Illustrations List *(Main Page)*

```
// Fig. 21.1: fig21_01.cpp
// Demonstrating data type bool.
#include <iostream>
#include <iomanip>
using namespace std;

```

Fig. 21.1 Demonstrating the fundamental data type `bool` (part 1 of 2).

```
int main()
{
   bool boolean = false;
   int x = 0;

   cout << "boolean is " << boolean
        << "\nEnter an integer: ";
   cin >> x;

   cout << "integer " << x << " is"
        << ( x ? " nonzero " : " zero " )
        << "and interpreted as ";

   if ( x )
      cout << "true\n";
   else
      cout << "false\n";

   boolean = true;
   cout << "boolean is " << boolean;
   cout << "\nboolean output with boolalpha manipulator is "
        << boolalpha << boolean << endl;

   return 0;
}
```

```
boolean is 0
Enter an integer: 22
integer 22 is nonzero and interpreted as true
boolean is 1
boolean output with boolalpha manipulator is true
```

Fig. 21.1 Demonstrating the fundamental data type `bool` (part 2 of 2).

```
// Fig. 21.2: fig21_02.cpp
// Demonstrating the static_cast operator.
#include <iostream.h>

class BaseClass {
public:
   void f( void ) const { cout << "BASE\n"; }
};

class DerivedClass: public BaseClass {
public:
   void f( void ) const { cout << "DERIVED\n"; }
};

void test( BaseClass * );

int main()
{
   // use static_cast for a conversion
   double d = 8.22;
   int x = static_cast< int >( d );

   cout << "d is " << d << "\nx is " << x << endl;

   BaseClass base;  // instantiate base object
   test( &base );   // call test

   return 0;
}

void test( BaseClass *basePtr )
{
   DerivedClass *derivedPtr;

   // cast base class pointer into derived class pointer
   derivedPtr = static_cast< DerivedClass * >( basePtr );
   derivedPtr->f();   // invoke DerivedClass function f
}
```

```
d is 8.22
x is 8
DERIVED
```

Fig. 21.2 Demonstrating operator `static_cast`.

```
// Fig. 21.3: fig21_03.cpp
// Demonstrating the const_cast operator.
#include <iostream.h>

class ConstCastTest {
public:
   void setNumber( int );
   int getNumber() const;
   void printNumber() const;
private:
   int number;
};

void ConstCastTest::setNumber( int num ) { number = num; }
```

```cpp

int ConstCastTest::getNumber() const { return number; }

void ConstCastTest::printNumber() const
{
   cout << "\nNumber after modification: ";

   // the expression number-- would generate compile error
   // undo const-ness to allow modification
   const_cast< ConstCastTest * >( this )->number--;

   cout << number << endl;
}

```

Fig. 21.3 Demonstrating the `const_cast` operator (part 1 of 2).

```cpp
int main()
{
   ConstCastTest x;
   x.setNumber( 8 );  // set private data number to 8

   cout << "Initial value of number: " << x.getNumber();

   x.printNumber();
   return 0;
}
```

```
Initial value of number: 8
Number after modification: 7
```

Fig. 21.3 Demonstrating the `const_cast` operator (part 2 of 2).

```cpp
// Fig. 21.4: fig21_04.cpp
// Demonstrating reinterpret_cast operator.
#include <iostream.h>

int main()
{
   unsigned x = 22, *unsignedPtr;
   void *voidPtr = &x;
   char *charPtr = "C++";
```

Fig. 21.4 Demonstrating operator `reinterpret_cast` (part 1 of 2).

```cpp

   // cast from void * to unsigned *
   unsignedPtr = reinterpret_cast< unsigned * >( voidPtr );

   cout << "*unsignedPtr is " << *unsignedPtr
        << "\ncharPtr is " << charPtr;

   // use reinterpret_cast to cast a char * pointer to unsigned
   cout << "\nchar * to unsigned results in: "
        << ( x = reinterpret_cast< unsigned >( charPtr ) );

   // cast unsigned back to char *
   cout << "\nunsigned to char * results in: "
```

```
        << reinterpret_cast< char * >( x ) << endl;

   return 0;
}
```

```
*unsignedPtr is 22
charPtr is C++
char * to unsigned results in: 4287824
unsigned to char * results in: C++
```

