CEIBO 8051C++ Compiler

User's Manual

Rev. 10/2017

contents

Preface

8051C++ Software P-1

Chapter 1 - The Embedded C++ language overview

The Embedded C++ language overview 1-1
Mutable specifier 1-2
Exception handling 1-2
Runtime type identification 1-3
Namespace 1-3
Template 1-4
Multiple inheritance and virtual inheritance 1-4
Library 1-4

Chapter 2 - Installation

Introduction 2-1
Software and Hardware Requirements 2-1
Directories structure 2-1

Chapter 3 - EC++ Projects

Creating EC++ Projects 3-1
Creating Custom EC++ Libraries 3-1
EC++ language extensions for 8051 3-2

Chapter 4 - Keywords

List of Keywords

Auto 4-3
Bool 4-3
Break 4-4
Case 4-6
Char 4-6
Class 4-7
Const 4-12
Continue 4-14
Default 4-15
Delete 4-15
Do 4-19
Double 4-20
Else 4-20
Enum 4-21
Extern 4-22

False 4-24
Float 4-24
For 4-25
Friend 4-27
Goto 4-29
If 4-31
Inline 4-32
Int 4-33
Long 4-34
New 4-34
Operator 4-35
Private 4-38
Protected 4-41
Public 4-44
Register 4-45
Return 4-46
Short 4-47
Signed 4-47
sizeof 4-48
Static 4-49
Struct 4-50
Switch 4-52
This 4-54
True 4-56
Typedef 4-56
Union 4-57
Unsigned 4-59
Virtual 4-59
Void 4-61
Volatile 4-61
While 4-62
Support for RTX51 Tiny real-time multitasking operating system 4-63

Chapter 5 - Errors and Warnings
Warnings 5-1
Errors 5-3
Inline asm code 5-68

Chapter 6 - EC++ libraries
List of EC++ libraries 6-1
Double_complex and float_complex 6-2
Abs 6-9

Arg 6-10
Conj 6-10
Cos 6-10
Cosh 6-11
Exp 6-11
Imag 6-11
Log 6-11
log10 6-12
Norm 6-12
Polar 6-13
Pow 6-13
Real 6-14
Sin 6-14
Sinh 6-14
Sqrt 6-15
operator!= 6-15
*operator** 6-15
operator+ 6-16
operator- 6-17
operator/ 6-17
operator<< 6-18
operator== 6-18
Operator>> 6-19
String 6-20
Swap 6-45
Stdiobuf 6-46
Streambuf 6-48
Ios 6-63
Istream 6-86
Ostream 6-97
Iostream 6-102
Iomanip 6-103
New 6-105

Index I-1

PREFACE

PREFACE

8051C++ Software

This manual complements the information provided by Keil Software - 8051 User Manual.

CEIBO C++ is an implementation of the Embedded C++ language as defined by the ISO SC22/WG21 standard. It also defines some dedicated extensions to 8051 microcontroller families.

CEIBO C++ translates EC++ source programs into ANSI C instructions, which can be further compiled by a Keil C51 compilers into object code according to the selected target processor. The C compilers have been updated to support EC++ extensions.

A complete syntax and semantic analysis is performed, including error checking. The generated code preserves source code information with the corresponding line numbers, which is helpful in generating symbolic debugging information compatible with most of the available debuggers, simulators and emulators.

Every compiled source file keeps the information about classes, attributes and methods defined by these classes, including the original name, original type and definition location. The debuggers create accurate source code browsers using this information.

CHAPTER 1

The Embedded C++ language overview

CHAPTER 1

The Embedded C++ language overview

Introduction

The goal of Embedded C++ is to provide embedded systems programmers with a subset of C++ that is easy for the C programmer to understand and use. The subset offers upward compatibility with the full version of Standard C++ and retains the major advantages of C++. Meanwhile, the subset fulfills the particular requirements of embedded systems designs.

The three major requirements of embedded system designs are:

- Avoiding excessive memory consumption
- Taking care not to produce unpredictable responses
- Making code ROMable

Embedded C++ is not a new language specification that competes with existing Standard C++. Rather, it is a pure subset for the practical user of C++. It supports a new methodology in embedded system designs, where a large number of programmers are currently involved.

This compiler includes Embedded C++ libraries, which conforms to Embedded C++ syntax and semantics. The set is limited to the minimum requirements for embedded applications.

Mutable specifier

In embedded systems programming, an object that is specified 'const' is intended to be located in ROM. However, any class members declared 'mutable' can be modified, even if the object itself has been declared 'const'. Therefore, objects that belong to a class that have a 'mutable' member does not fit into ROM.

Also, 'mutable' is specified for a class member. That means, it is specified in the class definition and not in each object declaration. Looking only at the declaration, it is not possible to assume whether or not it is really 'const'.

Therefore, the 'mutable' specifier is not included in the Embedded C++ specifications

Exception handling

Exception handling is useful for dealing with errors. However, there are some drawbacks for embedded systems programmers.

It is difficult to estimate the time between when an exception has occurred and control has passed to a corresponding exception handler, as well as the memory consumption for exception handling.

As control passes from a throw point to a handler, destructors are invoked for all automatic objects constructed since the try block was entered.

If an object has a destructor, then it has to be destroyed by calling the appropriate destructor. That implies execution time to destroy automatic objects.
In general, the exception mechanism requires compiler-generated data structures and runtime support. This sometimes adds unexpected oversize to the program.

In embedded systems it is important for a programmer to be able to easily estimate processing time. Small code size is also a requirement. Therefore, the two drawbacks mentioned previously cannot be ignored for embedded systems programming.

For these reasons, exception handling is not included in the Embedded C++ specifications.

Runtime type identification

To support the runtime type identification (RTTI) facility, there is at least some program size overhead, because type information for polymorph classes is needed. The compiler automatically generates the information, and it would be included in programs that do not use the RTTI facility.

For a program that uses polymorphism heavily, the RTTI facility can provide an advantage. But in the case of programs that make little use of polymorphism and do not need the RTTI facility, it has no merit. Program size is critical for embedded systems applications. Also, programs such as the previously mentioned are advanced and out of the range of the Embedded C++ specifications, which aim at easy understandability and predictability of generated object code by C programmers.

Therefore, RTTI facility is not included in the Embedded C++ specifications.

Namespace

The typical target CPU for the Embedded C++ specifications does not have much memory. Therefore, the size of application programs cannot be very large. Under such conditions, names seldom, if ever, come into conflict. If name conflict becomes a serious problem, it can avoided by using static member of a class. Namespace facility is not essential for the Embedded C++ specifications.

Therefore, they are not included in the Embedded C++ specifications.

Template

Templates are useful for making generic classes or functions. However if used carelessly, templates might cause unexpected code explosion. Furthermore, they may increase the time of compilation. Therefore, the technical committee decided that templates will not be included in Embedded C++ specifications.

Multiple inheritance and virtual inheritance

It is difficult even for an expert programmer to design a class hierarchy using multiple inheritances, or to recognize the overall hierarchy of it and use it correctly. Programs that do not use multiple inheritances appropriately tend to be less readable, less re-usable, and more difficult to maintain.

The Embedded C++ aims at providing specifications that are easy to learn and are used in actual embedded systems programming. Multiple inheritances do not fit this principle.

Therefore, the technical committee decided not to include multiple inheritances in the Embedded C++ specifications.

Virtual inheritance makes sense only if multiple inheritances are used, so virtual inheritance is also absent from the Embedded C++ specifications.

Library

Exception

Since the exception is out of Embedded C++ syntax and semantics, libraries with exception are not supported as the Standard Embedded C++ library.

Template

Since the template is out of Embedded C++ syntax and semantics, libraries with template (the Standard Template class Libraries) are not supported.

Therefore, the technical committee decided not to include the standard template class libraries in the Embedded C++ specification.

Some classes are partly supported as non-template class libraries.

Classes 'string', 'complex', 'ios', 'streambuf', 'istream' and 'ostream' are supported in the Embedded C++ specification. The reasons are given below.

Regarding string libraries, 'string' class which is equivalent to typedef basic_string<char> string in C++, is only supported as a non-template class.

String libraries is used generally, useful for the embedded application and necessary for the support in Embedded C++.

Regarding complex libraries, 'float_complex' and 'double_complex' which are equivalent to typedef complex<float> float_complex typedef complex<double> double_complex in C++, are supported as non-template classes. The reason is that the importance of the complex libraries is increasing for the embedded application and necessary for the support in Embedded C++.

Regarding stream libraries, 'istream' and 'ostream' which are equivalent to typedef basic_istream<char> istream typedef basic_ostream<char> ostream in C++, are supported as non-template classes. The minimum stream library for input/output is necessary for debugging.

wchar_t, long double

The libraries for type of wchar_t or long double are little used for the embedded application and are almost not necessary in the present circumstances.

Therefore, the technical committee decided not to include libraries for type of wchar_t or long double in the Embedded C++ specification.

For example, 'wstream', 'long_double_complex' are not supported.

File-operations

The libraries for file-operations are not supported because of dependence on OS.
Therefore, the technical committee decided not to include libraries for file-operations in the Embedded C++ specification.

Localization libraries

The localization libraries need much memory, are inconvenient to users and have no necessity for most embedded environment.
Therefore, the technical committee decided not to include localization libraries in the Embedded C++ specification.

CHAPTER 2

INSTALLATION

CHAPTER 2

INSTALLATION

Introduction

1. Make sure PK51 Professional Developer Kit of Keil uVision is installed on your PC.

2. Run "setup.exe".

Software and Hardware Requirements

Verify that your PC meets the minimum requirements listed below:

- Windows XP or higher (7/8/10)
- 16 MB RAM at least

3.2 Directories structure

The setup program copies files allowing Embedded C++ support into subfolders of uVision installation folder. The default is "C:\KEIL".

The structure of EC++ installation is following:

Folder	Description
C:\KEIL\C51\BIN	C++ to C conversion tools
C:\KEIL\C51\EXAMPLES\ECPP	EC++ sample applications
C:\KEIL\C51\INC\ECPP	EC++ include files
C:\KEIL\C51\LIB\ECPP	EC++ libraries

CHAPTER 3

EC++ Projects

CHAPTER 3

EC++ Projects

Creating EC++ Projects

Basically they are created like a regular uVision project with slightly differences. Here they are:

- Always select the checkbox "Use extended linker (LX51) instead of BL51" in the dialog "Select Device for Target".

- Select the option "File Extensions, Books and Environment" from the submenu "Project" and enter under "File extensions - C Source Files": "*.c;*.cpp". Then you can add ".cpp" files to your project using the standard uVision dialogs.

- One extra object file is always generated per EC++ project. For successful linkage you should always add to your project the above object file. If you build an executable, the name of this special object file is "project_lnk.obj". If you build an EC++ library, its name is "X_lnk.obj", where X stands for the name of the library. Add the above object to your project even if it yet does not exist - it will be generated right before the linkage.

Creating Custom EC++ Libraries

Ceibo C++ supports creation of library files. Input modules may be EC++ source files, C source files, object files or library files. The library manager is launched with object files and libraries and produces the library file.
All of the above described in "Creating EC++ Projects" is also relevant here.
While creating EC++ libarary always start the library name with the prefix ECPP.

EC++ language extensions for 8051

Ceibo C++ provides a number of extensions for EC++ standard. Most of these provide direct support for elements of the 8051 architecture.
The extensions included are for:

- Memory Types and Areas on the 8051
- Memory Models
- Memory Type Specifiers
- Variable Data Type Specifiers
- Bit variables and bit-addressable data
- Special Function Registers
- Pointers
- Function Attributes

CHAPTER 4

KEYWORDS

CHAPTER 4

KEYWORDS

To facilitate many of the features of the 8051, the EC++ front-end adds a number of new keywords to the scope of the EC++ language. The following is a list of the keywords available in Ceibo C++ :

_at_
alien
auto
bdata
bit
bool
break
case
char
class
code
compact
const
continue
data
default
delete
do
double
else
enum
extern
false
float
for
friend

goto
idata
if
inline
int
interrupt
large
long
new
operator
pdata
priority
private
protected
public
reentrant
register
return
sbit
sfr
sfr16
short
signed
sizeof
small
static
struct
switch
this
true
typedef
task
union
unsigned
using
virtual
void
volatile
while
xdata

auto

The **auto** keyword is the storage-class specifier indicating that the variable has local (automatic) extent. It is the default storage-class specifier for block-scoped variable declarations. Local objects explicitly declared **auto** or **register** or not explicitly declared **static** or **extern** have *automatic storage duratio*n. The storage for these objects lasts until the block in which they are created exits. Declarations of **auto** variables can include initializers. Since variables with **auto** storage class are not initialized automatically, you should either explicitly initialize them when you declare them, or assign them initial values in statements within the block. The values of uninitialized **auto** variables are undefined. (A local variable of **auto** or **register** storage class is initialized each time it comes in scope if an initializer is given.)

Example

```
{
auto int i;     // Explicitly declared as auto.
int      j;     // Implicitly auto.
}
```

Example

```
int  f(auto int n)
{
   return ++n;
}
```

bool

This keyword is an integral type. A variable of this type can have values **true** and **false**. There are no signed, unsigned, short, or long bool types or values.
All conditional expressions now return a value of type **bool**. For example, `i!=0` now returns **true** or **false** depending on the value of `i`.
The values **true** and **false** have the following relationship:

```
!false == true
!true == false
```

The value `false` is converted to zero and the value `true` is converted to one.

In the following statement:

```
if (expres1) statement1;
```

If `expres1` is **true**, `statement1` is always executed; if `expres1` is **false**, `statement1` is never executed.
When a postfix or prefix **++** operator is applied to a variable of type **bool**, the variable is set to **true**. The postfix or prefix **--** operator cannot be applied to a variable of this type.

The **bool** type participates in integral promotions. An r-value of type **bool** can be converted to an r-value of type **int**, with **false** becoming zero and **true** becoming one.

Example

```
void main()
{
 bool a,b;
 a=true;
 b = false;
 if (a||b);
}
```

Example

```
void main()
{
 bool gata;
 char ch;
 gata=false;
 while(!gata)
 {
  ch=getchar();
  if (ch='$')
  {gata=true;continue;}
  putchar(ch+1);//take the next letter of alphabet
 }
}
```

break

The **break** statement shall occur only in an *iteration-statement* or a **switch** statement and causes termination of the most tightly enclosing loop or **switch** statement; control passes to the statement following the terminated statement, if any. The following example illustrates the use of the **break** statement in a **for** loop:

Example

```
for( ; ; )    // No termination condition.
{
    if( List->AtEnd() )
        break;
    List->Next();
}
cout << "Control transfers to here.\n";
```

Example

```
for ( i = 0; i < LENGTH; i++ )  /* Execution returns here when */
{                               /* break statement is executed */
    for ( j = 0; j < WIDTH; j++)
    {
        if ( lines[i][j] == '\0' )
```

```
        {
            lengths[i] = j;
            break;
        }
    }
}
```

The example processes an array of variable-length strings stored in `lines`. The **break** statement causes an exit from the interior **for** loop after the terminating null character (`'\0'`) of each string is found and its position is stored in `lengths[i]`.

Example

```
#include <iostream.h>
void main()
{
  int t;
  for (t=0;t<100;t++)
  {
    cout<<t;
    if (t==10) break;
  }
}
```

The above code displays the first 10 numbers starting from 0. After the first ten values the **for** loop terminates because of the **break** statement.
For an example of using the **break** statement within the body of a **switch** statement, see the **switch** statement.

Example

```
void search(char *name)
{
 do
 {
  if (kbhit()) break;
 } while (!gasit)
}
```

In the above example a key stroke stops the execution of the **search** function.
Let's illustrate the situation when **break** causes the termination of the most inner loop:

Example

```
for (int t=0;t<100;t++)
{
 int contor=1;
 for (;;)
 {
  cout<<contor;
  contor++
  if (contor==10) break;
 }
}
```

The above code displays numbers from 1 to 10 hundred times. Whenever the compiler encounters the **break** statement the control is passed to the extern **for**.

The following example illustrates the use of the **break** statement in a **switch** statement:

Example

```
void menu()
{
 char ch;
 cout<<"1. Spelling check \n";
 cout<<"2. Correct the misspellings \n";
 cout<<"3. Display the misspellings \n";
 cout<<"   Press a key for exiting the program\n";
 cout<<"   Select an option: ";
 cin>>ch; // Read the selection from the keyboard
 switch(ch)
 {
  case '1':
        spell_check();
        break;
  case '2':
        correct_misspell();
        break;
  case '3':
        display_misspell();
        break;
  default:
        cout<<"You didn't select any option!";
 }
}
```

In a **switch** statement the **break** statement is optional; **break** terminates the block of statement associated with a constant. If the **break** is omitted the execution is going on with the statements from the next **case** until a **break** is encountered or until the end of the **switch.**

Example

```
void manev_intr(int i)
{
 int mark;
 mark=-1;
 switch(i)
 {
  case 1:
  case 2:
  case 3: mark=0;break;
  case 4: mark=1;
  case 5: error(mark);break;
  default: processed(I)
 }
}
```

case

The **case** label is used by the **switch** statement. (see **switch** statement)

char

The **char** keyword represents a data type (one of the fundamental data types recognized by C/C++). A variable of type **char** occupies 1 byte of memory.

