getarea()<<endl;

return 0;
}
```

More About Virtual Functions

•Sometimes when a virtual function is declared in the base class there is no meaningfull operation for it to perfom.
•When there is no meaningful action for base class virtual function to perform, the implication is that any derived class must override this function. To ensure that, C++ supports *pure virtual functions*.
•A pure virtual function has no definition relative to the base class. Only the function's protype is included. To make a pure virtual function, 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 complier that no body exists for this function relative to the base class. It forces any derived class to override it. If a derived class does not override it, a compile time error results.
•When a class contains at least one pure virtual function, it is referred to as an *abstract class*.

Example: (Computing area of rectangle and triangle using virtual function)

```
#include <iostream>
using namespace std;
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()=0;//pure virtual function
};
class rectangale :public area {
public:  getarea(){ double d1,d2;
getdim(d1,d2);
return d1*d2;}
 };
class triangale :public area {
public:  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()<<endl;

p=&t;
cout<<"Triangle has area: "<<p->getarea()<<endl;

return 0;
}
```

Example: (Virtual functions retain their virtual nature when inherited.)

```
#include <iostream>
using namespace std;
class base {public:
virtual void func ()
{cout<<”using base version of func():  ”<<i<<endl;}
};
class derived1 :public base
{public:
void func(){cout<<”using derived1's version of func(): \n”;}
};
class derived2 :public derived1
{public:
void func(){cout<<”using derived2's version of func():\n”;}
 };
```

```
int main()
{
base *p,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;}
```

Chapter 11 Templates and Exception Handling

- Using templates, it is possible to create generic functions and classes.
- In a generic function or class, the type of data that is operated upon is specified as a parameter. This allows us to use one function or class with several different types of data without having to expilicitly recode a specific version for each different data type. Thus, templates allow you to create reusable code.

- With C++ exception handling, your program can automatically invoke an error handling routine when an error occurs. The proper use of exception handling helps you create resilient code.

Generic Functions

•A generic function defines a general set of operations that will be applied to various types of data.
•A generic function has the type of data that it will operate upon passed to it as a parameter. Using this mechanism, the same general procedure can be applied to a wide range of data.
•Many algorithms are logically the same no matter what type of data is beening operated upon.
•By creating a generic function, you can define, independent of data, the nature of algorithm.
•When you create a generic function you are creating a function that can automatically overload itself.

The general form of a template function definition is shown here:
template <class Ttype> return-type function-name(parameter list)
{//body of function}

Here ***Ttype*** is a placeholder name for a data type used by function. You can also use keyword **typename** for **class**.

Example: (Function template)

```
#include <iostream>
using namespace std;
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=10.1,y=23.3;
cout<<”original i,j: ”<<i<<”  “<<j<<endl;
cout<<”original x,y: ”<<x<<”  “<<y<<endl;
swapargs(i,j);
swapargs(x,y);
cout<<”swapped i,j: ”<<i<<”  “<<j<<endl;
cout<<”swapped x,y: ”<<x<<”  “<<y<<endl;
return 0;
}
```

Example:

```
template <typename X>   //Declaration using typename.
void swapargs(X &a, X &b)
{ X temp;     temp=a;
a=b;
b=temp;
}
```

Example: More than one generic data type

```
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{ cout<<x<<”  “<<y<<endl;}
int main(){
myfunction(10,”hi”);
myfunction(0.23,10L);
return 0;
}
```

•Generic functions are similar to overloaded functions except that they are more restrictive.
•When functions are overloaded, you can have different actions performed within the body of each function. But a generic function must perform the same general action for all version.

If you overload a generic function, that overloaded function overrides the generic function relative to that specific version.

Example: (Overriding a template (generic) function.)

```
#include <iostream>
using namespace std;
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)”<<endl
}
int main() {int i=10,j=20;float x=10.1,y=23.3;
cout<<”original i,j: ”<<i<<”  “<<j<<endl;
cout<<”original x,y: ”<<x<<”  “<<y<<endl;
swapargs(i,j); //calls overloaded swapargs(int,int)
swapargs(x,y);
cout<<”swapped i,j: ”<<i<<”  “<<j<<endl;
cout<<”swapped x,y: ”<<x<<”  “<<y<<endl;
return 0;
}
```

Generic Classes

•In addition to defining generic function, you can also generic classes. When you do this, you create a class that defines all algorithms used by that class, but the actual type of data being manipulated will be specified as a parameter when object of that class are created.
•Generic classes are usefull when a class contains generalized logic. For example the same algorithm that maintains a queue of integers will also work for a queue of characters.
A general form of a generic class declaration is shown here:
template <class Ttype> class class-name
{
//members
}
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-seperated list.
Once you have created a generic class, you create a specific instance of that class 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.

Example: (A simple generic linked list)