Fig. 21.4 Demonstrating operator `reinterpret_cast` (part 2 of 2).

```
// Fig. 21.5: fig21_05.cpp
// Demonstrating namespaces.
#include <iostream>
using namespace std;  // use std namespace

int myInt = 98;       // global variable

namespace Example {
   const double PI = 3.14159;
   const double E = 2.71828;
   int myInt = 8;
   void printValues();

   namespace Inner {   // nested namespace
      enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
   }
}

namespace {           // unnamed namespace
   double d = 88.22;
}

int main()
{
   // output value d of unnamed namespace
   cout << "d = " << d;

   // output global variable
   cout << "\n(global) myInt = " << myInt;

   // output values of Example namespace
   cout << "\nPI = " << Example::PI << "\nE = "
        << Example::E << "\nmyInt = "
        << Example::myInt << "\nFISCAL3 = "
        << Example::Inner::FISCAL3 << endl;

   Example::printValues();  // invoke printValues function

   return 0;
}

void Example::printValues()
{
   cout << "\n\nIn printValues:\n" << "myInt = "
        << myInt << "\nPI = " << PI << "\nE = "
        << E << "\nd = " << d << "\n(global) myInt = "
        << ::myInt << "\nFISCAL3 = "
        << Inner::FISCAL3 << endl;
```

```
}
```

Fig. 21.5 Demonstrating the use of **namespace**s (part 1 of 2).

```
d = 88.22
(global) myInt = 98
PI = 3.14159
E = 2.71828
myInt = 8
FISCAL3 = 1992


In printValues:
myInt = 8
PI = 3.14159
E = 2.71828
d = 88.22
(global) myInt = 98
FISCAL3 = 1992
```

Fig. 21.5 Demonstrating the use of **namespace**s (part 2 of 2).

```
// Fig. 21.6: fig21_06.cpp
// Demonstrating RTTI capability typeid.
#include <iostream.h>
#include <typeinfo.h>

template < typename T >
T maximum( T value1, T value2, T value3 )
{
   T max = value1;

   if ( value2 > max )
      max = value2;

   if ( value3 > max )
      max = value3;

   // get the name of the type (i.e., int or double)
   const char *dataType = typeid( T ).name();

   cout << dataType << "s were compared.\nLargest "
        << dataType << " is ";

   return max;
}

int main()
{
   int a = 8, b = 88, c = 22;
   double d = 95.96, e = 78.59, f = 83.89;
```

Fig. 21.6 Demonstrating **typeid** (part 1 of 2).

```

   cout << maximum( a, b, c ) << "\n";
   cout << maximum( d, e, f ) << endl;
```

```

   return 0;
}
```

```
ints were compared.
Largest int is 88
doubles were compared.
Largest double is 95.96
```

Fig. 21.6 Demonstrating `typeid` (part 2 of 2).

```
// Fig. 21.7: fig21_07.cpp
// Demonstrating dynamic_cast.
#include <iostream.h>

const double PI = 3.14159;

```

Fig. 21.7 Demonstrating `dynamic_cast` (part 1 of 3).

```
class Shape {
   public:
      virtual double area() const { return 0.0; }
};

class Circle: public Shape {
public:
   Circle( int r = 1 ) { radius = r; }

   virtual double area() const
   {
      return PI * radius * radius;
   };
protected:
   int radius;
};

class Cylinder: public Circle {
public:
   Cylinder( int h = 1 ) { height = h; }

   virtual double area() const
   {
      return 2 * PI * radius * height +
             2 * Circle::area();
   }
private:
   int height;
};

void outputShapeArea( const Shape * );    // prototype

int main()
{
   Circle circle;
   Cylinder cylinder;
   Shape *ptr = 0;

   outputShapeArea( &circle );    // output circle's area
```

```
   outputShapeArea( &cylinder );  // output cylinder's area
   outputShapeArea( ptr );        // attempt to output area
   return 0;
}

void outputShapeArea( const Shape *shapePtr )
{
   const Circle *circlePtr;
   const Cylinder *cylinderPtr;

   // cast Shape * to a Cylinder *
   cylinderPtr = dynamic_cast< const Cylinder * >( shapePtr );
```