Example

```
{
 char          ch;
 unsigned char ch;//same as "char  ch;"(unsigned is default)
 char c='1';
 char s=65;
 scanf("%c%c",&s,&c);
 printf("%c%c",s,c);
}
```

Example

```
{
 char sir[10]={'I',' ','l','i','k','e',' ','C','+','+'};
 char str[10]="I like C++";
}
```

Example

```
void main()
{
 char sir[80];
 gets(sir);
 printf("The length is %d",strlen(sir));
}
```

Example

```
void writestring(char *s)
{
 while (*s) putchar(*s++);
}
```

Example

```
search(char *p[],char *name)
{
 int i;
 for (int t=0;p[t];t++)
  if (!strcmp(p[t],nume))
   return t;
 return -1; //not found
}
```

Example

```
{
 const char ar2[] = "128";
 const char ae[34] = "gyugyugu ";
 const char ar5[2][2] = {2,5,6,7};
 char ar[4];
```

```
}
```
Example

```
int f(const char ars[] = "34"){
      printf("%s",ars);
      return 0;
void main()
{
 const char ar2[] = "128";
 f(ar2);
}
```

Example

```
void main()
{
 char ar3[] = (char * const) ar1; // ill-formed, initialize only
with
                                  //string literals or curly
                               braces;
 char ar6[6] = ar;      //error;
 char ar[0]; // error, cannot have zero length array;
 char ar7[2] = "12"; // error, array out of bounds. not space for
                     // trailing '\0'.}
```

class

Syntax

class [*tag* [: *base-list*]]
{
member-list
} [*declarators*];
[**class**] *tag declarators;*

The **class** keyword declares a class type or defines an object of a class type.
The elements of a class definition are as follows:

tag

Names the class type. The tag becomes a reserved word within the scope of the class.

base-list

Specifies the class(es) from which the class is derived (its base classes). Each base class's name can be preceded by an access specifier (**public**, **private** , **protected**) and the **virtual** keyword.

member-list

Declares members or friends of the class. Members can include data, functions, nested classes, enums, bit fields, and type names. Friends can include functions or classes. Explicit data initialization is not allowed. A class type cannot contain itself as a nonstatic member. It can contain a pointer or a reference to itself.

declarators

Declares one or more objects of the class type.

Example

```
class myclass
{
 int i;//private member can be accessed only by members of class B
or by
       //a friend function
public:
 void Set_i(int n) {i=n;}
 int Get_i() {return i;}
}
void main()
{
 myclass ob1,ob2;
 ob1.Set_i(99);
 ob2=ob1;//copy data from ob1 to ob2
 cout<<This is i fom ob2: "<<ob2.Get_i();
}
```

Example

```
class A
{
protected:
  int i,j;
public:
  void Set_ij(int i, int j) {i=a;j=b;}
  void Display_ij() {cout<<i<<"  "<<j<<endl;}
};

class B: public A  //class B can access i,j members from class A
{
int k;//private member can be accessed only by members of class B
or by
      //a friend function
public:
  void Setk() {k=i+j;}
  void Displayk() {cout<<k<<endl;}
};

void main()
{
 B ob;
 ob.Set_ij(2,3);//Set_ij(…) is still public in derived class
 ob.Display_ij();
 ob.Setk()
 ob.Displayk();
}
```

Example

```
class BaseClass
{
protected:
   int protectFunc();
```

```
};

class DerivedClass : public BaseClass
{
public:
   int useProtect()
      { protectFunc(); }    // protectFunc accessible
                           //    from derived class
};

void main()
{
BaseClass aBase;
DerivedClass aDerived;
   aBase.protectFunc();     // Error: protectFunc not
                            //    accessible
   aDerived.protectFunc();  // Error: protectFunc not
                            //    accessible in derived class
}
```

Example

```
// Example of the private keyword
class BaseClass
{ public:
   // privMem accessible from member function
   int pubFunc() { return privMem; }
private:
   void privMem;
};

class DerivedClass : public BaseClass
{
public:
   void usePrivate( int i )
      { privMem = i; }   // Error: privMem not accessible
                        //    from derived class
};

class DerivedClass2 : private BaseClass
{
public:
   // pubFunc() accessible from derived class
   int usePublic() { return pubFunc(); }
};

void main()
{
   BaseClass aBase;
   DerivedClass aDerived;
   DerivedClass2 aDerived2;
   aBase.privMem = 1;     // Error: privMem not accessible
   aDerived.privMem = 1;  // Error: privMem not accessible
                          //    in derived class
```

```
   aDerived2.pubFunc();   // Error: pubFunc() is private in
                          //    derived class
}
```

Example

```
class base
{
public:
      virtual void f()=0;
      virtual void g()=0;
};
base b; // error : cannot instantiate b because of f and g from base

class derived :public base
{
      void f(); // override f from base
};

derived* d = new derived(); // error : cannot instantiate d because
of g from base

class derived_derived: public derived
{
      void g();
};

derived_derived dd[2]; // Ok all pure functions have been overridden
```

Example

```
class point { /* ... */ };
class shape { // abstract class
      point center;
      // ...
public:
      point where() { return center; }
      void move(point p) { center=p; draw(); }
      virtual void rotate(int) = 0; // pure virtual
      virtual void draw() = 0; // pure virtual
      // ...
};

//An abstract class shall not be used as a parameter type, as a
function return type, or as the type of an
//explicit conversion. Pointers and references to an abstract class
can be declared. [Example:
shape x; // error: object of abstract class
shape* p; // OK
shape f(); // error
void g(shape); // error
shape& h(shape&); // OK
```

Example

```
class CLASS1{
public:
      class CLASS1_1{
            int a;
            int method1_1(){};
            public:
            static int m_si;
      };
      int a;
      static int m_si;
      CLASS1_1 ob1_1;
      int method1(){};
};

CLASS1 ob1;

void main(){
      CLASS1::m_si = 1;
      CLASS1::CLASS1_1::m_si = 2;
}
```

Example

```
#include <iostream.h>
void f();
void main()
{
 f();//myclass isn't known here
}
void f()
{
 class myclass
 {
  int i;
 public:
  void Set_i(int n) {i=n;}
  int get_i() {return i;}
 } ob;
 ob.Set_i(10);
 cout<<ob.Get_i();
}
```

Example

```
void f()
{
      {
      class local
      {
      public:
            int local_memb;
```

```
            void local_method(){
                  int local_method_attr;
                  class local_local
                  {
                        int local_local_memb;
                        void local_local_method(){
                              int local_local_method_attr;
                        }
                  };
                  local_local ll;

            }
            class nested
            {
                  int nested_memb;
                  void nested_method(){
                        int nested_method_attr;
                  }
            };
            nested n;
      };
      local l;
      l.local_method();
      }
}

class D
{
      void functie(){}
};
```

Example

```
#include <iostream.h>
class common_class
{
 static int a;
 int b;
public:
 void Get_ij(int i,int j) {a=i;b=j;}
 void Show();
};
int common_class::a;//define a
void common_class::Show()
{
 cout<<This is static a: "<<a;
 cout<<This is b (non static): "<<b<,endl;
}
```

const

const *declaration*
member-function **const**
When modifying a data declaration, the **const** keyword specifies that the object or variable is not modifiable (tells the compiler to prevent the programmer for modifying it). A **const** variable can get an initial value.

```
const int i = 5;
i = 10; // Error
i++;    // Error
```

The **const** keyword instead of the #define preprocessor directive to define constant values.
You can specify the size of an array with a **const** variable as follows:

```
const int maxarray = 255;
char store_char[maxarray];
```

The **const** keyword can also be used in pointer declarations.

```
char *const aptr = mybuf;  // Constant pointer
*aptr = 'a';       // Legal
aptr = yourbuf;    // Error
```

A pointer to a variable declared as **const** can be assigned only to a pointer that is also declared as **const**.

```
const char *bptr = mybuf;  // Pointer to constant data
*bptr = 'a';       // Error
bptr = yourbuf;    // Legal
```

When following a member function's parameter list, the **const** keyword specifies that the function doesn't modify the object for which it is invoked.

```
char *strcpy( char *strDestination, const char *strSource );// the
//source string strSource cannot be modified.
```

When used in conjunction with a function return type, **const** keyword specifies that the result of the function cannot be modified:

```
const char* sir()
{return "1234";}
void main()
{
 char *s1=sir();//Error, the returned value cannot be modified
 const char *s2=sir();//OK
}
```

Declaring a member function with the **const** keyword specifies that the function is a "read-only" function that does not modify the object for which it is called.
To declare a constant member function, place the **const** keyword after the closing parenthesis of the argument list. The **const** keyword is required in both the declaration and the definition. A constant member function cannot modify any data members or call any member functions that aren't constant. You cannot declare constructors or destructors with the **const** keyword.

```
class Date
{
public:
   Date( int mn, int dy, int yr );
   int getMonth() const;       // A read-only function
   void setMonth( int mn );    // A write function;
                               //    cannot be const
private:
   int month;
};

int Date::getMonth() const
{
   return month;        // Doesn't modify anything
}
void Date::setMonth( int mn )
{
   month = mn;          // Modifies data member
}
```

You can call constant member functions only for a constant object. This ensures that the object is never modified.

```
class BirthDay
{
 int mm,dd,yy;
public:
 BirthDay(int m,int d,int y) {mm=m;dd=d;yy=y;}
 int getMonth() const
 {return mm;}
 int getDay() const
 {return dd;}
 int getYear() const
 {return yy;}
 void setMonth(int m) {mm=m;}
 void setDay(int d) {dd=d;}
 void setYear(int y) {yy=y;}
};

void main()
{
 const BirthDay birthday(14,10,1970);
 birthday.getMonth();    // Okay
 birthday.setMonth( 4 ); // Error
```

```
}
```

continue

The **continue** statement forces immediate transfer of control to the loop-continuation statement of the smallest enclosing loop. (The "loop-continuation" is the statement that contains the controlling expression for the loop.) Therefore, the **continue** statement can appear only in the dependent *statement* of an iteration statement (although it may be the sole statement in that *statement*).
The next iteration of a **do**, **for**, or **while** statement is determined as follows:

- Within a **do** or a **while** statement, the next iteration starts by reevaluating the expression of the **do** or **while** statement.
- A **continue** statement in a **for** statement causes the first expression of the **for** statement to be evaluated. Then the compiler reevaluates the conditional expression and, depending on the result, either terminates or iterates the statement body.

Example

```
while ( i-- > 0 )
{
    x = f( i );
    if ( x == 1 )
        continue;
    y += x * x;
}
```

The following example shows how the **continue** statement can be used to bypass sections of code and skip to the next iteration of a loop:

```
#include <conio.h>
// Get a character that is a member of the zero-terminated
//  string, szLegalString. Return the index of the character
//  entered.
int GetLegalChar( char *szLegalString )
{
    char *pch;
    do
    {
        char ch = _getch();
        // Use strchr library function to determine if the
        //  character read is in the string. If not, use the
        //  continue statement to bypass the rest of the
        //  statements in the loop.
        if( (pch = strchr( szLegalString, ch )) == NULL )
            continue;
        // A character that was in the string szLegalString
        //  was entered. Return its index.
        return (pch - szLegalString);
        // The continue statement transfers control to here.
    } while( 1 );
```

```
    return 0;
}
```
The following example displays the number of spaces contained in the string introduced by the user:
```
#include <iostream.h>
void main()
{
 char s[80],*sir;
 int space;
 cout<<"Introduce a string: ";
 cin>>s;
 sir=s;
 for(space=0;*sir;sir++)
 {
  if(*sir!=' ') continue;
  space++;
 }
 cout<<space<<" spaces\n";
}
```
In the following example the **continue** statement forces the conditional test to be executed faster:
```
void code()
{
 char gata,ch;
 gata=0;
 while(!gata)
 {
  ch=getchar();
  if (ch='$')
  {gata=1;continue;}
  putchar(ch+1);//take the next letter of alphabet
 }
}
```

default

The **default** keyword is used within the **switch** statement. For more information see **switch.**

delete

Syntax

delete *pointer*
delete [ ] *pointer*

The **delete** keyword deallocates a block of memory. The argument *pointer* must point to a block of memory previously allocated by the **new** operator. If *pointer* points to an array, place empty brackets before *pointer*.

Example

```
void main()
{
  int *p;
  p=new int (87); // allocates an integer and initializes it
  if (!p)
  {cout<<"Allocation error!\n";exit(1);}
  cout<<"*p= "<<*p;
  delete p;
 }
```

Example

```
void main()
{
  int *p,i;
  p=new int[10]; //allocates memory for an array of 10 elements.
  if (!p)
  {cout<<"Allocation error!\n";exit(1);}
  for (i=0;i<10;i++)
   p[i]=2*i;
  for (i=0;i<10;i++)
   cou<<p[i]<<" ";
  delete []p; //the block of memory pointed to by p is deallocated
}
```

We use the same syntax for dynamically memory allocation/deallocation of objects:

Example

```
class account
{
 double val;
 char name[80];
public:
 void SetVal(double n, char* s) {val=n;strcpy(name,s);}
 void GetVal(double &n,char *s) {n=val;strcpy(s,name);}
};
void main()
{
 account *p;
 char s[80];
 double n;
 p=new account;
 if (!p) {cout<<Allocation error!\n;exit(1)}
 p->SetVal(12387.87,"Ralph Wilson");
 p->GetVal(n,s);
 cout<<s<<" has an account of: "<<n<<"$"<<endl;
 delete p;
}
```

The following example uses an initialization:

Example

```
class account
{
 double val;
 char name[80];
public:
```

```
 account(double n, char *s)
 {
  val=n;
  strcpy(name,s);
 }
 ~acount()
 {
  cout<<Destructor;  cout<<name<<endl;
 }
 void GetVal(double &n,char *s) {n=val;strcpy(s,name);}
};
void main()
{
 account *p;
 char s[80];
 double n;
 p=new acount(12387.87,"Ralph Wilson");
 if (!p) {cout<<Allocation error!\n;exit(1)}
 p->GetVal(n,s);
 cout<<s<<" has an account of: "<<n<<"$"<<endl;
 delete p;
}
```

The following example illustrates the memory allocation/deallocation for an array of objects:

```
class account
{
 double val;
 char name[80];
public:
 account() {}   //a constructor without parameters
 account(double n, char *s)
 {
  val=n;
  strcpy(name,s);
 }
 ~acount()
 {
  cout<<Destructor;  cout<<name<<endl;
 }
 void SetVal()
 {
  val=n;  strcpy(name,s);
 }
 void GetVal(double &n,char *s) {n=val;strcpy(s,name);}
};
void main()
{
 account *p;
 char s[80];
 double n;
 int i;
 p=new account[3];//allocates memory for the whole array
```

```
 if (!p) {cout<<Allocation error!\n;exit(1)}
 p[0].SetVal(1238.87,"Ralph Wilson");
 p[1].SetVal(144.00,"A.C. Conners");
 p[2].SetVal(-11.23,"B.J. Huston");
 for (i=0;i<3;i++)
 {
  p[i].GetVal(n,s);
  cout<<s<<" has an account of: "<<n<<"$"<<endl;
 }
 delete []p;//the destructor is called for each object member of the
            //array
}
```

The **delete** operator can be overloaded in order to perform certain actions. The following example illustrates the overloading of the **delete** operator relative to a class:

```
class place
{
 int longitude,latitude;
public:
 place() {}
 place(int lg,int lt)
 {
  longitude=lg;
  latitude=lt;
 }
 void show()
 {
  cout<<longitude<<" "<<latitude<<endl;
 }
 void *operator new(size_t dim);
 void operator delete(void *p);
};

void *place::operator new(size_t dim)
 {
  cout<<"In my new\n";
  return malloc(dim);
 }
void place::operator delete(void *p)
 {
  cout<<"In my delete\m";
  free(p);
 }
}
void main()
{
 place *p1,*p2;
 p1=new place(10,20);
 if(!p1) {cout<<"Allocation error!\n";exit(1);}
 p2=new place(-10,-20);
 if (!p2) {cout<<"Allocation error!\n";exit(1);}
 p1->show();
 p2->show();
```

```
 delete p1;
 delete p2;
}
```
The next example illustrates a global overloading of **new** and **delete** operators:
```
class place
{
 int longitude,latitude;
public:
 place() {}
 place(int lg,int lt)
 {
  longitude=lg;
  latitude=lt;
 }
 void show()
 {
  cout<<longitude<<" "<<latitude<<endl;
 }
};
//new global
void *operator new(size_t dim)
{
 return malloc(dim);
}
//delete global
void operator delete(void *p)
{
 free(p);
}
void main()
{
 place *p1,*p2;
 p1=new place(10,20);
 if(!p1) {cout<<"Allocation error!\n";exit(1);}
 p2=new place(-10,-20);
 if (!p2) {cout<<"Allocation error!\n";exit(1);}
 float *f=new float;            //uses the overloaded new too
 if (!f) {cout<<"Allocation error!\n";exit(1);}
 *f=10.10;
 cout<<*f<<endl;
 p1->show();
 p2->show();
 delete p1;
 delete p2;
 delete f;                      //uses the overloaded delete too
}
```

do

The **do** statement executes a *statement* repeatedly until the specified termination condition (the *expression*) evaluates to zero. The test of the termination condition is

made after each execution of the loop; therefore, a **do** loop executes one or more times, depending on the value of the termination expression. The **do-while** statement can also terminate when a **break**, **goto**, or **return** statement is executed within the statement body.