```
#include <iostream>
using namespace std;
template <class data_t> class list
{
data_t data;
list *next;
public:
list(data_t d)
{
data=d;next=0;
}
void add(list *node) {node->next=this;next=0;}
list *getnext () {return next;}
data_t getdata () {return data;}
};
```

```
int main() {
list<char> start('a'),*p,*last;
int i;
last=&start;
for(i=1;i<26;i++){
p=new list<char> ('a'+i);
p->add(last);
last=p;
}
p=&start;//follow the list
while(p){
cout <<p->getdata();
p=p->getnext();
}
return 0;
}
```

•In this example, objects and pointers are created inside **main()** that specify that the data type of the list will be **char**. However, by simply changing the type of datas pecified when **list** object created, you can change the type of data stored by the list. For example, you could create another object that stores integers, doubles floats or data types that you create.
•For example, if you want to store address information, use this structure:

struct addr {
char name[40];
char streat[40];
char city[30];
char state[3];
char zip[12]; }
To use **list** to generate objects that will store objects of type **addr**, use a declaration like this (assuming that **structvar** contains a valid **addr** structure):
list<addr> obj (structvar);

Example: (This program demonstrates a generic stack.)

```
#include <iostream>
using namespace std;
define SIZE=10;
template <class StackType> class stack
{
stackType stck[SIZE]; //holds the stack
int tos; //index of top of stack;
public:
void init() {tos=0;//initialize stack}
void push(StackType ch);//push object on stack
StackType pop(); pop object from stack
};
template <class StackType>
void stack<StackType>::push(StackType ob)
{if (tos==SIZE){cout<<”stack is full. \n”; return 0;}
stck[tos]=ob;tos++;
}
```

```
template <class StackType>
StackType stack<StackType>::pop()
{if (tos==SIZE){cout<<”stack is empty. \n”; return 0;}
tos--;
return stck[tos];
}
int main() {stack <char> s1,s2; int i; //create two stacks
s1.init(); s2.init();
s1.push('a');s2.push('x');s1.push('b');s2.push('y');
s1.push('c');s2.push('z');
for (i=0;i<3;i++) cout<<”pop s1:”<<s1.pop()<<endl;
for (i=0;i<3;i++) cout<<”pop s2:”<<s2.pop()<<endl;
stack double ds1,ds2;
ds1.init(); ds2.init();
ds1.push('1.1');ds2.push('2.2');ds1.push('3.3');
ds2.push('4.4');ds1.push('5.5');ds2.push('6.6');
for (i=0;i<3;i++) cout<<”pop ds1:”<<ds1.pop()<<endl;
for (i=0;i<3;i++) cout<<”pop ds2:”<<ds2.pop()<<endl;
```

Example: (This example uses two generic data types in a class definition.)

```
#include <iostream>
using namespace std;
template <class Type1, class Type2> class myclass
{
Type1 i;Type2 j;
public:
myclass(Type1 a, Type2 b) {i=a;j=b;}
void show() {cout<<i<<”   “<<j<<endl;}
};
int main() {myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X,”This is a test”);
ob1.show();
ob2.show();
return 0;
}//
```

This program displays the following output

```
10   0.23
X   This is a test
```

Exception Handling

•C++ provides a built-in error handling mechanism that is called *exception handling*.
•Using exception handling, you can easily manage and respond to run-time errors.
•C++ exception handling built upon three keywords: **try**, **catch**, and **throw**.
•In the most general terms, program statements that you want to monitor for exceptions are contained in the **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.

The general form of ***try***, and ***catch*** are shown here:

```
try {//try block}
catch (type1 arg){ //catch block}
catch (type2 arg){ //catch block}
    .
    .
    .
catch (typeN arg){//catch block}
```

The general form of throw statement is shown here:

throw *exception*;

•Throw must be executed either from within the try block proper or from any function that the code within the block calls. *exception* is the value thrown.
•If you throw an exception for which there is no applicable **catch** statement, an abnormal program termination might occur. If your comlier complies with standard C++, throwing an unhandled exception causes standard library function, **terminate()** calls **abort()** to stop your program.

Example: A simple exception handling program

```
#include <iostream>
using namespace std;
int main() {
cout<<”start”<<endl;
try {//start a try block
cout<<”inside try block”<<endl;
throw 10; //throw an error
cout<<”This will not execute”<<endl;}
catch (int i) {//catch the error
cout<<”Caught one! Number is: ”<<i<<endl; }
cout <<”end”<<endl;
return 0;
}//The program displays the following output:
```

start
inside try block
Caught one! Number is: 10
end

Example: This example will not work.