Fig. 21.7 Demonstrating `dynamic_cast` (part 2 of 3).

```

   if ( cylinderPtr != 0 )  // if true, invoke area()
      cout << "Cylinder's area: " << cylinderPtr->area();
   else {  // shapePtr does not refer to a cylinder

      // cast shapePtr to a Circle *
      circlePtr = dynamic_cast< const Circle * >( shapePtr );

      if ( circlePtr != 0 )  // if true, invoke area()
         cout << "Circle's area: " << circlePtr->area();
      else
         cout << "Neither a Circle nor a Cylinder.";
   }

   cout << endl;
}
```

```
Circle's area: 3.14159
Cylinder's area: 12.5664
Neither a Circle nor a Cylinder.
```

Fig. 21.7 Demonstrating `dynamic_cast` (part 3 of 3).

.

Operator	Operator keyword	Description
Logical operator keywords		
`&&`	`and`	logical AND
`||`	`or`	logical OR
`!`	`not`	logical NOT
Inequality operator keyword		
`!=`	`not_eq`	inequality
Bitwise operator keywords		
`&`	`bitand`	bitwise AND
`|`	`bitor`	bitwise inclusive OR
`^`	`xor`	bitwise exclusive OR

Fig. 21.8 Operator keywords as alternatives to operator symbols.

Operator	Operator keyword	Description
`~`	`compl`	bitwise complement
Bitwise assignment operator keywords		
`&=`	`and_eq`	bitwise AND assignment
`|=`	`or_eq`	bitwise inclusive OR assignment
`^=`	`xor_eq`	bitwise exclusive OR assignment

Fig. 21.8 Operator keywords as alternatives to operator symbols.

```
// Fig. 21.9: fig21_09.cpp
// Demonstrating operator keywords.
#include <iostream>
#include <iomanip>
#include <iso646.h>
using namespace std;

int main()
{
   int a = 8, b = 22;

   cout << boolalpha
        << "   a and b: " << ( a and b )
        << "\n    a or b: " << ( a or b )
        << "\n     not a: " << ( not a )
        << "\na not_eq b: " << ( a not_eq b )
        << "\na bitand b: " << ( a bitand b )
        << "\na bit_or b: " << ( a bitor b )
        << "\n   a xor b: " << ( a xor b )
        << "\n   compl a: " << ( compl a )
        << "\na and_eq b: " << ( a and_eq b )
        << "\n a or_eq b: " << ( a or_eq b )
        << "\na xor_eq b: " << ( a xor_eq b ) << endl;

   return 0;
}
```

```
   a and b: true
    a or b: true
     not a: false
a not_eq b: true
a bitand b: 22
a bit_or b: 22
   a xor b: 0
   compl a: -23
a and_eq b: 22
 a or_eq b: 30
a xor_eq b: 30
```

Fig. 21.9 Demonstrating the use of the operator keywords.

```
// Fig 21.10: array2.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<( ostream &, const Array & );
public:
   Array( int = 10 );  // default/conversion constructor
   ~Array();           // destructor
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
};

#endif
```

Fig. 21.10 Single-argument constructors and implicit conversions (part 1 of 4).

```
// Fig 21.10: array2.cpp
// Member function definitions for class Array
#include <assert.h>
#include "array2.h"

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   cout << "Array constructor called for "
        << size << " elements\n";

   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
}

```

Fig. 21.10 Single-argument constructors and implicit conversions (part 2 of 4).

```
// Destructor for class Array
Array::~Array() { delete [] ptr; }

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ )
      output << a.ptr[ i ] << ' ' ;

   return output;   // enables cout << x << y;
}
```