Syntax

```
do
statement
while( expression );
```

The following function uses the **do** statement to wait for the user to press a number less or equal to 100:

Example

```
do
{
  scanf("%d",&num)
}while (num>100);
```

The following example shows the use of the **do-while** loop to create a menu selection function.

```
void menu()
{
 char ch;
 cout<<"1. Spelling check \n";
 cout<<"2. Correct the misspellings \n";
 cout<<"3. Display the misspellings \n";
 cout<<"   Press a key for exiting the program\n";
 cout<<"   Select an option: ";
 do {
  cin>>ch; // Read the selection from the keyboard
  switch(ch)
  {
   case '1':
        spell_check();
        break;
   case '2':
        correct_misspell();
        break;
   case '3':
        display_misspell();
        break;
  }
 }while (ch!='1' && ch!='2' &&ch!='3');
}
```

The following example prompts users for a password and continued to prompt them until they enter one that matches the value stored in checkword.

```
#include <stdio.h>
#include <string.h>
void main ()
{
  char checkword[80] = "password";
  char password[80]  = "";
```

```
  do {
    printf ("Enter password: ");
    scanf("%s", password);
  } while (strcmp(password, checkword));
}
```

double

The **double** keyword keyword represents a data type (one of the fundamental data types recognized by C/C++). A variable of type **double** occupies 8 bytes of memory. The value representation of floating-point types is implementation-defined.

Example

```
double d;
long double dl;
double f(double i);
double g(int j);
```

else

See the **if** keyword.

Enum

The **enum** keyword specifies an enumerated type.

Syntax

enum [tag] {enum-list} [declarator];

An enumerated type is a user-defined type consisting of a set of named constants called enumerators. By default, the first enumerator has a value of 0, and each successive enumerator is one larger than the value of the previous one, unless you explicitly specify a value for a particular enumerator. Enumerators needn't have unique values. The name of each enumerator is treated as a constant and must be unique within the scope where the **enum** is defined. An enumerator can be promoted to an integer value. However, converting an integer to an enumerator requires an explicit cast, and the results are not defined. Enumerators defined within a class are accessible only to member functions of that class unless qualified with the class name (for example, `class_name::enumerator`). You can use the same syntax for explicit access to the type name (`class_name::tag`).

Example

```
enum { a, b, c=0 };
```

```
enum { d, e, f=e+2 };
```

The above example defines a, c, and d to be zero, b and e to be 1, and f to be 3.

Example

```
enum Days
{
      saturday,     // saturday = 0 by default
      sunday = 0,   // sunday = 0 as well
      monday,       // monday = 1
      tuesday,      // tuesday = 2
      wednesday,    // wednesday = 3
      thursday,     // thursday = 4
      friday        // friday = 5
 }today             // Variable today has type Days

int tuesday;        // Error, redefinition of tuesday

enum Days yesterday;
Days tomorrow;

yesterday = monday;

int i = tuesday;         // Legal; i = 2
yesterday = 0;           // Error; no conversion
yesterday = (Days)0;     // Legal, but results undefined
```

Example

```
enum  coins {penny,nickel,dime,quarter,dolar,half_dolar};
enum coins money;
money=dime;//OK
if (money==quarter)
 printf("Money is a quarter.\n");
printf("%d %d",penny,dime);//displays 0 2
```

The value of the enumeration can be used like an index:

Example

```
char
name[][15]={"penny","nickel","dime","quarter","dolar","half_dolar"}
;
money=dolar;
printf("%s",name[money]);
```

The value of an enumerator or an object of an enumeration type is converted to an integer by integral promotion.

Example

```
enum color { red, yellow, green=20, blue };
color col = red;
color* cp = &col;
if (*cp == blue) // ...
```

makes `color` a type describing various colors, and then declares `col` as an object of that type, and `cp` as a

pointer to an object of that type. The possible values of an object of type `color` are `red`, `yellow`, `green`, `blue`; these values can be converted to the integral values 0, 1, 20, and 21. Since enumerations are distinct types, objects of type `color` can be assigned only values of type `color`.

```
color c = 1;     // error: type mismatch,
                 // no conversion from int to color
int i = yellow; // OK: yellow converted to integral value 1
                 // integral promotion
```

Example

```
class X {
public:
enum direction { left='l', right='r' };
int f(int i)
{ return i==left ? 0 : i==right ? 1 : 2; }
};
void g(X* p)
{
direction d; // error: direction not in scope
int i;
i = p->f(left); // error: left not in scope
i = p->f(X::right); // OK
i = p->f(p->left); // OK
// ...
}
```

extern

extern *declarator* // used when variable or function has external linkage
extern *string-literal declarator* // used when linkage conventions of another
// language are being used for the declarator
extern *string-literal* **{** *declarator-list* **}** // used when linkage conventions of another
// language are being used for the declarators

The **extern** keyword declares a variable or function and specifies that it has external linkage (its name is visible from files other than the one in which it's defined). When modifying a variable, **extern** specifies that the variable has static duration (it is allocated when the program begins and deallocated when the program ends). The variable or function may be defined in another source file, or later in the same file. If a function definition does not include a *storage-class-specifier*, the storage class defaults to **extern**. You can explicitly declare a function as **extern**, but it is not required.

If the declaration of a function contains the *storage-class-specifier* **extern**, the identifier has the same linkage as any visible declaration of the identifier with file scope. If there is no visible declaration with file scope, the identifier has external linkage. If an identifier has file scope and no *storage-class-specifier*, the identifier has external linkage. External linkage means that each instance of the identifier denotes the same object or function.

When used with a string, **extern** specifies that the linkage conventions of another language are being used for the declarator(s).
string-literal is the name of a language. The language specifier "C++" is the default. "C" is the only other language specifier currently supported by this compiler. This allows you to use functions or variables defined in a C module.
Example

```
// Example of the extern keyword
extern "C" int printf( const char *, ... );

extern "C"
{
   int getchar( void );
   int putchar( int );
}
```

Example

```
/***        SOURCE FILE ONE     ***/

extern int i;                /* Reference to i, defined below */
void next( void );           /* Function prototype            */

void main()
{
    i++;
    printf( "%d\n", i );     /* i equals 4 */
    next();
}

int i = 3;                   /* Definition of i */
void next( void )
{
    i++;
    printf( "%d\n", i );     /* i equals 5 */
    other();
}

/***        SOURCE FILE TWO     ***/

extern int i;                /* Reference to i in  */
                             /* first source file  */
void other( void )
{
    i++;
    printf( "%d\n", i );     /* i equals 6 */
}
```

The two source files in this example contain a total of three external declarations of `i`. Only one declaration is a "defining declaration." That declaration, `int i = 3;` defines the global variable `i` and initializes it with initial value 3. The "referencing" declaration of `i` at the top of the first source file using **extern** makes the global variable visible prior to its defining declaration in the file. The referencing

declaration of `i` in the second source file also makes the variable visible in that source file. If a defining instance for a variable is not provided in the translation unit, the compiler assumes there is an

```
extern int x;
```

referencing declaration and that a defining reference

```
int x = 0;
```

appears in another translation unit of the program.

All three functions, `main, next, and other`, perform the same task: they increase `i` and print it. The values 4, 5, and 6 are printed.

If the variable `i` had not been initialized, it would have been set to 0 automatically. In this case, the values 1, 2, and 3 would have been printed.

There is another use (an optional use) for **extern** keyword: when you use a global variable within the body of a function you can declare it as **extern** like in the following example:

```
int first,last;    //first and last are defined like global variables
void main()
{
extern int first; //optional use of the extern declaration;
                  //it isn't necessary
}
```

false

This keyword is one of the two values that a variable of type **bool** can have. If `i` is of type **bool**, then the statement `i=false;` assigns **false** to `i`. See the **bool** keyword.

float

The **float** keyword keyword represents a data type (one of the fundamental data types recognized by C/C++). A variable of type **float** occupies 4 bytes of memory. The value representation of floating-point types is implementation-defined.

Example

```
float f=1.2;
float myFunction(float i);
float g(int j);
```

for

for statement lets you repeat a statement or compound statement a specified number of times. The body of a **for** statement is executed zero or more times until an optional

condition becomes false. You can use optional expressions within **for** statement to initialize and change values during the **for** statement's execution.

Syntax

iteration-statement :
for **(** [*init-expression*] **;** [*cond-expression*] **;** [*loop-expression*] **)**
statement

Execution of a **for** statement proceeds as follows:

1. The *init-expression*, if any, is evaluated. This specifies the initialization for the loop. There is no restriction on the type of *init-expression*.

2. The *cond-expression*, if any, is evaluated. This expression must have arithmetic or pointer type. It is evaluated before each iteration. Three results are possible:
   - If *cond-expression* is true (nonzero), *statement* is executed; then *loop-expression*, if any, is evaluated. The *loop-expression* is evaluated after each iteration. There is no restriction on its type. Side effects will execute in order. The process then begins again with the evaluation of *cond-expression*.

   - If *cond-expression* is omitted, *cond-expression* is considered true, and execution proceeds exactly as described in the previous paragraph. A **for** statement without a *cond-expression* argument terminates only when a **break** or **return** statement within the statement body is executed, or when a **goto** (to a labeled statement outside the **for** statement body) is executed.

   - If *cond-expression* is false (0), execution of the **for** statement terminates and control passes to the next statement in the program.

A **for** statement also terminates when a **break**, **goto**, or **return** statement within the statement body is executed. A **continue** statement in a **for** loop causes *loop-expression* to be evaluated. When a **break** statement is executed inside a **for** loop, *loop-expression* is not evaluated or executed. This statement

```
for( ;; );
```

is the customary way to produce an infinite loop which can only be exited with a **break**, **goto**, or **return** statement.

The following example waits for the user to press key 'A':

```
char ch='\0';
for(;;)
{
 ch=getchar();       //read a character
 if (ch=='A') break; //terminates the for loop
}
printf("You pressed A");
```

The following example counts `space (' ')` and `tab ('\t')` characters in the array of characters named `line` and replaces each tab character with a space.

Example

```
for ( i = space = tab = 0; i < MAX; i++ )
{
    if ( line[i] == ' ' )
        space++;
    if ( line[i] == '\t' )
    {
        tab++;
        line[i] = ' ';
    }
}
```

The following example removes spaces from the beginning of the string `str`. It uses a **for** loop without body:

```
for(;*str==' ';str++);
```

You can use the **for** loop to generate a delay:

```
for(int t=0;t<SOME_VALUES;t++);
```

If you want to control the **for** loop by more than one variable you will use the **, operator:**

```
for(x=0,y=0;x+y,10;++x)
{
 y=getchar();
 y=y-'0';
 .
 .
 .
}
```

Other examples:

```
void message(int line,char *msg)
{
 int i,j;
 for (i=1,j=strlen(msg);i<j;i++,j--)
 {
  _settextposition(line,i);printf("%c",msg[i-1]);
  _settextposition(line,j);printf("%c",msg[j-1]);
 }
}
```

For example, you can use the following function to enter a user in a remote system. The user can try three times to enter the password. The **for** loop ends when all the trials ended or when the user enters the correct password:

```
void sign()
{
 char sir[20];
 int x;
 for (x=0;x<3 && strcmp(sir,"parola");++x)
 {printf("Enter the password: ");gets(sir);}
 if (x==3) return;
/*else login the user to … */
}
```

The following **for** loop uses the initialization and incrementation sequences in an unusual but perfect valid manner:

```
int numsquare(int num);
int numread();
int prompt();
void main()
{
 int t;
 for (prompt();t=numread();prompt()) //if the number=0 the for loop
      numsquare(t);                  //terminates.
}

int prompt()
{
 printf("Enter a number: ");
 return 0;
}
int numread()
{
 int t;
 scanf("%d",&t);
 return t;
}
int numsquare(int num)
{
 printf("%d",num*num);
 return num*num;
}
```

friend

friend *class-name;*
friend *function-declarator;*

The **friend** keyword allows a function or class to gain access to the private and protected members of a class. The name of a friend is not in the scope of the class, and the friend is

not called with the member access operators unless it is a member of another class.

The following example illustrates the differences between members and friends:

```
class X {
int a;
friend void friend_set(X*, int);
public:
void member_set(int);
};
void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }
void f()
{
X obj;
```

```
friend_set(&obj,10);
obj.member_set(10);
}
```
Example
```
class A {
typedef int I; // private member
I f();
friend I g(I);
static I x;
};
A::I A::f() { return 0; }
A::I g(A::I p = A::x);
A::I g(A::I p) { return 0; }
A::I A::x = 0;
```
Example of a friend class
```
class YourClass
{
friend class YourOtherClass;  // Declare a friend class
private:
   int topSecret;
};

class YourOtherClass
{
public:
   void change( YourClass yc );
};

void YourOtherClass::change( YourClass yc )
{   yc.topSecret++;    }// Can access private data
```
Example
```
class A {
class B { };
friend class X;
};
class X : A::B { // ill-formed: A::B cannot be accessed
// in the base-clause for X
A::B mx; // OK: A::B used to declare member of X
class Y : A::B { // OK: A::B used to declare member of X
A::B my; // ill-formed: A::B cannot be accessed
// to declare members of nested class of X
};
};
```
Example
```
class X {
enum { a=100 };
friend class Y;
};
class Y {
int v[X::a]; // OK, Y is a friend of X
};
```

```
class Z {
int v[X::a]; // error: X::a is private
};
```
Example of a friend function
```
class Complex
{
public:
   Complex( float re, float im );
   friend Complex operator+( Complex first, Complex second );
private:
   float real, imag;
};

Complex operator+( Complex first, Complex second )
{
   return Complex( first.real + second.real,
                   first.imag + second.imag );
}
```
Friendship is neither inherited nor transitive.

Example
```
class A {
friend class B;
int a;
};
class B {
friend class C;
};
class C {
void f(A* p)
{
p->a++; //error: C is not a friend of A
// despite being a friend of a friend
}
};
class D : public B {
void f(A* p)
{
p->a++; //error: D is not a friend of A
// despite being derived from a friend
}
```

goto

The **goto** statement performs an unconditional transfer of control to the named label. The label must be in the current function and can appear before only one statement in the same function.

Syntax

```
statement :
labeled-statement
jump-statement
```

```
jump-statement :
goto identifier ;
labeled-statement :
identifier : statement
```

A statement label is meaningful only to a **goto** statement; in any other context, a labeled statement is executed without regard to the label.

A *jump-statement* must reside in the same function and can appear before only one statement in the same function. The set of *identifier* names following a **goto** has its own name space so the names do not interfere with other identifiers. Labels cannot be redeclared.

Example

```
void main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 3; j++ )
        {
            printf( " Inner loop executing. j = %d\n", j );
            if ( i == 5 )
                goto stop;
        }
    }
    /* This message does not print: */
    printf( "Loop exited. i = %d\n", i );
    stop: printf( "Jumped to stop. i = %d\n", i );
}
```

The following code fragment illustrates use of the **goto** statement and an *identifier* label to escape a tightly nested loop:

```
for( p = 0; p < NUM_PATHS; ++p )
{
    NumFiles = FillArray( pFileArray, pszFNames )
    for( i = 0; i < NumFiles; ++i )
    {
        if( (pFileArray[i] = fopen( pszFNames[i], "r" )) == NULL )
            goto FileOpenError;
        // Process the files that were opened.
    }
}
FileOpenError:
    cout << "Fatal file open error. Processing interrupted.\n" );
```

It is illegal to jump past a declaration with an initializer unless: the declaration is enclosed in a block that is not entered or the jump is from a point where the variable has already been initialized.

The following is an example of this error:

```
void func()
{
   goto label1;
```

```
   int i = 1;          // error, initialization skipped
   {
      int j = 1;       // OK, this block is never entered
   }
label1:;
}
```

Here is another example of this error:

```
void f()
{
// ...
goto lx; // ill-formed: jump into scope of a
// ...
ly:
X a = 1;
// ...
lx:
goto ly; // OK, jump implies destructor
// call for a followed by construction
// again immediately following label ly
}
```

if

The **if** statement controls conditional branching. The body of an **if** statement is executed if the value of the expression is nonzero.

Syntax

if(*expression* **)**
statement1
[**else**
statement2]

The **if** keyword executes *statement1* if *expression* is true (nonzero); if **else** is present and *expression* is false (zero), it executes *statement2*. After executing *statement1* or *statement2*, control passes to the next statement.

Example

```
if ( i > 0 )
    y = x / i;
else
{
    x = i;
    y = f( x );
}
```

In this example, the statement `y = x/i;` is executed if `i` is greater than 0. If `i` is less than or equal to 0, `i` is assigned to `x` and `f( x )` is assigned to `y`. Note that the statement forming the **if** clause ends with a semicolon.

When nesting **if** statements and **else** clauses, use braces to group the statements and clauses into compound statements that clarify your intent. If no braces are present, the compiler resolves ambiguities by associating each **else** with the closest **if** that lacks an **else**.