```
#include <iostream>
using namespace std;
int main() {
cout<<”start”<<endl;
try {// start a try block
cout<<”inside try block”<<endl;
throw 10; // throw an error
cout<<”This will not execute”<<endl;}
catch (double i) {//Won’t work for an int exception
cout<<”Caught one! Number is: ”<<i<<endl; }
cout <<”end”<<endl;
return 0;
}// The program displays the following output:
```

start
inside try block
Abnormal program termination

Example : Throwing an exception from a function outside the try block.

```
#include <iostream>
using namespace std;
void Xtest(int test){
cout<<”Inside Xtest, test is: “<<test<<endl;
if (test) throw test;}
int main() { cout<<”start”<<endl;
try {// start a try block
cout<<”inside try block”<<endl;
Xtest(0);Xtest(1);Xtest(2);}
catch (int i) {//catch an int error
cout<<”Caught one! Number is: ”<<i<<endl; }
cout <<”end”<<endl;
return 0;
}
```

The program displays the following output:

```
start
inside try block
Inside Xtest, test is: 0
Inside Xtest, test is: 1
Caught one! Number is: 1
end
```

Example: A try/catch can be inside a function other than main().

```
#include <iostream>
using namespace std;
void Xhandler(int test){
try { if (test) throw test;}
catch (int i) {//catch an integer error
cout<<”Caught one! Ex. #: ”<<i<<endl; }
}
int main() { cout<<”start”<<endl;
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout <<”end”<<endl;
return 0;
}//
```

The program displays the following output:

```
start
Caught one! Ex. #: 1
Caught one! Ex. #: 2
Caught one! Ex. #: 3
end
```

Example: Different type of exceptions can be caught

```
#include <iostream>
using namespace std;
void Xhandler(int test){
try { if (test) throw test;
else throw “value zero”;}
catch (int i) { cout<<”Caught one! Ex. #: ”<<i<<endl; }
catch (char *str) {cout<<”Caught a string: ”<<str<<endl;}
}
int main() { cout<<”start”<<endl;
Xhandler(1);Xhandler(2);
Xhandler(0);
Xhandler(3);
cout <<”end”<<endl;
return 0;
}//
```

The program displays the following output:

```
start
Caught one! Ex. #: 1
Caught one! Ex. #: 2
Caught a string: Value is zero
Caught one! Ex. #: 3
end
```

More About Exception Handling

In some circumstances you will want an exception handler to catch all exceptions instead of just a certain type. To catch all types of exceptions use this form of **catch**:

catch(...) { //Process all exceptions }

You can restrict the type of exceptions that a function can throw back to its caller. The general form of this is shown here:

ret-type func-name(arg-list) throw (type-list)
{// program}

Here only those data types contained in the comma-seperated *type-list* may be thrown by the function. Throwing any other type of expression will cause abnormal program termination. If you don’t want a function to be able to throw any exception, use an empty list.

Example: (This example catches all exceptions.)

```
#include <iostream>
using namespace std;
void Xhandler(int test){
try { if (test==0) throw test;// throw int
try { if (test==1) throw 'a';// throw char
try { if (test==2) throw 123.23;// throw double
}
catch (...) { cout<<”Caught one!”<<endl;}//catch all exceptions
}
int main() { cout<<”start”<<endl;
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout <<”end”<<endl;
return 0;}//
```

This program displas the following output:

start
Caught one!
Caught one!
Caught one!
end

Example: (This example uses catch as a default.)