Fig. 21.10 Single-argument constructors and implicit conversions (part 3 of 4).

```
// Fig 21.10: fig21_10.cpp
// Driver for simple class Array
#include <iostream.h>
#include "array2.h"

void outputArray( const Array & );

int main()
{
   Array integers1( 7 );

   outputArray( integers1 );   // output Array integers1

   outputArray( 15 );  // convert 15 to an Array and output

   return 0;
}

void outputArray( const Array &arrayToOutput )
{
   cout << "The array received contains:\n"
        << arrayToOutput << "\n\n";
}
```

```
Array constructor called for 7 elements
The array received contains:
0 0 0 0 0 0 0

Array constructor called for 15 elements
The array received contains:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Fig. 21.10 Single-argument constructors and implicit conversions (part 4 of 4).

```
// Fig. 21.11: array3.h
// Simple class Array (for integers)
#ifndef ARRAY1_H
#define ARRAY1_H

#include <iostream.h>

class Array {
   friend ostream &operator<<( ostream &, const Array & );
public:
   explicit Array( int = 10 );  // default constructor
   ~Array();                    // destructor
private:
   int size; // size of the array
   int *ptr; // pointer to first element of array
};

#endif
```

Fig. 21.11 Demonstrating an `explicit` constructor (part 1 of 4).

```
// Fig. 21.11: array3.cpp
// Member function definitions for class Array
#include <assert.h>
#include "array3.h"
```

```

// Default constructor for class Array (default size 10)
Array::Array( int arraySize )
{
   size = ( arraySize > 0 ? arraySize : 10 );
   cout << "Array constructor called for "
        << size << " elements\n";

   ptr = new int[ size ]; // create space for array
   assert( ptr != 0 );    // terminate if memory not allocated

   for ( int i = 0; i < size; i++ )
      ptr[ i ] = 0;          // initialize array
}

// Destructor for class Array
Array::~Array() { delete [] ptr; }

// Overloaded output operator for class Array
ostream &operator<<( ostream &output, const Array &a )
{
   int i;

   for ( i = 0; i < a.size; i++ )
      output << a.ptr[ i ] << ' ' ;

   return output;   // enables cout << x << y;
}
```

Fig. 21.11 Demonstrating an **`explicit`** constructor (part 2 of 4).

```
// Fig. 21.11: fig21_11.cpp
// Driver for simple class Array
#include <iostream.h>
#include "array3.h"

void outputArray( const Array & );

int main()
{
   Array integers1( 7 );

   outputArray( integers1 );   // output Array integers1

   outputArray( 15 );  // convert 15 to an Array and output

```

Fig. 21.11 Demonstrating an **`explicit`** constructor (part 3 of 4).

```
   outputArray( Array( 15 ) ); // really want to do this!

   return 0;
}

void outputArray( const Array &arrayToOutput )
{
   cout << "The array received contains:\n"
        << arrayToOutput << "\n\n";
}
```

```
Compiling...
Fig21_11.cpp
Fig21_11.cpp(14) : error: 'outputArray' :
   cannot convert parameter 1 from 'const int' to
   'const class Array &'
Array3.cpp
```

Fig. 21.11 Demonstrating an `explicit` constructor (part 4 of 4).

```
// Fig. 21.12: fig21_12.cpp
// Demonstrating storage class specifier mutable.
#include <iostream.h>

class TestMutable {
public:
   TestMutable( int v = 0 ) { value = v; }
   void modifyValue() const { value++; }
   int getValue() const { return value; }
private:
   mutable int value;
};

int main()
{
   const TestMutable t( 99 );

   cout << "Initial value: " << t.getValue();

   t.modifyValue();   // modifies mutable member
   cout << "\nModified value: " << t.getValue() << endl;

   return 0;
}
```

```
Initial value: 99
Modified value: 100
```

Fig. 21.12 Demonstrating a `mutable` data member.