Example

```
if ( i > 0 )            /* Without braces */
    if ( j > i )
        x = j;
    else
        x = i;
```

The **else** clause is associated with the inner **if** statement in this example. If `i` is less than or equal to 0, no value is assigned to `x`.

Example

```
if ( i > 0 )             /* With braces */
{
    if ( j > i )
        x = j;
}
else
    x = i;
```

The braces surrounding the inner **if** statement in this example make the **else** clause part of the outer **if** statement. If `i` is less than or equal to 0, `i` is assigned to `x`.
A name introduced by a declaration in a *condition* (either introduced by the *type-specifier-seq* or the *declarator* of the condition) is in scope from its point of declaration until the end of the substatements controlled by the condition. If the name is redeclared in the outermost block of a substatement controlled by the condition, the declaration that re-declares the name is ill-formed:

Example

```
if (int x = f()) {
int x; // ill-formed, redeclaration of x
}
else {
int x; // ill-formed, redeclaration of x
}
```

A usual programming construction is **if-else-if.** The syntax is:

```
if (expression)
    statement1;
else if (expression)
    statement2;
else if (expression)
    statement3;
    .
    .
    .
else
    statementN;
```

The conditions are evaluated from bottom to top. As soon as a true condition is encountered, the associated statement will be executed and the rest of the construction will be ignored. If none of the conditions is true, the final **else** will be executed. That means that if all other conditions fail, the last **else** statement will be

executed. If there is no final **else** there will be no action, in case that other conditions are **false.**

Example

```
void main()
{
 int magic,guess;
 magic=rand();//generates the magic number
 printf("Guess the number: ");
 scanf("%d",&guess);
 if (guess==magic)
 {
  printf("**Correct**");
  printf(" %d is the magic number",magic);
 }
 else if (guess>magic)
    printf("Error, too big!");
 else printf("Error, too small!");
}
```

inline

The **inline** specifier instructs the compiler to replace function calls with the code of the function body. This substitution is "inline expansion" (sometimes called "inlining"). Inline expansion alleviates the function-call overhead at the potential cost of larger code size. A function defined within a class definition is an inline function. The inline specifier shall not appear on a block scope function declaration. As with normal functions, there is no defined order of evaluation of the arguments to an inline function. In fact, it could be different from the order in which the arguments are evaluated when passed using normal function call protocol.

An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears. No diagnostic is required. An inline function with external linkage shall have the same address in all translation units.

Example

```
#include <iostream.h>
inline int max(int a, int b)
{
 return a>b ? a : b;
}
void main()
{
 cout<<max(10,20);
 cout<<" "<<max(99,88);
 return 0;
}
```

In the above example **max()** function is expanded inline instead of being called.

A class's member functions can be declared inline either by using the **inline** keyword or by placing the function definition within the class definition.

Example

```
class MyClass
{
public:
   void print() { cout << i << ''; }  // Implicitly inline
private:
   int i;
};
```

Example

```
class myclass
{
 int a,b;
public:
 void init(int i,int j);
 void show();
};
inline void clasamea::init(int i,int j)
{
 a=i;b=j;
}
inline void show()
{
 cout<<a<<" "<<b<<endl;
}
```

int

The **int** keyword represents a data type (one of the fundamental data types recognized by C/C++). The size of a signed or unsigned **int** item is the standard size of an integer on a particular machine. For example, in 16-bit operating systems, the **int** type is usually 16 bits, or 2 bytes. In 32-bit operating systems, the **int** type is usually 32 bits, or 4 bytes. Thus, the **int** type is equivalent to either the **short int** or the **long int** type, and the **unsigned int** type is equivalent to either the **unsigned short** or the **unsigned long** type, depending on the target environment. The **int** types all represent signed values unless specified otherwise.

Example

```
int i, j, n;
short int si;
unsigned int ui;
void f(int i);
float g(int i);
```

long

The **long** keyword is a conversion specifier. For more information about conversion specifiers see **signed.**
If the **long** keyword is used alone or together with **int** it designates a 32-bit integer. The **long** keyword can be preceded by either the keyword **signed** or the keyword **unsigned**. The **int** keyword is optional and can be omitted.
`[ signed | unsigned ] long [ int ]` *`declarator-list`*
The **long** keyword can also be used together with **double** type. In this case it designates a 80 bits double**.**
`long double` *`declarator-list`*

Example

```
long int    i;
long        i;//same as "long int i;"
long double d;
unsigned long int l;
```

new

Syntax

`new` [*`placement`*] *`type-name`* [*`initializer`*]
`new` [*`placement`*] **`(`** *`type-name`* **`)`** [*`initializer`*]
The **new** keyword allocates memory for an object of *type-name* from the free store and returns a suitably typed, nonzero pointer to an object. If unsuccessful, by default **new** returns zero.

The following list describes the elements of **new**:

placement

Provides a way of passing additional arguments if you overload **new**.

type-name

Specifies type to be allocated. If the type specification is complicated, it can be surrounded by parentheses to force the order of binding.

initializer

Provides a value for the initialized object. Initializers cannot be specified for arrays. The **new** operator will create arrays of objects only if the class has a default constructor.

The **new** operator cannot be used to allocate a function; however, it can be used to allocate a pointer to a function.

When **new** is used to allocate a single object, it yields a pointer to that object; the resultant type is *new-type-name* ***** or *type-name* *****. When **new** is used to allocate a singly dimensioned array of objects, it yields a pointer to the first element of the array, and the resultant type is *new-type-name* ***** or *type-name* *****. When **new** is used to allocate a multidimensional array of objects, it yields a pointer to the first element of the array, and the resultant type preserves the size of all but the leftmost array dimension. For example

```
new float[10][25][10]
```

yields type `float (*)[25][10]`. Therefore, the following code will not work because it attempts to assign a pointer to an array of `float` with the dimensions `[25][10]` to a pointer to type `float`:

```
float *fp;
fp = new float[10][25][10];
```

The correct expression is:

```
float (*cp)[25][10];
cp = new float[10][25][10];
```

The definition of `cp` allocates a pointer to an array of type `float` with dimensions `[25][10]` - it does not allocate an array of pointers.

All but the leftmost array dimensions must be constant expressions that evaluate to positive values; the leftmost array dimension can be any expression that evaluates to a positive value. When allocating an array using the **new** operator, the first dimension can be zero - the **new** operator returns a unique pointer.

The *type-specifier-list* cannot contain **const**, **volatile**, class declarations, or enumeration declarations. Therefore, the following expression is illegal:

```
volatile char *vch = new volatile char[20];
```

The **new** operator does not allocate reference types because they are not objects.

If there is insufficient memory for the allocation request, by default **operator new** returns **NULL.**

For more examples see the **delete** operator.

operator

Syntax

type **operator** *operator-symbol* **(** *parameter-list* **)**

The **operator** keyword declares a function specifying what *operator-symbol* means when applied to instances of a class. This gives the operator more than one meaning, or "overloads" it. The compiler distinguishes between the different meanings of an operator by examining the types of its operands.

You can overload the following operators: **+ - * / % ^ ! = < > += -= ^= &= |= << >> <<= <= >= && || ++ -- ( ) [ ] new delete & | ~ *= /= %= >>= == != , -> ->***

If an operator can be used as either a unary or a binary operator, you can overload each use separately.

You can overload an operator using either a nonstatic member function or a global function that's a friend of a class. A global function must have at least one parameter that is of class type or a reference to class type.

If a unary operator is overloaded using a member function, it takes no arguments. If it is overloaded using a global function, it takes one argument.

If a binary operator is overloaded using a member function, it takes one argument. If it is overloaded using a global function, it takes two arguments.

You cannot define new operators, such as **.

You cannot change the precedence or grouping of an operator, nor can you change the numbers of operands it accepts.

You cannot redefine the meaning of an operator when applied to built-in data types.

Overloaded operators cannot take default arguments.

You cannot overload any preprocessor symbol, nor can you overload the following operators: **. .* :: ?:**

The assignment operator has some additional restrictions. It can be overloaded only as a nonstatic member function, not as a friend function. It is the only operator that cannot be inherited; a derived class cannot use a base class's assignment operator.

Example

```
class Complex
{
public:
   Complex( float re, float im );
   Complex operator+( Complex &other );
   friend Complex operator+( int first, Complex &second );
private:
   float real, imag;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other )
{
return Complex( real + other.real, imag + other.imag );
```

```
};

// Operator overloaded using a friend function
Complex operator+( int first, Complex &second )
{
return Complex( first + second.real, second.imag );
}
```

Example for **[]** operator

```
class myclass
{
 int a[3];
public:
 myclass(int i,int j,int k)
 {a[0]=i;a[1]=j;a[2]=k;}
 int &operator[](int i)
 {return a[i];}
};
void main()
{
 myclass ob(1,2,3);
 cout<<ob[1];//displays 2
 cout<<" ";
 ob[1]=25;//[]is in the left of =
 cout<<ob[1];//displays 25
}
```

Example for **()** operator

```
class place
{
 int longitude,latitude;
public:
 place() {}
 place(int lg,int lt)
 {
  longitude=lg;
  latitude=lt;
 }
 void show()
 {
  cout<<longitude<<" "<<latitude<<endl;
 }
 place operator+(place op2);
 place operator ()(int i,int j);
};
place place::operator+(place op2)
{
 place temp;
 place.longitude=op2.longitude+ longitude;
 place.latitude=op2. Latitude+ latitude;
 return temp;
}
place place::operator()(int i,int j)
{
```

```
 longitude=i;
 latitude=j;
 return *this;
}
void main()
{
 place ob1(10,20),ob2(1,1);
 ob1.show();
 ob1(7,8);//can be executed alone
 ob1.show();
 ob1=ob2+ob1(10,10);//can be used in expressions
 ob1.show();
}
```

Example for **->** operator

```
class myclass
{
 public:
 int i;
 myclass operator->() {return this;}
};
void main()
{
 myclass ob;
 ob->i;//it's the same with ob.i
}
```

Example for **<<** operator

```
class PhoneBook
{
 char name[80];
 int zipcode;
 int prefix;
 int number;
public:
 PhoneBook() {}
 PhoneBook(char *n,int a,int p,int num)
 {
  strcpy(name,n);
  zipcode=a;
  prefix=p;
  number=num;
 }
friend ostream &operator<<(ostream &os,PhoneBook &pb);
friend istream &operator<<(istream &is,PhoneBook &pb);
};
ostream &operator<<(ostream &os,PhoneBook &pb)
{
 os<<pb.name<<" ";
 os<<"("<<pb.zipcode<<")";
 os<<pb.prefix<<"-"<<pb.number<<endl;
 return os;
}
istream &operator<<(istream &is,PhoneBook &pb)
```

```
{
 cout<<"Enter the name:";
 is>>pb.name;
 cout<<"Enter the zipcode";
 is>>pb.zipcode;
 cout<<"Enter the prefix";
 is>>pb.prefix;
 cout<<"Enter the number";
 is>>pb.number;
 return is;
}
void main()
{
 PhoneBook pb;
 cin>>a;
 cout<<a;
}
```

For examples about overloading the **delete** and **new** operators see the **delete** operator.

private

Syntax
private: [*member-list*]
private *base-class*

When preceding a list of class members, the **private** keyword specifies that those members are accessible only from member functions and friends of the class. This applies to all members declared up to the next access specifier or the end of the class.
When preceding the name of a base class, the **private** keyword specifies that the public and protected members of the base class are private members of the derived class.
Default access of members in a class is private. Default access of members in a structure or union is public.
Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

Example

```
class X
{
 int a;     // X::a is private by default
};
```

Example

class C
{

private:

```
      C(){} // private constructor
public:
      C(int a, int b)

      {}

};


void main ()

{

 C c; // no access

 C c2(1, 2); //  Ok

 C* pc = new C; // no access to private constructor

 C* pc2 = new C(1,2); // Ok : public constructor


}
```

Example

```
class BaseClass
{
private:
  int priv_b;
public:
  int Geti() {return i;}
};

class Derived1: public BaseClass
{
public:
 usePrivate(int i) {priv_b=i;}      //Error: priv_b not accessible
                                    //from derived class
};

class Derived2: private BaseClass
{
public:
// pubFunc() accessible from derived class
int usePublic() { return pubFunc(); }
```

```
};

void main()
{
   BaseClass aBase;
   Derived1 aDerived;
   Derived2 aDerived2;
   aBase.priv_b = 1;     //Error:priv_b not accessible
   aDerived.priv_b = 1;  //Error:priv_b not accessible in derived
class
   aDerived2.pubFunc();  //Error:pubFunc() is private in derived
class
}
```

Example

```
class B; // forward declaration
class A
{
friend void both_friend(A& ra, B& rb);
private:
 int private_mA;
 friend void Afriend(A a);
public:
 int public_mA;
};

void Afriend(A a)
{
 a.private_mA = 3; // Ok: private_m is private in class
}

class B: private A
{
 friend void both_friend(A& ra, B& rb);
private:
 int private_mB;
public:
 friend void  Bfriend()
 {
  B b;
  b.public_mA=1;    // Ok: m is private in class B
  b.A::public_mA=4; // Ok: same as above
  b.private_mA = 2; // Wrong: private_m is inherited in B with no
acces
                    // rights
 }
};

void both_friend(A& ra, B& rb)
{
 ra.private_mA = 10; // Ok: friend of A
 rb.private_mB = 11; // Ok: friend of B
}
```

protected

Syntax
protected: [*member-list*]
protected *base-class*
When preceding a list of class members, the **protected** keyword specifies that those members are accessible only from member functions and friends of the class and its derived classes. This applies to all members declared up to the next access specifier or the end of the class.
When preceding the name of a base class, the **protected** keyword specifies that the public and protected members of the base class are protected members of the derived class.
Default access of members in a class is private. Default access of members in a structure or union is public.
Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

Example

```
class BaseClass
{
protected:
   int protectFunc();
};

class DerivedClass : public BaseClass
{
public:
   int useProtect()
      { protectFunc(); }    // protectFunc accessible
                           //  from derived class
};

void main()
{
BaseClass aBase;
DerivedClass aDerived;
aBase.protectFunc();     // Error: protectFunc not accessible
aDerived.protectFunc();  // Error: protectFunc not
                         // accessible in derived class
}
```

Example

```
class Base
{
 protected:
  int i,j;
 public:
  void Set_ij(int a,int b) {i=a;j=b;}
```

```
  void Show_ij()(cout<<i<<" "<<j<<endl;}
};
class Derived1: protected Base
{
  int k;
 public:
  void Set_k(){Set_ij(10,12);k=i*j;}
  void ShowAll() {cout<<k<<" ";Show_ij();}
};
void main()
{
 Derived1 ob;
 ob.Set_ij(2,3);// illegal; Set_ij() is a protected member of
Derived1
 ob.Set_k();    // OK, public member of Derived1
 ob.ShowAll();  // OK, public member of Derived1
 ob.Show_ij();  // illegal; Show_ij() is a protected member of
Derived1
}
```

Example

```
class C
{
private:
        C(){} // private constructor
protected:
        C(int a){}
public:
        C(int a, int b)
        {}
};

void main ()
{
 C c; // no access
 C c1(1); // no access
 C c2(1, 2); //  Ok
 C* pc = new C; // no access to private constructor
 C* pc1 = new C(1); // no access to protected constructor
 C* pc2 = new C(1,2); // Ok : public constructor
}
```

Example

```
class C
{
public:
 void publicFC()
 {
 }
protected:
 void protectedFC()
 {
 }
```

```
private:
 void privateFC()
 {
 }
};

class D: public C
{
public:
 void privateFC() // overrided method from base class
 {
 }
};

void main()
{
 C c;
 c.publicFC(); // Ok
 c.protectedFC(); // no access to protected method
 c.privateFC(); // no access to private method
 D d;
 d.C::publicFC(); // Ok
 d.C::protectedFC(); // no access to protected method from base
class
 d.C::privateFC(); // no access to private method from base class
 d.privateFC(); // Ok: call to overrided method from class D
}
```

public

Syntax
public: [*member-list*]
public *base-class*

When preceding a list of class members, the **public** keyword specifies that those members are accessible from any function. This applies to all members declared up to the next access specifier or the end of the class.

When preceding the name of a base class, the **public** keyword specifies that the public and protected members of the base class are public and protected members, respectively, of the derived class.

Default access of members in a class is private. Default access of members in a structure or union is public.

Default access of a base class is private for classes and public for structures. Unions cannot have base classes.