```
#include <iostream>
using namespace std;
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) { cout<<”Caught ”<<i<<endl;}//catch an int exception
catch (…) { cout<<”Caught one!”<<endl;}//catch all other exceptions
}
int main() { cout<<”start”<<endl;
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout <<”end”<<endl;
return 0;}//
```

This program displas the following output:

start
Caught 0
Caught one!
Caught one!
end

Example: (Restriction function throw types. The function can throw only int char and double)

```
#include <iostream>
using namespace std;
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”<<endl;
try{Xhandler(0);}//also try passing 1 and 2 to Xhandler()
catch (int i) { cout<<”Caught int”<<endl;}
catch (char c) { cout<<”Caught char”<<endl;}
catch (double d) { cout<<”Caught double”<<endl;}
cout <<”end”<<endl;
return 0;}//
```

//This function can throw no exception

```
void Xhandler(int test) throw()
{
if (test==0) throw test;
if (test==1) throw 'a';
if (test==2) throw 123.23;
}
```

Example: (Rethrowing an exception)

```
#include <iostream>
using namespace std;
void Xhandler(){
try {char x[20]=”Hello”;
throw x;}// throw char *
catch (char *c) { cout<<”Caught  char inside Xhandler\n”;
throw c;}//rethrow char * out of function
}
int main() { cout<<”start”<<endl;
try {Xhandler();}
catch(char *c) { cout<<”Caught  char inside main”<<endl;}
cout <<”end”<<endl;
return 0;}//
```

This program displas the following output:

```
start
Caught  char * inside Xhandler
Caught  char * inside main
end
```

Handling Exceptions Thrown by New

In standard C++, when an allocation request cannot be honored, **new** throws a **bad_alloc** exception. If you don’t catch this exception, your program will be terminated. To have access to this exception, you must include the header **<new>** in your program.
In standard C++ it is also possible to to have **new** return null instead of throwing an exception when an allocation failure occurs. This form of **new** is shown here:

p_var=new(nothrow) type;

Here *p_var* is a pointer of **type**.

Example: 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[10];} //allocate memory for int
catch(bad_alloc xa) { cout<<”Allocation failure”<<endl;
return 1;}
for(*p=0;*p<10;(*p)++) cout<<*p<<” “;cout<<endl;
delete p;//free the memory
return 0;
}
```

Example: (Force an allocation failure)

```
#include <iostream>
#include <new>
using namespace std;
int main() {
double *p;
//This will eventually run out of memory
do{
try {p=new double[100000];} //allocate memory
catch(bad_alloc xa) { cout<<”Allocation failure.”<<endl;
return 1;}
cout<<”Allocation okay.”<<endl;
}while(p);
return 0;
}
```

Example: (Demonstrate the new(nothrow) alternative)

```
#include <iostream>
#include <new>
using namespace std;
int main() {
double *p;
//This will eventually run out of memory
do{
p=new(nothrow) double[100000];// allocate memory
If (p) cout<<”Allocation Okay”<<endl;
else cout<<”Allocation Error.”<<endl;
}while(p);
return 0;
}
```

Chapter-12 Run-Time Type Identification and the Casting Operators

RTTI allows you to identify the type of an object during the execution of your program. The casting operator give you safer, more controlled ways to cast. One of the casting operators, **dynamic_cast**, relates directly to RTTI, so it makes sense to discuss these two topics in the same chapter.

Understanding Run-Time Type Identification (RTTI)

•It is not always possible to know in advance what type of object will be pointed to by a base pointer at any given moment in time. This determination must be made at run time, using run-time type identification.
•To obtain an object's type, use **typeid**. You must include the header **<typeinfo>** in order to use **typeid**. The most common form of **typeid** is shown here:

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 the followin public members:

bool operator==(const type_info &ob) //comparision
bool operator!=(const type_info &ob) //comparision
bool before(const type_info &ob) //for internal use
const char *name();//returns a pointer to the name of the type

The **before()** function returns true if the invoking object is before the object used as a parameter in collation order.

The second form of ***typeid*** is shown here:

typeid(type-name)

Example:

```
#include <iostream>
#include <typeinfo>
using namespace std;
class BaseClass {
public:
virtual void f () {};//make BaseClass polimorphic
};
class Derived1 :public BaseClass {
//
};
class Derived2 :public BaseClass {
//
};
int main(){
int i;BaseClass *p,baseob; Derived1 ob1;Derived2 ob2;
cout<<”Typeid of i is:”<<typeid(i).name()<<endl;
```

```
p=&baseob;
cout<<”p is pointing to an object of type “;
cout<<typeid(*p).name()<<endl;
p=&ob1;
cout<<”p is pointing to an object of type “;
cout<<typeid(*p).name()<<endl;
p=&ob2;
cout<<”P is pointing to an object of type “;
cout<<typeid(*p).name()<<endl;
return 0;}//
```

This program displays the following output:

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

Example:

```
#include <iostream>
#include <typeinfo>
using namespace std;
class BaseClass {
public:
virtual void f () {};// make BaseClass polimorphic
};
class Derived1 :public BaseClass {
//
};
class Derived2 :public BaseClass {
//
};
Void WhatType(BAseClass &ob){
cout<<”ob is referencing an object of type “;
cout<<typeid(ob).name()<<endl;
}
```

```
int main(){
int i;BaseClass baseob;
Derived1 ob1;
Derived2 ob2;
WhatType(baseob);
WhatType(ob1);
WhatType(ob2);
return 0;
}
```

This program displays the following output:

ob is referencing an object of type class BaseClass
ob is referencing an object of type class Derived1
ob is referencing an object of type class Derived1

Example: Demonstrate == and != relative to typeid

```
#include <iostream>
#include <typeinfo>
using namespace std;
class X {
public:
virtual void f () {};
};
class Y {
public:
virtual void f () {};
};
```

```
int main(){
X x1,x2;
Y y1;
if(typeid(x1)==typeid(x2))
cout<<”x1 and x2 are same types”<<endl;
else cout<<”x1 and x2 are different types”<<endl;
if(typeid(x1)==typeid(y1))
cout<<”x1 and y1 are same types”<<endl;
else cout<<”x1 and y1 are different types”<<endl;
return 0;
}
```

This program displays the following output:

x1 and x2 are same types
x1 and y1 are different types

Example: (This program uses RTTI)

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;
class Shape {public:virtual void example ()=0;};
class Rectangle: public Shape
{ public: void example()
{cout<<"****\n* *\n* *\n*****\n";}};
class Triangle: public Shape
{ public:
void example(){cout<<"*\n* *\n* *\n*****\n";}};
class Line: public Shape
{ public:
 void example(){cout<<"******\n";}};
class NullShape: public Shape{public: void example(){}};
```

```
Shape *generator() //return a pointer to a shape object
{ switch (rand()%4){
case 0: return new Line;
case 1: return new Rectangle;
case 2: return new Triangle;
case 3: return new NullShape;}return NULL;}
int main(){int i; Shape *p;
for (i=0;i<10;i++)
{p=generator();//get next object
cout<<typeid(*p).name()<<endl;
//draw object only if it is not a NullShape
if (typeid(*p)!=typeid(NullShape())) p->example();
}
return 0;}
```

4Line

******

9Rectangle

******

* *

* *

******

9Rectangle

******

* *

* *

******

8Triangle

*

* *

* *

******

9NullShape

Example: (typeid can be used with templates.)

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <typeinfo>
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 Num<T>::x*Num<T>::x;}};
template <class T> class Sqr_root:public Num<T>{
public: Sqr_root(T i):Num<T>(i){}
T get_val() {return sqrt((double) Num<T>::x);}};
```

```
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>"<<endl;
p1=&ob2;
if(typeid(*p1)!=typeid(ob1))
cout<<"Value is: "<<p1->get_val()<<endl;
cout<<"Now generate some objects"<<endl;
```

```
for (i=0;i<10;i++)
{p1=generator();//get next object
if(typeid(*p1)==typeid(Square<double>));
cout<<”Square object”<<endl;
if(typeid(*p1)==typeid(Sqr_root<double>));
cout<<”Sqr_root object”<<endl;
cout<<”Value is: “<< p1->get_val()<<endl;
}
return 0;}
```

This program displays the following output:

```
3NumIdE
6SquareIdE
8Sqr_rootIdE
is Square<double>: Value is: 10000
Now generate some objects
Sqr_root object: Value is: 9.27362
Sqr_root object: Value is: 3.87298
Sqr_root object: Value is: 5.91608
Square object: Value is: 8464
Sqr_root object: Value is: 4.58258
Square object: Value is: 729
Square object: Value is: 3481
Sqr_root object: Value is: 5.09902
Square object: Value is: 676
Square object: Value is: 1296
```

Using dynamic_cast

Altough C++ still supports the traditional casting operator defined by C, C++ adds four new ones. They are ***dynamic_cast***, ***const_cast***, ***reinterpret_cast***, and ***static_cast***. The ***dynamic_cast*** operator performs a run-time cast that verifies the validity of a cast. If, at the time ***dynamic_cast*** is executed, the cast is invalid, the cast fails. The general form of ***dynamic_cast*** is shown here:

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

Example: (Assume that ***Base*** is polimorphic 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
dp=dynamic_cast<Derived *> (bp);
if (dp)<<”Cast is Okay”;/
```

Example: (The cast fails because ***bp*** is pointing to a ***Base*** object and it is illegal to cast a base object into a derived object.)