```
// Fig. 21.13: fig21_13.cpp
// Demonstrating operators .* and ->*
#include <iostream.h>

class Test {
public:
   void function() { cout << "function\n"; }
   int value;
};

void arrowStar( Test * );
void dotStar( Test * );

int main()
```

```
{
   Test t;

   t.value = 8;
   arrowStar( &t );
   dotStar( &t );
   return 0;
}

```

Fig. 21.13 Demonstrating the `.*` and `->*` operators (part 1 of 2).

```
void arrowStar( Test *tPtr )
{
   void ( Test::*memPtr )() = &Test::function;
   ( tPtr->*memPtr )();  // invoke function indirectly
}

void dotStar( Test *tPtr )
{
   int Test::*vPtr = &Test::value;
   cout << ( *tPtr ).*vPtr << endl;  // access value
}
```

```
function
8
```

Fig. 21.13 Demonstrating the `.*` and `->*` operators (part 2 of 2).

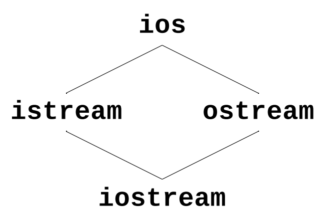

Fig. 21.14 Multiple inheritance to form class `iostream`.

```
// Fig. 21.15: fig21_15.cpp
// Attempting to polymorphically call a function
// multiply inherited from two base classes.
#include <iostream.h>

class Base {
public:
   virtual void print() const = 0;  // pure virtual
};

class DerivedOne : public Base {
public:
   // override print function
   void print() const { cout << "DerivedOne\n"; }
};

class DerivedTwo : public Base {
public:
```

```
   // override print function
   void print() const { cout << "DerivedTwo\n"; }
};

class Multiple : public DerivedOne, public DerivedTwo {
public:
   // qualify which version of function print
   void print() const { DerivedTwo::print(); }
};

int main()
{
   Multiple both;   // instantiate Multiple object
   DerivedOne one;  // instantiate DerivedOne object
   DerivedTwo two;  // instantiate DerivedTwo object
```

Fig. 21.15 Attempting to call a multiply inherited function polymorphically (part 1 of 2).

```

   Base *array[ 3 ];
   array[ 0 ] = &both;   // ERROR--ambiguous
   array[ 1 ] = &one;
   array[ 2 ] = &two;

   // polymorphically invoke print
   for ( int k = 0; k < 3; k++ )
      array[ k ] -> print();

   return 0;
}
```

```
Compiling...
fig21_14.cpp
fig21_14.cpp(36) : error: '=' :
   ambiguous conversions from 'class Multiple *' to
   'class Base *'
```

Fig. 21.15 Attempting to call a multiply inherited function polymorphically (part 2 of 2).

```
// Fig. 21.16: fig21_16.cpp
// Using virtual base classes.
#include <iostream.h>

class Base {
public:
   // implicit default constructor

   virtual void print() const = 0; // pure virtual
};

class DerivedOne : virtual public Base {
public:
   // implicit default constructor calls
   // Base default constructor

   // override print function
   void print() const { cout << "DerivedOne\n"; }
};

```

```
class DerivedTwo : virtual public Base {
public:
   // implicit default constructor calls
   // Base default constructor

   // override print function
   void print() const { cout << "DerivedTwo\n"; }
};

class Multiple : public DerivedOne, public DerivedTwo {
public:
   // implicit default constructor calls
   // DerivedOne and DerivedTwo default constructors

   // qualify which version of function print
   void print() const { DerivedTwo::print(); }
};

int main()
{
   Multiple both;   // instantiate Multiple object
   DerivedOne one;  // instantiate DerivedOne object
   DerivedTwo two;  // instantiate DerivedTwo object

   Base *array[ 3 ];
   array[ 0 ] = &both;
   array[ 1 ] = &one;
   array[ 2 ] = &two;

```

Fig. 21.16 Using `virtual` base classes (part 1 of 2).

```
   // polymorphically invoke print
   for ( int k = 0; k < 3; k++ )
      array[ k ] -> print();

   return 0;
}
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 21.16 Using `virtual` base classes (part 2 of 2).