Example

```
class BaseClass
{
public:
   int pubFunc();
```

```
};

class DerivedClass : public BaseClass
{
};
void main()
{
   BaseClass aBase;
   DerivedClass aDerived;
   aBase.pubFunc();       // pubFunc() is accessible from any
function
   aDerived.pubFunc();    // pubFunc() is still public in derived
class
}
```

Example

```
class C
{
private:
        C(){} // private constructor
protected:
        C(int a){}
public:
        C(int a, int b)
        {}
};

void main ()
{
 C c; // no access
 C c1(1); // no access
 C c2(1, 2); //  Ok
 C* pc = new C; // no access to private constructor
 C* pc1 = new C(1); // no access to protected constructor
 C* pc2 = new C(1,2); // Ok : public constructor
}
```

Example

```
class Base
{
 int i,j;
public:
 void Set_ij(int a, int b) {i=a;j=b;}
 void Show() {cout<<i<<" "<<j<<"\n";}
};
class Derived: public Base
{
 int k;
public:
 Derived(int x) {k=x;}
 void Show_k() {cout<<k<<"\n";}
};
void main()
{
```

```
 Derived ob(3);
 ob.Set_ij(1,2);//access to the base member
 ob.Show();     //access to the base member
 ob.Show_k();   //uses the derived class member
}
```

register

The **register** keyword specifies that the variable is to be stored in a machine register, if possible.
The **register** specifier like the **auto** specifier can be applied only to names of objects declared in a block or to function parameters. They specify that the named object has automatic storage duration. An object declared without a *storage-class-specifier* at block scope or declared as a function parameter has automatic storage duration by default.
The register specifier can be applied only to names of objects declared in a block (local variables) or to function parameters. The **register** variables are ideal for loop control; it is recommended to use a **register** variable in places where that variable is referred many times. Usually there can be kept at least two **register** variables of type char or int in the CPU registers.

Example

```
int  powerint(register int m, register int e)
{
      register int temp;      //m, e, temp are declared as register
                              //variables
      temp=1;                 //for speed optimization.
      for (;e;e--) temp*=m;
      return temp;
}
```

Example

```
void back_display(char *s);
void                                                        main()
{
 back_display("I like C++");
}
void back_display(char *s)
{ register int t;
  for (t=strlen(s)-1;t>0;t--) putchar(s[t]);
}
```

return

Syntax
return [*expression*];
The **return** statement allows a function to immediately transfer control back to the calling function (or, in the case of the main function, transfer control back to the

operating system). The **return** statement accepts an expression, which is the value passed back to the calling function. Functions of type **void**, constructors, and destructors cannot specify expressions in the **return** statement; functions of all other types must specify an expression in the **return** statement.
When the flow of control exits the block enclosing the function definition, the result is the same as it would be if a **return** statement with no expression had been executed. This is illegal for functions that are declared as returning a value.
A function can have any number of **return** statements.

Example

```
double fMultip()
{
   double a;
   a = 4.87 * 3;
   return (a);
}
```

Example

```
void draw( int I, long L );
long sq( int s );
int main()
{
    long y;
    int x;
    y = sq( x );
    draw( x, y );
}

long sq( int s )
{
    return( s * s );
}

void draw( int i, long L )
{
    /* Statements defining the draw function here */
    return;
}
```

In this example, the `main` function calls two functions: `sq` and `draw`. The `sq` function returns the value of `x * x` to `main`, where the return value is assigned to `y`. The `draw` function is declared as a **void** function and does not return a value.
The following function returns a pointer to the first occurrence of character `c` in the string `s`.

Example

```
char *find(char c,char *s)
{
 while(c!=*s && *s) s++;
```

```
 return (s);
}
```
When an object is returned by a function, a temporary object is created to store the return value. After the value is returned the object is destroyed.

Example

```
class myclass
{
 int i;
public:
 void Set_i(int n) {i=n;}
 int Get_i(return i;}
};
myclass f();
void main()
{
 myclass o;
 o=f();
 cout<<o.Get_i()<<endl;
}
myclass f()
{
 myclass x;
 x.Set_i(1);
 return x;
}
```

short

The **short** keyword is a conversion specifier. For more information about conversion specifiers see **signed.**
The **short** keyword designates a 16-bit integer. The **short** keyword can be preceded by either the keyword **signed** or the keyword **unsigned**. The **int** keyword is optional and can be omitted.

```
[ signed | unsigned ] short [ int ] declarator-list;
```

Example

```
short int i;
short     i;//same as "short int i;"
```

signed

The **signed** keyword is a conversion specifier.
All fundamental types excepted **void** can be preceded by several conversion specifiers. A conversion specifier is used for the fundamental type modifying in order to adapt to many different cases. Here is the list of conversion specifiers: **signed, unsigned, long, short.**

The **signed** keyword indicates that the most significant bit of an integer variable represents a sign bit rather than a data bit. This keyword is optional and can be used with any of the character and integer types.

```
[ signed ] type-qualifier[ int ]identifier-name;
```

Example

```
signed int  i; //signed is default;
signed      i; //same as "signed int i;"
signed char ch;//unsigned is default for char
```

sizeof

Syntax
sizeof *expression*

The **sizeof** keyword gives the amount of storage, in bytes, associated with a variable or a type (including aggregate types). This keyword returns a value of type **size_t**. The *expression* is either an identifier or a type-cast expression (a type specifier enclosed in parentheses).

When applied to a structure type or variable, **sizeof** returns the actual size, which may include padding bytes inserted for alignment. When applied to a statically dimensioned array, **sizeof** returns the size of the entire array. The **sizeof** operator cannot return the size of dynamically allocated arrays or external arrays.

Examples

```
size_t  i = sizeof( int );

struct s{
    char c;
    int i;
};
size_t size = sizeof(s);

int  array[] = { 1, 2, 3, 4, 5 };     // sizeof( array ) is 20
                                      // sizeof( array[0] ) is 4
size_t  sizearr = sizeof( array )/sizeof( array[0] );
                                      // Count of items in array
union u
{
 char ch;
 int i;
 float f;
 char sir[10];
};
size_t size_u=sizeof(sir);    //sizeof(u)=size   of   the   biggest
                              element //of the union; in this case
                              sir occupies //the largest amount of
                              memory
```

static

The keyword static can be used to declare a local variable with static storage duration.
(the variable is allocated when the program begins and deallocated when the program ends) and initializes it to 0 unless another value is specified.
When modifying a variable or function at file scope, the **static** keyword specifies that the variable or function has internal linkage (its name is not visible from outside the file in which it is declared).
The keyword static applied to a class data member in a class definition gives the data member static storage duration.
EXAMPLE

```
static int i;         // Variable accessible only from this file

static void func();   // Function accessible only from this file
```

EXAMPLE

```
int max_so_far( int crt )
{
   static int maxim;    // Variable whose value is retained
                        // between each function call
   if( crt > maxim )
      maxim = crt;
   return maxim;
}
```

Example

```
class Account
{
public:
   static void setInterest( float newValue )  // Member function
      { currentRate = newValue; }             // that accesses
                                              // only static
                                              // members
private:
   char name[30];
   float total;
   static float currentRate;    // One copy of this member is
                                // shared among all instances
                                // of Account
};

// Static data members must be initialized at file scope, even
// if private.
float Account::currentRate = 0.3;
```

When you specify a base class as **private**, it affects only nonstatic members. Public static members are still accessible in the derived classes. However, accessing members of the base class using pointers, references, or objects can require a conversion, at which time access control is again applied.

Example

```
class Base
{
public:
    int Print();             // Nonstatic member.
    static int CountOf();    // Static member.
};
// Derived1 declares Base as a private base class.
class Derived1 : private Base
{
};
// Derived2 declares Derived1 as a public base class.
class Derived2 : public Derived1
{
    int ShowCount();    // Nonstatic member.
};
// Define ShowCount function for Derived2.
int Derived2::ShowCount()
{
   // Call static member function CountOf explicitly.
    int cCount = Base::CountOf();     // OK.

   // Call static member function CountOf using pointer.
    cCount = this->CountOf();  // Error. Conversion of
                               //  Derived2 * to Base * not
                               //  permitted.
    return cCount;
}
```

struct

Syntax

struct [*tag*] **{** *member-list* **}** [*declarators*];
[**struct**] *tag declarators*;

The **struct** keyword defines a structure type and/or a variable of a structure type. A structure type is a user-defined composite type. It is composed of "fields" or "members" that can have different types.

A structure is the same as a class except that its members are public by default.

Structure variables can be initialized. The initialization for each variable must be enclosed in braces.

Example

```
struct PERSON              // Declare PERSON struct type
{
   int   age;              // Declare member types
```

```
   long  ss;
   float weight;
   char  name[25];
} family_member;           // Define object of type PERSON

   struct PERSON sister;
   PERSON brother;

   sister.age  = 13;       // assign values to members
   brother.age = 7;
```

Example

```
struct POINT               // Declare POINT structure
{ int x;                   // Define members x and y
  int y;
} spot = { 20, 40 };       // Variable spot has // values x=20,
y=40
```

Example

```
struct CELL             // Declare CELL bit field
{
   unsigned character  : 8;  // 00000000 ????????
   unsigned foreground : 3;  // 00000??? 00000000
   unsigned intensity  : 1;  // 0000?000 00000000
   unsigned background : 3;  // 0???0000 00000000
   unsigned blink      : 1;  // ?0000000 00000000
} screen[25][80];            // Array of bit fields
```

Example

```
#define DELAY 128000
struct my_hour
{
 int hour,min,sec;
};
void delay()
{long int t;
 for(t=0;t<DELAY;++t);
}
void match(my_hour *t)
{
 t->sec++;
 if(t->sec==60)
 {t->sec=0;t->min++};
 if(t->min==60)
 {t->min=0;t->hour++};
 if(t->hour==24)
  t->hour=0;
 delay();
}
void show(my_hour *t)
```

```
{
 print("%02d:",t->hour);
 print("%02d:",t->min);
 print("%02d:",t->sec);
}
void main()
{
 my_hour systhour;
 systhour.hour= systhour.min= systhour.sec=0;
 for(;;)
 {
  match(&systhour);
  show(&systhour);
 }
}
```

Example

```
struct phone_address
{
 char street[40];
 char city[20];
 unsigned long int zip_code;
 char phonenr[10];
};
struct employee
{
 char name[20];
 phone_address pa;
 float salary;
};
void main()
{
 employee worker;
 worker.pa.zip_code=93546;
}
```

switch

The C++ **switch** statement allows selection among multiple sections of code, depending on the value of an expression.
The expression enclosed in parentheses, the "controlling expression" shall be of integral type, enumeration type, or of a class type for which a single conversion function to integral or enumeration type exists. If the condition is of class type, the condition is converted by calling that conversion function, and the result of the conversion is used in place of the original condition for the remainder of this section. Integral promotions are performed. Any statement within the `switch` statement can be labeled with one or more case labels as follows:
`case` *constant-expression* **:** *statement*

where the *constant-expression* shall be an integral *constant-expressio*n. The integral constant-expression is implicitly converted to the promoted type of the switch condition. No two of the case constants in the same switch shall have the same value after conversion to the promoted type of the switch condition.

There shall be at most one label of the form

`default :` *statement*

within a **switch** statement.

When the **switch** statement is executed, its condition is evaluated and compared with each case constant.

If one of the case constants is equal to the value of the condition, control is passed to the statement following the matched case label. If no case constant matches the condition and if there is a **default** label control passes to the statement labeled by the default label. If no case matches and if there is no **default** then none of the statements in the switch is executed.

An inner block of a **switch** statement can contain definitions with initializations as long as they are reachable - that is, not bypassed by all possible execution paths. Names introduced using these declarations have local scope. The following code fragment shows how the **switch** statement works:

Example

```
switch( tolower( *argv[1] ) )
{
    // Error. Unreachable declaration.
    char szChEntered[] = "Character entered was: ";
case 'a' :
    {
    // Declaration of szChEntered OK. Local scope.
    char szChEntered[] = "Character entered was: ";
    cout << szChEntered << "a\n";
    }
    break;
case 'b' :
    // Value of szChEntered undefined.
    cout << szChEntered << "b\n";
    break;
default:
    // Value of szChEntered undefined.
    cout << szChEntered << "neither a nor b\n";
    break;
}
```

Switch statements can be nested; in such cases, **case** or **default** labels associate with the most deeply nested **switch** statements that enclose them.

Example

```
switch( x )
{
case 1 :
        switch (y)
        {
         case 0: printf("Division by 0, error!");
```

```
                break;
         case 1: processed(x,y);
        }
        break;
case 2 :
.
.
.
}
```

`case` and `default` labels in themselves do not alter the flow of control, which continues unimpeded across such labels. To stop execution at the end of a part of the compound statement, insert a **break** statement. This transfers control to the statement after the **switch** statement. This example demonstrates how control "drops through" unless a **break** statement is used:

Example

```
BOOL fClosing = FALSE;

...

switch( wParam )
{
case IDM_F_CLOSE:       // File close command.
    fClosing = TRUE;
    // fall through
case IDM_F_SAVE:        // File save command.
    if( document->IsDirty() )
        if( document->Name() == "UNTITLED" )
            FileSaveAs( document );
        else
            FileSave( document );
    if( fClosing )
        document->Close();
    break;
}
```

The preceding code shows how to take advantage of the fact that **case** labels do not impede the flow of control. If the **`switch`** statement transfers control to `IDM_F_SAVE`, `fClosing` is `FALSE`. Therefore, after the file is saved, the document is not closed. However, if the **switch** statement transfers control to `IDM_F_CLOSE`, `fClosing` is set to `TRUE`, and the code to save a file is executed. As we've seen in the preceding example, the **break** statement is optional in a **switch** statement; **break** terminates the block of statement associated with a constant. If the **break** is omitted the execution is going on with the statements from the next **case** until a **break** is encountered or until the end of the **switch.**

Example

```
switch( c )
{
    case 'A':
        capa++;
```

```
    case 'a':
        lettera++;
    default :
        total++;
}
```

All three statements of the **switch** body in this example are executed if `c` is equal to `'A'` since a **break** statement does not appear before the following case. Execution control is transferred to the first statement `(capa++;)` and continues in order through the rest of the body. If `c` is equal to `'a'`, `lettera` and `total` are incremented. Only `total` is incremented if `c` is not equal to `'A'` or `'a'`.
A single statement can carry multiple **case** labels, as the following example shows:

Example

```
case 'a' :
case 'b' :
case 'c' :
case 'd' :
case 'e' :
case 'f' :  hexcvt(c);
```

In this example, if *constant-expression* equals any letter between `'a'` and `'f'`, the `hexcvt` function is called.

this

The **this** pointer is a pointer accessible only within the member functions of a **class**, **struct**, or **union** type. It points to the object for which the member function is called. Static member functions do not have a **this** pointer.
When a nonstatic member function is called for an object, the address of the object is passed as a hidden argument to the function. For example, the following function call

```
myDate.setMonth( 3 );
```

can be interpreted this way:

```
setMonth( &myDate, 3 );
```

The object's address is available from within the member function as the **this** pointer. It is legal, though unnecessary, to use the **this** pointer when referring to members of the class.
The expression **(*this)** is commonly used to return the current object from a member function.

Example

```
class Date
{
  int month, day, year;
public:
  void setMonth(int mn);
  void setDay(int dy);
```

```
  void setYear(int yr);
  int getMonth() {return month;}
  int getDay() {return day;}
  int getYear() {return year;}
};

void Date::setMonth( int mn )
{
   month = mn;             // These three statements
   this->month = mn;       //     are equivalent
   (*this).month = mn;
}
```

In the following example the overloaded operator **()** returns the object that called it (the **()** operator).

Example

```
class place
{
 int longitude,latitude;
public:
 place() {}
 place(int lg,int lt)
 {
  longitude=lg;
  latitude=lt;
 }
 void show()
 {
  cout<<longitude<<" "<<latitude<<endl;
 }
 place operator+(place op2);
 place operator ()(int i,int j);
};
place place::operator+(place op2)
{
 place temp;
 place.longitude=op2.longitude+ longitude;
 place.latitude=op2. Latitude+ latitude;
 return temp;
}
place place::operator()(int i,int j)
{
 longitude=i;
 latitude=j;
 return *this;
}
```

Example for **->** operator

```
class myclass
{
 public:
 int i;
 myclass operator->() {return this;}
};
```

```
void main()
{
 myclass ob;
 ob->i;//it's the same with ob.i
}
```

true

This keyword is one of the two values that a variable of type **bool** can have. If `i` is of type **bool**, then the statement `i=true;` assigns **true** to `i`. See the **bool** keyword.

typedef

typedef *type-declaration synonym*;
The **typedef** keyword defines a synonym for the specified *type-declaration*. The identifier in the *type-declaration* becomes another name for the type, instead of naming an instance of the type. You cannot use the **typedef** specifier inside a function definition.
A **typedef** declaration introduces a name that, within its scope, becomes a synonym for the type given by the *decl-specifiers* portion of the declaration. **typedef** declarations do not introduce new types , they introduce new names for existing types.

Example

```
typedef unsigned long ulong;

ulong ul;     // Equivalent to "unsigned long ul;"

typedef struct mystructtag
{
   int   i;
   float f;
   char  c;
} mystruct;

mystruct ms;   // Equivalent to "struct mystructtag ms;"
```

In a given scope, a typedef specifier can be used to redefine the name of any type declared in that scope to refer to the type to which it already refers.
Example:

```
typedef struct s { /* ... */ } s;
typedef int I;
typedef int I;
typedef I I;
```

In a given scope, a `typedef` specifier shall not be used to redefine the name of any type declared in that
scope to refer to a different type.

Example:

```
class complex { /* ... */ };
typedef int complex; // error: redefinition
```

Similarly, in a given scope, a class or enumeration shall not be declared with the same
name as a *typedef-name* that is declared in that scope and refers to a type other than the class or enumeration itself.

Example:

```
typedef int complex;
class complex { /* ... */ }; // error: redefinition
```

If the typedef declaration defines an unnamed class (or enum), the first *typedef-name* declared by the declaration to be that class type (or enum type) is used to denote the class type (or enum type) for linkage purposes only.