```
Base *bp,b_ob;
Derived *dp, d_ob;
bp=&b_ob;// base pointer points to Base object
dp=dynamic_cast<Derived *> (bp);
if (!dp)<<”Cast Fails.”
```

Example: (dynamic_cast)

```
#include <iostream>
using namespace std;
class Base {
public:
virtual void f () {cout<<”İnside Base”<<endl;};
};
class Derived :public Base {
public:
void f () {cout<<”İnside Derived”<<endl;};
};
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”<<endl; cout<<endl;
```

```
bp=dynamic_cast<Base *> (&d_ob);
if (bp) { cout<<"Cast from Derived * to Base OK.\n";
bp->f();} else cout<<"Error"<<endl; cout<<endl;
bp=dynamic_cast<Base *> (&b_ob);
if (bp) { cout<<"Cast from Base * to Base OK.\n";
bp->f();} else cout<<"Error"<<endl; cout<<endl;
dp=dynamic_cast<Derived *> (&b_ob);
if (dp) { cout<<"Error"<<endl;} else
cout<<"Cast from Base * to Derived not OK.\n";
cout<<endl;
bp=&d_ob;
dp=dynamic_cast<Derived *> (bp);
if (dp) { cout<<"Casting bp to a Derived * OK.\n"<<
"because bp is really pointing to a Derived object.\n";
bp->f();} else cout<<"Error"<<endl; cout<<endl;
```

```
bp=&b_ob;//bp points to Base object.
dp=dynamic_cast<Derived *> (bp);
if (dp)  cout<<”Error”<<endl; else
{ cout<<”Now Casting bp to a Derived * is not OK.\n”<<
“because bp is really 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”<<endl;
 cout<<endl;
return 0;
}
```

The program displays the following output:

```
Cast from Derived * to Derived OK.
İnside Derived
Cast from Derived * to Base OK.
İnside Derived

Cast from Base * to Base OK.
İnside Base

Cast from Base * to Derived not OK.

Casting bp to a Derived * OK.
because bp is really pointing to a Derived object.
İnside 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.
İnside Derived
```

Example: use dynamic_cast to replace typeid

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base {
public:
virtual void f () {};
};
class Derived :public Base {
public:
void derivedOnly () {cout<<”İs a Derived object\n”;}};
int main() {Base *bp,b_ob;
Derived *dp,d_ob;
bp=&b_ob;//typeid kullan
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";

bp=&b_ob; //use dynamic_cast
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;
}
```

Previous program displays the following output:

Cast from Base to Derived failed.
İs a Derived object
Cast from Base to Derived failed.
İs a Derived object

Using const_cast, reinterpret_cast and static_cast

General forms are shown here:

reinterpret_cast<target-type> (expr)
const_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
In general, these casting operators provide a safer, more explicit means of performing certain type conversion than that provided by the C-style cast.

The reinterpret_cast operator changes one pointer type into another, fundamentally different, pointer type. It can also change a pointer into an integer an integer into a pointer. A reinterpret_cast should be used for casting inheritly incompatible pointer types.

Example: (An example that uses reinterpret_cast)

```
#include <iostream>
using namespace std;
int main() { int i;
char *p=”This is a string”;
i=reinterpret_cast<int &>(*p);//cast pointer to integer.
cout<<i<<endl<<p<<endl;
return 0;
}
```

The ***const_cast*** operator is used to explicitly override ***const*** and/or ***volatile*** in a cast. The target type must be the same as the source type except for the alteration of its ***const*** or ***volatile*** attributes. The most common use of ***const_cast*** is to remove ***const-***nes.

Example: (const_cast)

```
#include <iostream>
using namespace std;
void f(const int *p)
{int *v;
v=const_cast<int *>(p);
*v=100;}
int main() { int x=99;
cout<<"x before call: "<<x<<endl;
f(&x);
cout<<"x after call: "<<x<<endl;
return 0;
}//This program displays the following:
x before call: 99
x after call: 100
```

Example: (const_cast)

```
#include <iostream>
using namespace std;

void print (char * str)
{ cout << str << endl;}
int main () {
  const char * c = "sample text";
  print ( const_cast<char *> (c) );
  //print (  (c) );
  return 0;
}
```

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.

Example: (static_cast)

```
#include <iostream>
using namespace std;
int main() { int i;
float f;
f=199.22;
i=static_cast<int>(f);
cout<<i<<endl;
return 0;
}//
```

Program çıktısı

199