Example:

```
typedef struct { } *ps, S; // S is the class name for linkage
purposes
```

If the *typedef-name* is used where a *class-name* (or *enum-nam*e) is required, the program is ill-formed. For example:

```
typedef struct
{
  S();            //error: requires a return type because S is
                  // an ordinary member function, not a constructor
} S;
```

union

Syntax

union [*tag*] **{** *member-list* **}** [*declarators*];
[**union**] *tag declarators*;

The **union** keyword declares a union type and/or a variable of a union type.
A union is a user-defined data type that can hold values of different types at different times. It is similar to a structure except that all of its members start at the same location in memory. A union variable can contain only one of its members at a time. The size of the union is at least the size of the largest member.

A union is a limited form of the class type. It can contain access specifiers (public, protected, private), member data, and member functions, including constructors and destructors. It cannot contain virtual functions or static data members. It cannot be used as a base class, nor can it have base classes. Default access of members in a union is public.

Example

```
union UNKNOWN           // Declare union type
{ char ch;
int i;
long l;
float f;
double d;
}var1;                  // Optional declaration of union variable

union UNKNOWN var1;
UNKNOWN var2;
```

A variable of a union type can hold one value of any type declared in the union. Use the member-selection operator (**.**) to access a member of a union:

```
var1.i = 6;           // Use variable as integer
var2.d = 5.327;       // Use variable as double
```

A union of the form **`union { member-list };`** is called an anonymous union; it defines an unnamed object of unnamed type. A global anonymous union must be static. A local anonymous union must be either static or automatic, not external. The *member-list* of an anonymous union shall only define non-static data members (nested types and functions cannot be declared within an anonymous union). The names of the members of an anonymous union shall be distinct from the names of any other entity in the scope in which the anonymous union is declared. For the purpose of name lookup, after the anonymous union definition, the members of the anonymous union are considered to have been defined in the scope in which the anonymous union is declared.

Example*:*

```
void f()
{
union { int a; char* p; };
a = 1;
// ...
p = "Jennifer";
// ...
}
```

Here `a` and `p` are used like ordinary (nonmember) variables, but since they are union members they have the same address.

Anonymous unions declared in a named namespace or in the global namespace shall be declared static.
Anonymous unions declared at block scope shall be declared with any storage class allowed for a block-scope variable, or with no storage class. A storage class is not allowed in a declaration of an anonymous union in a class scope. An anonymous union shall not have private or protected members. An anonymous union shall not have function members.
A union for which objects or pointers are declared is not an anonymous union.
Example:

```
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1;      // error
ptr->aa = 1; // OK
```

The assignment to plain `aa` is ill formed since the member name is not visible outside the union, and even if it were visible, it is not associated with any particular object.

unsigned

The **unsigned** keyword is a conversion specifier. For more information about conversion specifiers see **signed.**
The **unsigned** keyword indicates that the most significant bit of an integer variable represents a data bit rather than a signed bit.
This keyword is optional and can be used with any of the character and integer types.

```
[ unsigned ] type-qualifier[ int ]identifier-name;
```

Example

```
void f(unsigned Pc);     // void f(unsigned int)
void g(unsigned int Pc); // void g(unsigned int)
unsigned int      i;
unsigned          i;//same as "unsigned int i;"
unsigned long int l;//int OK, not needed
unsigned char    ch;//unsigned is default for char
```

virtual

The **virtual** keyword declares a virtual function.

Syntax

virtual *member-function-declarator*

member-function-declarator
Declares a member function.

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Example

```
class BaseClass
{
public:
  virtual void vfunc() {cout<<"This is vfunc() from
BaseClass"<<endl;}
};
class Derived1: public BaseClass
{
public:
  void vfunc(){cout<<"This is vfunc() from Derived1"<<endl;}
};
class Derived2: public BaseClass
{
public:
  void vfunc(){cout<<"This is vfunc() from Derived2"<<endl;}
};
```

You can execute different versions of `vfunc( )` depending on the type of object you're calling it for.

```
void main()
{
  BaseClass *p,b;
  Derived1 d1;
  Derived2 d2;
  P=&b;           //points to BaseClass
  p->vfunc();     //call BaseClass:: vfunc()
  p=&d1;          //points to Derived1
  p->vfunc();     //call Derived1:: vfunc()
  p=&d2;          //points to Derived2
  p->vfunc();     //call Derived2:: vfunc()
}
```

The **virtual** keyword is needed only in the base class's declaration of the function; any subsequent declarations in derived classes are virtual by default.

A derived class's version of a virtual function must have the same parameter list and return type as those of the base class. If these are different, the function is not considered a redefinition of the virtual function. A redefined virtual function cannot differ from the original only by return type.

A virtual member function does not have to be visible to be overridden, for example:

```
struct B {
virtual void f();
};
struct D : B {
void f(int);
};
```

```
struct D2 : D {
void f();
};
```
the function `f(int)` in class `D` hides the virtual function `f()` in its base class B; `D::f(int)` is not a virtual function. However, `f()` declared in class `D2` has the same name and the same parameter list as `B::f()`, and therefore is a virtual function that overrides the function `B::f()` even though `B::f()` is not visible in class `D2`.
Even though destructors are not inherited, a destructor in a derived class overrides a base class destructor declared virtual. The `virtual` specifier implies membership, so a virtual function cannot be a nonmember function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function declared in a class shall be defined, or declared pure in that class, or both, but no diagnostic is required. A class with alt least one pure virtual function is an abstract class.
The access rules for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it.
Example:
```
class B {
public:
virtual int f();
};
class D : public B {
private:
int f();
};
void f()
{
D d;
B* pb = &d;
D* pd = &d;
pb->f(); //OK: B::f() is public,
// D::f() is invoked
pd->f(); //error: D::f() is private
}
```
Access is checked at the call point using the type of the expression used to denote the object for which the member function is called `(B*` in the example above). The access of the member function in the class in which it was defined `(D` in the example above) is in general not known.

void

When used as a function return type, the **void** keyword specifies that the function does not return a value. When used for a function's parameter list, void specifies that

the function takes no parameters. When used in the declaration of a pointer, void specifies that the pointer is "universal".
If a pointer's type is **void ***, the pointer can point to any variable that is not declared with the **const** or **volatile** keyword. A void pointer cannot be dereferenced unless it is cast to another type. A void pointer can be converted into any other type of data pointer.
A void pointer can point to a function, but not to a class member in C++.
You cannot declare a variable of type void.

Example

```
void vobject;              // Error
void *pv;                  // OK
int  *pint; int i;
void main()                // main has no return value
{
pv = &i;
pint = (int *)pv;          // Cast optional in C
                           // required in C++
}
```

volatile

The **volatile** keyword is a type qualifier used to declare that an object can be modified in the program by something other than statements, such as the operating system, the hardware, or a concurrently executing thread.
The following example declares a volatile integer `nVint` whose value can be modified by external processes:

```
int volatile nVint;
```

Objects declared as **volatile** are not used in optimizations because their value can change at any time. The system always reads the current value of a volatile object at the point it is requested, even if the previous instruction asked for a value from the same object. Also, the value of the object is written immediately on assignment.
One use of the **volatile** qualifier is to provide access to memory locations used by asynchronous processes such as interrupt handlers.
The two type qualifiers, **const** and **volatile**, can appear only once in a declaration. Type qualifiers can appear with any type specifier; however, they cannot appear after the first comma in a multiple item declaration. For example, the following declarations are legal:

```
typedef volatile int VI;
const int ci;
```

These declarations are not legal:

```
typedef int *i, volatile *vi;
```

```
float f, const cf;
```
The **const** and **volatile** specifiers can be used together. For example, we assume that 0x30 is a port value that is modified only by the external conditions; the following declaration will prevent any occurrence of secondary accidentally effects:
```
const volatile unsigned char *port=0x30;
```

while

The **while** keyword is used either with **do** keyword representing the **do-while** statement, either alone representing the **while** statement.
The **while** statement executes a *statement* repeatedly until the termination condition (the *expression*) specified evaluates to zero. The test of the termination condition takes place before each execution of the loop; therefore, a **while** loop executes zero or more times, depending on the value of the termination expression.

Syntax

while(*expression* **)**
statement;

The following code (a keyboard input function) uses a **while** loop to wait for the user to press 'A':
```
char wait_char()
{
  char ch='\0';
  while (ch!='A')
    ch=getchar();
  return ch;
}
```
The following example copies characters from `string2` to `string1`. If `i` is greater than or equal to 0, `string2[i]` is assigned to `string1[i]` and `i` is decremented. When `i` reaches or falls below 0, execution of the **while** statement terminates.

Example
```
while ( i >= 0 )
{
    string1[i] = string2[i];
    i--;
}
```
The following function uses the **do** statement to wait for the user to press a specific key:
```
void WaitKey( char ASCIICode )
{
    char chTemp;

    do
    {
        chTemp = _getch();
    }
    while( chTemp != ASCIICode );
```

```
}
```

A **do** loop rather than a **while** loop is used in the preceding code - with the **do** loop, the **_getch** function is called to get a keystroke before the termination condition is evaluated. This function can be written using a **while** loop, but not as concisely:

```
void WaitKey( char ASCIICode )
{
    char chTemp;

    chTemp = _getch();

    while( chTemp != ASCIICode )
    {
        chTemp = _getch();
    }
}
```

The *expression* must be of an integral type, a pointer type, or a class type with an unambiguous conversion to an integral or pointer type.

Support for RTX51 Tiny real-time multitasking operating system

When ***-ECKRTX*** command line directive is used, Ceibo C++ defines the ***main*** function of EC++ program as:

void _main_task(void) _task_ 0

Task 0 (main task) is reserved by Ceibo C++, such as the user can use only the tasks with id > 0.

To do this type "***-ECKRTX"*** under Project - Options for Target - C51 - Misc Controls.

Limitations when using 8051 extensions

Virtual functions suppose an indirect call through a pointers table. Keil C compiler also has some restrictions regarding the number and the type of arguments for functions which are called using pointers to them. Therefore, for these back-ends the virtual functions must be declared with ***reentrant*** keyword.

CHAPTER 5

ERRORS AND WARNINGS

CHAPTER 5

ERRORS AND WARNINGS

CHAPTER 5

ERRORS AND WARNINGS

Warnings

T13: "PROTECTION PLUG IS MISSING OR SERIAL NUMBER IS INVALID"

T100: "Pragma directive '%s' was discarded: must have parameter between %i
and %i";

The compiler ignored a pragma because its parameter was out of range.
These are examples of this warning:

```
#pragma FF(9)      // error - parameter must be in the range 0..7
#pragma PW(21)     // error - parameter must be in the range 78..132
```

T101: "Pragma directive '%s' was discarded : must have first parameter between %i and %i";

The compiler ignored a pragma because its first parameter was out of range.
This is an example of this warning:

```
#pragma OT( 89, sIZe)    // error - the first parameter must be in
                         the
                            range 0..9
```

T102: "Pragma directive '%s'is unknown or not well-formed: Will be discarded";

The compiler did not recognize a pragma and ignored it.
Make sure that the pragma is allowed by the type of compiler being used or the pragma is well formed.

T103: "Couldn't find a destructor for object of type: '%s'";

The compiler couldn't find a destructor for the specified object.

T104: "'%s' is not a pointer-to-object type. Destructor will not be called";

You tried to delete a void pointer.
The following is an example of this warning:

```
void main()
{
      void *p;
      delete p; // warning
}
```

T105: "Embedded C++ does not accept virtual inheritance: 'virtual' not considered";

A base class was specified as a virtual base. Since embedded C++ does not accept virtual inheritance, **virtual** is ignored.

T106: "Embedded C++ does not support multiple inheritance: only the first base class considered";

The given derived class has two or more direct base classes. Since embedded C++ does not support multiple inheritance, the compiler assumes that only the first base class is valid.

T107: "function '%s' should return a value. Value '0' will be returned";

The **main** function with a return type of **int** didn't return a value. The compiler assumed that the returned value was 0.

T108: "Type mismatch in function '%s' redeclaration; the first declaration used";

You redeclared the function with a different header. This warning is caused by having one of the following specifiers: **interrupt, register bank, priority, systemuser, saveregbank, saveregs, memorymodel** only once (in one declaration of the function)
The following is an example of this error:

```
void f();
void f() interrupt(9);  // warning
void main()
{
}
```

Errors

T200: "'#pragma C' directive cannot be used out of global scope";

The **#pragma C** directive that must be specified at a global level (that is, outside a function body) occurred within a function.
The following is an example of this error:

```
void main()
{
  #pragma C   // error
}
```

T201: "Name '%s' is undefined";

The specified name was not declared.
A variable's type must be specified in a declaration before it can be used. The parameters that a function uses must be specified in a declaration, or protoype, before the function can be used.

Tips
- Make sure any include file containing the required declaration is not omitted.
- Make sure the identifier name is spelled correctly.
- Make sure the identifier is using the correct upper- and lowercase letters.

T202: "%s '%s' is inaccessible";

The specified private or protected member of a class, structure, or union was accessed.

Tips
This error can be caused by attempting to access a member that is declared as private or protected, or by accessing a public member of a base class that is

inherited with private or protected access. The member should be accessed through a member function with public access or should be declared with public access.

T203: "'%s' cannot instantiate abstract class due to following virtual pure functions:";

An object of the specified abstract class or structure was declared.
A class (or structure) with one or more pure virtual functions cannot be instantiated. Each pure virtual function must be overridden in a derived class before objects of the derived class can be instantiated.

T204: "'%s': array initialization needs curly braces";

The given array requires values enclosed in curly braces for initialization.
The following is an example of this error:

```
void main()
{
   int s[3]=(1,3,5);//error
   int v[3]={1,3,5};//OK
}
```

T205: "'%s' is already defined";

The given identifier was defined more than once, or a subsequent declaration differed from a previous one.
The following are examples of this error:

```
int a;
char a;
void main()
{
}
void main()
{
  int a;
  int a;
}
void main()
{
  int a;
  float a;
}
```

T206: "Function '%s' is already defined";

The specified member function was defined or declared earlier.
This error can be caused by repeating the same formal parameter list in more than one function definition or declaration.

T207: "Function '%s' already has a body";

The function has already been defined.
The following is an example of this error:

```
class C {
 void f() {}
 void f() {} // error
};
```

T208: "cannot convert parameters types or the call is ambiguous for function '%s'";

The specified parameter of the specified function could not be converted to the required type or the specified overloaded function call could not be resolved.

Tips
In case of a conversion type error recheck the prototype for the given function and correct the argument noted in the error message. If necessary, an explicit conversion may need to be supplied.
The following are examples of this error:

```
#include<iostream.h>
class A {} a;
func( int, A );
int g(int i);
int g(int i, int j=1); void main()
{
   func( 1, 1 );        // error, no conversion from int to A
   cout<<g(3,5)<<endl;  // not ambiguous call
   cout<<g(10)<<endl;   // ambiguous call
}

int g(int i)
{return i;}
int g(int i, int j)
{return i*j;}
```

T209: "Class name '%s' cannot be redefined inside class";

The specified class, structure, or union was already defined.
A class, structure, or union can be defined only once.
The following is an example of this error:

```
class C
{
  class C {
  };          // error, class C is already defined
};
```

T210 :"'%s': return type or storage class specified for constructor/destructor is illegal";

A constructor/destructor in the specified class, structure, or union was declared with a **return** type or a storage class was illegally specified for the constructor/destructor.
A constructor/destructor cannot be declared with a **return** type.
inline is the only legal storage class for constructors/destructors.
The following are examples of this error:

```
class C
{
   int ~C();   // error, returns int
   void ~C();  // error, returns void
   extern C(); // error, only 'inline' can be applied
   ~C();       // OK
};
```

T211 :"'%s': storage 'static' for constructor/destructor is illegal";

The specified constructor/destructor was declared as **static**.
Constructors/Destructors cannot be declared as **static**.
The following is an example of this error:

```
class C
{
  static C(); //error
};
```

T212 :"'extern' storage-class specifier illegal on members";

A storage class was illegally specified for the given identifier.
The following is an example of this error:

```
class C
```

```
{
  extern C();     //error
  extern int i;   //error
};
```

T213 :"Invalid constructor definition";

The specified constructor was declared with a return type or the first parameter in the specified constructor was the same type as the class, structure, or union for which it was defined.
A constructor cannot be declared with a **return** type and the first parameter cannot be of the same type as the class/struct for wich it was defined.

T214 :"The constructor definition '%s' is not allowed.";

The first parameter in the specified copy constructor was the same type as the class, structure, or union for which it was defined.
A copy constructor for a type can take a reference to the type as the first parameter but not the type itself.
The following is an example of this error:

```
class A
{
   A( A );     // error, takes an A
};
class B
{
   B( B& );    // OK, reference to B
};
```

T215 :"Invalid destructor definition";

The specified destructor had a nonvoid formal parameter list or was declared with a return type.
A destructor can take only a **void** parameter and cannot be declared with a **return** type.

The following is an example of this error:

```
class C
{
public:
        int i;
        C() {i=0;}
        ~C(int a); // error
};
```

T216 :"Destructors cannot have arguments";

The specified destructor had a nonvoid formal parameter list.
A destructor can take only a void parameter. Other parameter types are not allowed.
The following is an example of this error:

```
class D
{
  ~D(int i);//error
  ~D();     //OK
};
```

T217 :"'%s': overriding virtual function differs from '%s' only by return type or calling convention";

The specified virtual function and a derived overriding function had identical parameter lists but different return types.
An overriding function in a derived class cannot be redefined to differ only by its return type from a virtual function in a base class.
A function in a derived class or structure overrides a virtual function in a base class only if their names, parameters, and return values are all identical. (If the functions have different parameters, the compiler treats them as different functions and does not override the virtual function.)
It may be necessary to cast the return value after the virtual function has been called.
The following is an example of this error:

```
class C
{
public:
      virtual int f();
};
class D : public C
{
public:
      void f();
};
```

T218 :"'%s': overloaded function differs from '%s' only by return type";

The indicated overloaded functions had different return types but the same parameter list.
Each overloaded function must have a distinctly different formal parameter list.
The following is an example of this error:

```
double g(int i);
```

```
void g(int i);
void main()
{
}
```

T219 :"Multiple initializations given for argument '%d' of function '%s'";

The specified default parameter in a member function was redefined.
A default parameter cannot be redefined. If another value for the parameter is required, then the default parameter should be left undefined.
The following is an example of this error:

```
void g(int i, int j=2);
void main()
{
}

void g(int i, int j=2)
{
}
```

T220 :"'%s' : missing default parameter for parameter '%d'";

A parameter was missing in a default parameter list.
If a default parameter is supplied anywhere in a parameter list, then all subsequent parameters on the right side of the default parameter must also be defined.
The following is an example of this error:

```
void func( int = 1, int, int = 3);  // error
void func( int, int, int = 3);      // OK
void func( int, int = 2, int = 3);  // OK
```

T221 :"Storage classes do not match for function '%s'";

A function was declared both extern (global) and static at a given scope.
The following is an example of this error:

```
char* g();        // g() has external linkage
static char* g()  // error: inconsistent linkage
{ /* ... */ }
```

T222 :"%s pure specifier only applies to virtual function - specifier ignored";

The specified nonvirtual function was specified as pure **virtual**.

The specifier was ignored.
The following is an example of this error:

```
class A
{
public:
  void f1() = 0;          // error, not virtual
  virtual void f2() = 0;  // OK
};
```

T223 :"Attempt to grant or reduce access to '%s'";

A derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class.
It can't add or reduce access rights.
The following is an example of this error:

```
class C
{
      int i;
public:
      int j,k;
      int f();
};
class D : private C
{
public:
      C::i; // error - attempt to grant access to C::i
      C::j; // OK - restores the public access for C::j

}
```

T224 :"Access can be only changed to public or protected";

A derived class can modify the access rights of a base class member, but only to public or protected.
A base class member can't be made private.

```
class C
{
      int i;
public:
      int j;
};
class D : public C
{
      C::j;
};
```

T225 :"'%s' is not a base class for '%s'";

An access declaration was made for the specified identifier, but it is not a member of a base class.
A member of a class or structure cannot be accessed in another class that is not derived from the member's class.
The following is an example of this error:

```
class X
{
public:
   int x;
};
class A
{
public:
   int a;
};
class B : public A
{
public:
   X::x;       // error, B is not derived from X
   A::a;       // OK
};
```

T226 :"Overloaded operator '%s' cannot have default parameters";

Default parameters cannot be specified for the given operator.
It is illegal for overloaded operators to have default argument values.
The following is an example of this error:

```
class C
{
      int i;
friend C operator+(C c, int i=1);   // error
};
```

T227 :"Overloaded binary operator '%s' must be declared with '%s' parameters";

The specified binary operator (declared as a member function) was declared with too few or too more parameters.
The following is an example of this error:

```
class C
{
```

```
public:
      int i;
      C(int n) {i=n;}
 C operator[]();  // error
 C operator[](int i);   // OK
};
```

T228 :"Overloaded operator '%s' must have the last parameter of type 'int'";

When a postfix operator ++ or operator -- is overloaded, the last parameter must be declared with the type int.
The following is an example of this error:

```
class X
{
public:
   X operator++ ( X );    // error, nonvoid parameter
   X operator++ ( int );  // OK, int parameter
};
```

T229 :"Overloaded operator '%s' must be declared with one or two parameters";
The specified overloaded operator was declared with none parameters or with more than two parameters.
The following is an example of this error:

```
class X
{
public:
  X operator++ ( int, inr, int );  // error - too many parameters
};
```

T230 :"Overloaded unary operator '%s' must be declared with '%s' parameters";

The specified overloaded unary operator was incorrectly declared with a nonvoid parameter list.
The following is an example of this error:

```
class X
{
public:
   X operator! ( X );     // error, nonvoid parameter list
   X operator! ( void );  // OK
};
```

T231 :"Overloaded operator '%s' must be a method or must have at least one argument of type class or enumeration";

The specified overloaded operator was declared outside a class and does not have an explicit parameter of class type.
The following is an example of this error:

```
class C
{
public:
      int i;
      C(int n) {i=n;}
};
int operator+( int, int );  // error
```

T232 :"Overloaded operator '%s' must be a non-static member function";

The specified overloaded operator was not a member of a class, structure, or union.
The following operators can only be overloaded in class scope as nonstatic members: assignment operator '**=**', class member access operator '**->**', subscripting operator '**[]**', and function call operator '**( )**.
The following is an example of this error:

```
class C;
C operator ->();  // error
int operator[](); // error

class C
{
public:
      int i;
      C(int n) {i=n;}
C operator ->();        // OK
int operator[](int i);  // OK
};
```

T233 :"Overloaded operator '%s' cannot be static member";

Only ordinary member functions and the operators new and delete can be declared static.
Constructors, destructors and other operators must not be static.
This is an example of this error:

```
class C
{
public:
```

```
      int i;
      C(int n) {i=n;}
      static C operator +(C c);
};
```

T234 :"Overloaded operator '%s' must have the return type 'void'";

A redefinition of the operator **delete** contained a **return** statement that returned a type other than **void**. The return value for the operator **delete** must be **void**.
This is an example of this error:

```
class C
{
public:
      C operator delete(void *p);   // error, return type isn't
void
      void operator delete(void *p);// OK
};
```

T235 :"Overloaded operator '%s' must have the first parameter of type 'void*'";

The first formal parameter for an overridden based operator **delete** must be of type **void*.**
The following is an example of this error:

```
class C
{
public:
      void operator delete(int *p);//error
      void operator delete(void *p);//OK
};
```

T236 :"Overloaded operator '%s' must have the return type 'void*'";

A redefinition of the nonbased form of the operator **new** contained a **return** statement that returned a type that was not a **void** pointer. The return value for the operator **new** must be **void *.**
The following is an example of this error:

```
class C
{
public:
      int *operator new(size_t m);//error
      void *operator new(size_t m);//OK
};
```

T237 :"Overloaded operator '%s' must have the first parameter of type'size_t'";

The first formal parameter of the **near** or **far** forms of the operator **new** must be an unsigned **int.**

The following is an example of this error:

```
class C
{
public:
      void *operator new(char m);   //error
      void *operator new(size_t m); //OK
};
```

T238 :"Allocation and deallocation functions cannot be declared static in global scope.";

The **new /delete** operator was declared static at global scope.
The following is an example of this error:

```
class C
{
public:
      int i;
      C(int n) {i=n;}
};

static void* operator new(size_t m); // error
```

T239 :"Operator '%s' cannot be overloaded";

The specified operator was overloaded.
The following operators cannot be overloaded: class member access operator (**.**), pointer to member operator (**.***), scope resolution operator (**::**), conditional expression operator (**? :**), and **sizeof** operator.
The following is an example of this error:

```
class C
{
public:
      int operator .*();// error
};
```

T240 :"Illegal expression: '%s' cannot be resolved";

The specified operator was not defined for the specified types.
The following is an example of this error:

```
class C {
      int i;
public:
      C(int n) {i=n;}
      operator int() {return i;}
};
class D
{
      int i;
public:
      D(int n) {i=n;}
};

void main()
{
      int i=1;
      C c(3);
      D d(2);
      i>>d; //error
      i>>c; //OK, operator int() defined
}
```

T241 :"Illegal expression: '%s' is ambiguous";

Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed. An explicit cast of one or more of the actual parameters can resolve the ambiguity.
The following is an example of this error:

```
class C
{
      int i;
public:
      C(int n) {i=n;}
      C operator+(char i) {this->i+=i;return *this;}
      C operator+(unsigned char i){this->i-=i;return *this;}
};

void main()
{
      C c1(3),c2(1);
      c2=c1+32; // error - ambiguity!
      c2=c1+(char)32; // OK
}
```

T242 :"'%s': Illegal use of type 'void'";

The given variable was declared with the keyword **void**, which can be used only in function declarations.
The following is an example of this error:

```
void main()
{
      void i; // error
}
```

T243 :"Type of bit field too small for number of bits";

The number of bits specified in the bit-field declaration exceeded the number of bits in the given base type.
The following is an example of this error:

```
struct S
{
        unsigned i:1;
        int j:17;     //error
        int j:3;     //OK
};
```

T244 :"cannot evaluate expression to a constant";

The context requires a constant expression.
The specified array bound was not a constant expression.
An array must be declared with a constant bound.
The following example shows an illegal way to declare an array:

```
int i;
int A[i];        // error, i is nonconstant
```

and legal ways to declare an array:

```
const j = 20;
int A[j];        // OK, j is constant
int B[32];       // OK, 32 is a literal
```

T245 :"'%s' : references must be initialized";

A reference was not initialized when it was declared.

The following cases are the only times a reference can be declared without initialization:

- It is declared with the keyword **extern.**
- It is a member of a class, structure, or union and is initialized in the class's constructor function.
- It is declared as a parameter in a function declaration or definition.
- It is declared as the return type of a function.

The following is an example of this error:

```
void main()
{
      int a;
      int &ref; //error
      int &ref=a; //OK
}
```

T246 :"'%s' : const object must be initialized if not extern";

The specified identifier was declared as **const** but was not initialized.
An identifier must be initialized when declared as **const** unless it is declared as **extern**.
The following is an example of this error:

```
const int j;           // error
extern const int i;    // OK, declared as extern
```

T247 :"Illegal declaration: '%s' specified twice";

The specified modifier was repeated in statement.
The following example causes this error:

```
inline inline void func ();   // error
```

T248 :"Illegal declaration: '%s' outside of class";

The given function specfier is not allowed here. Either the specified function was declared with the **friend** specifier outside of a class, structure, or union, or the specified global function or class was declared as virtual (the keyword **virtual** refers only to members of a class or structure).
The following are examples of this error:

```
class A
{
private:
```

```
   void func1();
   friend void func2();
   virtual void f();       // OK
};
friend void func1() {};    // error
void func2() {};           // OK
virtual void f();          // error
```

T249 :"Illegal declaration: virtual used for static";

A **static** member function was declared as **virtual**.
The virtual function mechanism relies on the specific object that calls the function to determine which virtual function is used. Since this is not possible for **static** functions, they cannot be declared as **virtual**.
The following is an example of this error:

```
class C
{
      int i;
public:
      C(int n) {i=n;}
      static virtual void f(); // error
      virtual void f();        // OK
};
```

T250 :"Illegal declaration: multiple storage classes";

A declaration contained more than one storage class, as in:

```
extern static int i;      // error
```

A declaration can never have more than one storage class, either **auto, register, static**, or **extern.**

T251 :"Illegal declaration: multiple types specified";

A declaration can never have more than one of these basic types: **char, class, int, float, double, struct, union, enum, typedef name.**
The following is an example of this error:

```
int float i;   // error
```

T252 :"'%s' : 'friend' cannot be used during type definition";

A complete class declaration was given in a **friend** declaration.

A **friend** declaration can specify a member function or an elaborated type specifier, but not a complete class declaration.
The following is an example of this error:

```
class D { void func( int ); };

class A {
   friend class B { int i; };   // error
   friend class C;              // OK
   friend void D::func(int);    // OK
};
```

T253 :"A reference that is not to 'const' cannot be bound to a non-lvalue";

A reference cannot be initialized from a non-lvalue function return.
A nonconst reference must be initialized with an l-value, which makes the reference a name for that l-value. A function call is only an l-value if the return type of the function is a reference.
This is an example of this error:

```
struct X
{
    int i;
    X( int n ) {i=n;}
};

X f() {X x(3);return x;}

void main()
{
  X& r = f();   // error
}
```

T254 :"user-defined conversion '%s' takes no formal parameters";

A user-defined type conversion was declared as having one or more formal parameters. User-defined type conversions cannot take formal parameters.
Remove the formal parameters or choose an operator to overload.
This is an example of this error:

```
class C
{
      int i;
public:
      C(int n) {i=n;}
      operator int(char i);// error
      operator int();       // OK};
```

T255 :"user-defined conversion cannot specify a return type";

A user-defined conversion cannot specify a return type.
The following is an example of this error:

```
class X
{
public:
   int operator int() { return value; }   // error
   operator int() { return value; }       // OK
private:
   int value;
};
```

T256 :"user-defined conversion '%s' must be a non-static member function";

The specified user-defined conversion function was not a member of a class, structure, or union, and/or was declared as **static**.
Type conversions are required to be members of classes:
The following is an example of this error:

```
class X
public:
   static operator int() { return value; }   // error
   operator int() { return value; }       // OK
private:
   int value;
};
operator int() ;// error, not a member-function
```

T257 :"Cannot find a %s '%s'";

The formal parameter list of a function or pointer to a function did not match those of another function or pointer to a member function, respectively.
The assignment of the functions or pointers could not be made because of incompatible declarations.
The following are examples of this error:

```
int f(char a);
void main()
{
      int (*fp)(float a);
      fp=f; //error
}
//------------------------------
class C
```

```
{
        int i;
public:
        C(int n) {i=n;}
        void f() {}
};


void main()
{
        C c(1);
        void (C::*func)(int);
        func=&C::f;     // error
}
```

T258 :"Functions cannot return arrays";

A function cannot return an array. It can return a pointer to an array.
The following is an example of this error:

```
typedef int (m)[5];
m f();            // error
m *f();           // OK
```

T259 :"Functions cannot return functions";

A function cannot return a function. It can return a pointer to a function.
The following is an example of this error:

```
typedef int (funcptr)();
funcptr f();            // error
funcptr *f();           // OK
```

T260 :"Array size must be a positive integer constant";

Arrays must be declared with positive integral constant size.
The following is an example of this error:

```
void main()
{
      int m[8.5]; //error
      int m[-8];  //error
      int m[8];   //OK
}
```

T261 :"'%s' : array bounds overflow";

Too many initializers were present for the given array.
Make sure that the array elements and initializers match in size and quantity.
This error can be caused by not leaving space for the null terminator in a string.
The following is an example of this error:

```
char abc[4] = "abcd"; // error, array contains 5 members
```

T262 :"Cannot have an array of '%s'";

You are not allowed to declare arrays of: functions, references, bit fields, undimensioned array, void, bit, sbit, sfr, sfr16.
This is an example of this error:

```
typedef int (funcptr)();
funcptr f[3];           // error - array of functions
typedef int (m)[];
void main()
{
      void a[3];//aerror - array of void type
      int i;
      int &ref[3];//error - array of references
      m n[3];//error - array of undimensioned array
}
```

T263 :"Cannot have a reference to a '%s'";

You cannot refer to another reference or to a bit field.
This is an example of this error:

```
struct S
{
      int i:1;
};

void main()
{
      S s;
      int &ref=s.i; // error
}
```

T264 :"Cannot have a pointer to a '%s'";

An attempt was made to use the address of a bit field, or a reference or a bit, sbit, sfr, sfr16 type.
The following is an example of this error:

```
struct S
{
      int i:1;
};
void main()
{
      S s;
      &s.i;// error - pointer to a bit field
}
```

T265 :"'%s': bit field must have type 'int', 'signed int', or 'unsigned int'";

Bit fields must have an integral type. This includes enumerations.
The following is an example of this error:

```
struct S
{
      float i:1; // error
      int i:1; // OK
};
```

T266 :"'%s': named bit field cannot have zero width";

The given named bit field had zero width. Only unnamed bit fields are allowed to have zero width.
The following is an example of this error:

```
struct S
{
     char i:0;  // error
     char i:2;  // OK
};
```

T267 :"Illegal primitive type";

Two type specifiers had missing code between them.
You are not allowed to combine the two type specifiers.
The following is an example of this error:

```
int float i;   // error
unsigned signed b;
short long l;
```

```
short float a;
```

T268 :"Return outside of function";

A **return** statement wasn't placed elsewhere than in a function body or the return type of the function is undefined.

T269 :"Function '%s' must return a value";

The specified function was declared as returning a value, but the function definition did not contain a **return** statement or contained the **return** command without an accompanying return value.
The following is an example of this error:

```
int f(int a) {;}           // error
int f(int a) {return ;}    // error
int f(int a) {return 2*a;} // OK
void main()
{
}
```

T270 :"'%s': 'void' function returning a value";

The indicated function was declared as a **void** function but returned a value.
This error can be caused by an incorrect function prototype. If the function returns a value, the return type must be specified in the function declaration.
The following is an example of this error:

```
void f() {return 1;} // error
void f() {;}         // OK
void main()
{
}
```

T271 :"return of value in constructor/destructor is illegal";

The specified constructor/destructor returned a value.
A constructor/destructor cannot return a value of any type. This error is caused by defining a constructor/destructor that returns a value of any type, including a **void** return type.
This error can be eliminated by removing the **return** statement from the constructor/destructor definition.
The following is an example of this error:

```
class C
{
```

```
      int i;
public:
      C(int n) {i=n;return i;}
      ~C() {return i;}
};
```

T272 :"more than one default";

A **switch** statement contained more than one **default** label.
The following is an example of this error:

```
void main()
{
        int mark;
        char ch;
        mark=-1;
        ch=getchar();
        switch (ch)
        {
        case '1': mark=0;break;
        case '2': mark=1;break;
        default: processed(ch);
        default:mark=-1;        // error
        }
}
```

T273 :"default outside of switch";

The keyword **default** can appear only within a **switch** statement.
The following is an example of this error:

```
void main()
{
        int mark=-1;
        char ch;
        switch (ch)
        {
        case '1': mark=0;break;
        case '2': mark=1;break;
        }
        default: mark=3;        // error
}
```

T274 :"case label outside of switch";

The keyword **case** can appear only within a **switch** statement.
The following is an example of this error:

```
void main()
{
      int mark=-1;
      char ch;
      switch (ch)
      {
      case '1': mark=0;break; // OK
      default: mark=3;break;
      }
      case '2': mark=1;break; // error
}
```

T275 :"initialization of '%s' is skipped by 'default' label'";

The specified identifier initialization can be skipped in a **switch** statement.
It is illegal to jump past a declaration with an initializer unless the declaration is enclosed in a block.
The scope of the initialized variable lasts until the end of the **switch** statement unless it is declared in an enclosed block within the **switch** statement.
The following is an example of this error:

```
void func( void )
{
   int x;
   switch (x)
   {
   case 0 :
      int i = 1;       // error, skipped by default
      { int j = 1; }   // OK, initialized in enclosing block
   default :
      int k = 1;       // OK, initialization not skipped
   }
}
```

T276 :"initialization of '%s' is skipped by 'case' label'";

The specified identifier initialization can be skipped in a **switch** statement.
It is illegal to jump past a declaration with an initializer unless the declaration is enclosed in a block.
The scope of the initialized variable lasts until the end of the **switch** statement unless it is declared in an enclosed block within the **switch** statement.
The following is an example of this error:

```
void func( void )
{
   int x;
```

```
   switch ( x )
   {
   case 0 :
      int i = 1;        // error, skipped by case 1
      { int j = 1; }    // OK, initialized in enclosing block
   case 1 :
      int k = 1;        // OK, initialization not skipped
   }
}
```

T277 :"case expression not constant";

Case expressions must be integral constants.
The following is an example of this error:

```
void main()
{
      int mark=-1;
      char ch,i;
      switch (ch)
      {
      case '0': mark=0;break; // OK
      case i: mark=i;break;   // error
      default: mark=3;break;
      }
}
```

T278 :"case value '%d' already used";

A case value in a **switch** statement can be used only once.
The following is an example of this error:

```
void f()
{
    int i;
    switch( i )
    {
    case 0:
        break;
    case 0:    // error, case value '0' already used
        break;
    }
}
```

T279 :"'%s' : too many initializers for %s";

The number of initializers exceeded the number of objects to be initialized.
The following is an example of this error:

```
void main()
{
      int i[3][3]={{0,0,1},{0,1,0},{1,0,0,3}};        // error
      int i[3][3]={{0,0,1},{0,1,0},{1,0,0},{1,1,1}}; // error
      int i[3][3]={{0,0,1},{0,1,0},{1,0,0}};          // OK
      int i={0,0,1};                                  // error
      int i=9;                                        // OK
}
```

T280 :"'%s' : initializer for scalar variable requires one element";

You used an empty list to initialize a scalar variable. A scalar variable requires only one element.
The following is an example of this error:

```
void main()
{
      int i={}; // error
}
```

T281 :"Multiple values given for simple initialization";

The number of initializers exceeded the number of objects to be initialized.
The following is an example of this error:

```
void main()
{
  int i(0,9);     // error
}
```

T282 :"'%s' : must be initialized by constructor, not by '{}'";

The specified identifier was incorrectly initialized.
An initializer list is needed to initialize the following types:

- An array
- A class, structure or union that doesn't have constructors, private or protected members, base classes or virtual functions.

These types are known as "aggregates."
The following are examples of this error:

```
class C
{
public:
      int i;
};
```

```
class B: public C
{
      int j;
};

void main()
{
      B oc[3]={{1},{2},{3}}; // error
}
```

T283 :"'%s' : array of unknown bound cannot be initialized with empty initializer list";

An empty initializer list was used to initialize an array of unknown size.
The initializer list must have at least one value.
The following is an example of this error:

```
void main()
{
      char s[]={};      // error;
      char s[]={'p'};   // OK
}
```

T284 :"'%s' : can not be initialized by '{...}'";

The specified identifier was incorrectly initialized.
An initializer list is needed to initialize the following types:

- An array
- A class, structure or union that doesn't have constructors, private or protected members, base classes or virtual functions.
  These types are known as "aggregates."

The following are examples of this error:

```
class C
{
      int i;
public:
      C(int n) {i=n;}
};

void main()
{
      C c={1}; // error
      C c(8);  // OK
}
```

T285 :"Illegal expression: pointer required for operator delete";

The **delete** operator was used on an object that was not a pointer.
The **delete** operator can only be used on pointers.
The following is an example of this error:

```
void main()
{
   int i;
   delete i;            // error, i is not a pointer
   int *ip = new int;
   delete ip;           // OK
}
```

T286 :"Illegal expression: can't find operator delete";

The **delete** operator was called to delete the given type, which was declared but not defined.
Make sure that the class, structure, or union is defined before using the **delete** operator.
The following example generates this error:

```
class  B;
void   main()
{
   B *pb;
   delete pb; //error
}
```

T287 :"delete on <UNKNOWN> type";

The **delete** operator was used on an object (pointer) of an undeclared type.
The folowing is an example of this error:

```
void main()
{
    B *pb;      // error
    delete pb;  // error
}
```

T288 :"New initializer for non-class must be a single expression";

It was used an initializer containing more than one values for a non-class variable.

Only when you allocate space for an object you can specify more than one values for initializing, according to the defined constructor.
The following is an example of this error:

```
class C
{
      int i,j;
public:
      C(int m,int n) {i=m;j=n;}
};


void main()
{
      int *pi=new int(1,2);    // error
      int *pi=new int(87);     // OK
      C *pc=new C(2,3);        // OK
}
```

T289 :"size in array new must have integral type";

An array of non integral constant size was allocated.
The constant expression used to allocate or declare an array must be an integral type greater than zero.
The following is an example of this error:

```
void main()
{
      int *pi=new int[3.5];
}
```

T290 :" no match for function '%s'";

The specified function was not declared for the given parameters. In all the versions of the overloaded function, none of them had the same parameters in the same order as specified in the erroneous function call.
This error can be caused by a mismatch in the parameter list of the specified function.

Tips

- Ensure that the correct arguments are being passed when calling the function.
- Make sure the arguments are in the correct order when calling the function.
- Make sure the argument names are spelled correctly.

The following is an example of this error:

```
class C
{
public:
      int i;
      C(int a) {i=a;}
      void f(int a, char *s);
};

void C::f(char *s, int a)
{
}
```

T291 :" no match for function expression";

A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.
Make certain that your call to a function has the same parameters as the function prototype.
The following are examples of this error:

```
struct C
{
      int i;
      void f(unsigned char c) {}
      void f(char c) {}
};
void main()
{
      void (C::*func)(unsigned char);
      C c;
      char s=10;
      (c.*func)();// error
}
//--------------------
void func(int, int) {}
void main()
{
      func(1);    // error
}
```

T292 :"Illegal expression: call to non-function";

A call was made to a function through an expression that did not evaluate to a function pointer.
This error is probably caused by attempting to call a nonfunction.
The following is an example of this error:

```
class C
```

```
{
        int i;
public:
        void f() {};

int i, j;
char* p;
void main()
{
   j = i();    // error, i is not a function
   p();        // error, p doesn't point to a function
   void (C::*func)(int*);
   C c;
   func=&C::f;
   c.*func();  // error
   (c.*func)();// OK
}
```

T293 :"Cannot call the function 'main'";

Recursive calls of main( ) are not allowed. Cannot call 'main' from within the program.
The following is an example of this error:

```
void main()
{
      main();     // error
}
```

T294 :"Illegal expression: conditional components have incompatible types";

The types of the expressions on both sides of the colon in the conditional expression operator (?:) must be the same, except for the usual conversions.
These are some examples of usual conversions

```
char to int
float to double
void* to a particular pointer
```

In this expression, the two sides evaluate to different types that are incompatible.
The following is an example of this error:

```
class X
{
public:
      int i;
};

X x;
void f()
```

```
{
      int i,j;
      j = i ?     x : 1; // error
}
```

T295 :"Illegal expression: class type incompatible with pointer to member";

The right operand of a .*, ->*, or ::operator was not a pointer to a member of a class that is either identical to (or an unambiguous accessible base class of) the left operand's class type.
The following is an example of this error:

```
class C {};
class D {};

void main()
{
   D d, *pd;
   C c, *pc;
   int C::*pmc;

   pd->*pmc = 0;    // error
   d.*pmc = 0;      // error
   pc->*pmc = 0;    // OK
   c.*pmc = 0;      // OK
}
```

T296 :"Illegal expression: pointer to member expected";

The right side of a dot-star (.*) or an arrow star (->*) operator must be declared as a pointer to a member of the class specified by the left side of the operator.
In this case, the right side is not a member pointer.

The following is an example of this error:

```
class C
{
public:
      int i;
      void f() {}
};

void main()
{
      void (C::*func)();
      int C::*date;
      C c;
      int f;
```

```
      func=&C::f;
      date=&C::i;
      c.*f;         // error
     (c.*f)();      // error
      c.*date;      // OK
     (c.*func)();   // OK
}
```

T297 :"Illegal expression: left of '.%s' must have class/struct/union type";

The left side of the specified class member access operator (.) was not a class (or structure or union) type.
The following is an example of this error:

```
struct S
{
public:
    int member;
} s, *ps;
void main()
{
   int i;
   i.member = 0;   // error, i is not a class type
   ps.member = 0;  // error, ps is a pointer to a structure
   s.member = 0;   // OK, s is a structure type
   ps->member = 0; // OK, ps points to a structure S
}
```

T298 :"Illegal expression: left of '->%s' must point to class/struct/union type";

The left side of the specified class member access operator (->) was not a pointer to a class, structure, or union.
The following is an example of this error:

```
struct S
{
public:
    int member;
} *pS;
void main()
{
    int  *pInt;
    pInt->member = 0;   // error, pInt points to an int
    pS->member = 0;     // OK, pS points to a structure S
}
```

T299 :"'type cast' : conversion from %s to %s exists, but is inaccessible";

The specified private or protected member of a class, structure was accessed.

Tips
This error can be caused by accessing a public member of a base class that is inherited with private or protected access. The member should be accessed through a member function with public access or should be declared with public access.
The following is an example of this error:

```
class C
{
public:
      int i;
      C() {i=0;}
      void f() {}
};
class D: protected C {};
void main()
{
        D *pd;
        pd->C::i=0; // error
}
```

T300 :"'type cast' : cannot convert from '%s'*(type1)* to '%s'*(type2)*";

The compiler was unable to cast from '*type1*' to '*type2*.'
The following example illustrates this error.

```
void f() {}
int& g(int &i) {return i;}
void main()
{
        int m;
        g(m)=f(); // error
        void *ptr;
        int *pi;
        pi=ptr;  // error
}
```

T301 :"Illegal expression: ambiguous cast from type '%s' to type '%s'";

Both of the named overloaded functions could be used with the supplied parameters. The conversion was ambiguous because it could be done with either the specified constructor or the specifed conversion operator. This ambiguity is not allowed.
The following is an example of this error:

```
struct A;
```

```
struct B
{
   B(A);
   B();
};
struct A
{
   operator B();
};
A a;
B b = B(a);  // error
```

T302 :"Illegal expression: illegal cast from type 'type1' to type 'type2'";

A cast from type 'type1' to type 'type2' is not allowed.
The following example illustrates this error.

```
class C
{
  int i,j;
public:
  C() {i=j=0;}
};
void main()
{
    C c;
    if ((int)c); // error
}
```

T303 :"Attempt to take the address of a non-lvalue";

Your source file used the address-of operator (&) with an expression that can't be used that way.
The following is an example of this error:

```
int g(int &i) {return i;}
void main()
{
      &(g(1)); // error
}
```

T304 :"Illegal expression: index must be integer type";

A nonintegral expression was used in an array subscript.
The following is an example of this error:

```
void main()
{
```

```
      int a[10];
      int *pi;
      a[pi]=0; // error
}
```

T305 :"Illegal expression: attempt to index non-array or non-pointer";

A subscript was used on a variable that was not an array.
The following is an example of this error:

```
     void main()
     {
      int a,i;
      a[i]=0; // error
     }
```

T306 :"Illegal expression: expression incompatible with Boolean";

The expression couldn't be evaluated to a Boolean type, because it contains one ore more members of type struct, class or union.
The following is an example of this error:

```
class C
{
public:
      int i;
      C() {i=0;}
      void f() {}
};

void main()
{
      C c;
      int i;
      if (c && i) // error
        i++;
}
```

T307 :"use of type 'bool' in operation is restricted.";

This error is generated when you use a **bool** variable or value in an unexpected way. For example, if you use **bool** variable in conjunction with operators such as: +=,-=,*=,/=.
The following is an example of this error:

```
void main()
{
      bool b=false;
      int a;
      b+=b;
      b-=a;
}
```

T308 :"indirection to different types.";

The pointer expressions used with the given operator had different base types.
The pointer expressions were used without conversion.
The following is an example of this error:

```
class C
{
public:
        int i;
        C() {i=0;}
        void f() {}
};

class D
{
};
void main()
{
        C c,*pc;
        D d,*pd;
        pc=&c;
        pd=&d;
        pc=pc-pd; // error
}
```

T309 :"l-value must specify non-const object";

An attempt was made to modify an item declared with **const** type or the left operand of the given operator was not an l-value.
The following is an example of this error:

```
int f(int i) {return i;}
void main()
{
        const int a=0;
        int b;
        a=9;    // error
        f(b)=a; // error
}
```

T310 :"left operand must be a non-const l-value";

An attempt was made to modify an item declared with **const** type or the left operand of the given operator was not an l-value.
The following is an example of this error:

```
int f(int i) {return i;}
void main()
{
        const int a=0;
        int b;
        a+=9;   // error
        f(b)=a; // error
}
```

T311 :"'++' and '--' operators requires a l-value as operand";

The given operator did not have an l-value operand.
The following is an example of this error:

```
int  f1(int i) {return i;}
int& f2(int i) {return i;}
void main()
{
    int b;
    f1(b)++;    // error
    f2(b)++;    // OK
}
```

T312 :"'%s': illegal on operands of type '%s'";

The given unary operator was used with an illegal operand type, as in the following example:

```
void g(bool fFlag) {
   --fFlag;   // error
   fFlag--;   // error
}
void main()
{
    float f;
    ~f; // error
}
```

T313 :"Use of undefined type '%s'";

The specified type was not defined.

A type cannot be used until it is defined.
The folowing is an example of this error:

```
class B;
void main()
{
    B::i=0;
}
```

314 :"Illegal declaration: type must be defined before being used";

The specified identifier was declared as a class, structure, or union that was not defined.
The following is an example of this error:

```
class B;
void main()
{
    B b;
}
```

T315 :"Undefined qualifier '%s'";

The specified qualifier is not defined.

T316 :"Qualified name is not previously defined";

You are trying to reference a function or a variable as a member of a class/struct specificated by the qualifier, but it is not a member.
Check your declarations.
The following is an example of this error:

```
class B
{
      int i;
};
int B::f(){return 0;}   // error
static int B::j=0;      // error
```

T317 :"Illegal use of non-static member";

Non-static members cannot be used without an object. This means that you have written **class::member** where **'member'** is an ordinary (non-static) member, and there is no class to associate with that member.
The following is an example of this error:

```
class C
```

```
{
public:
      int i;
      C() {i=0;}
      void f() {}
};

void main()
{
    C c;
    C::i=0;       // error
    c.C::i=0;     // OK
}
```

T318: "'%s' : illegal reference to data member in a static member function";

The identifier you specified is a member of the class. However, it is nonstatic and the current function is a static member function. To access the member, an instance of the class must be provided and accessed using the **.** or **->** operators.
The following is an example of this error:

```
class C
{

public:
        int i;
        static int j;
        C() {i=0;}
        static void f();
};
void C::f()
{
  i++; // error
  j++; // OK
}
```

T319 :"'%s' : member from enclosing class is not a type name, static, or enumerator";

From within a nested class, you attempted to access a member of the enclosing class that was not a type name, a static member, or an enumerator.
The following is an example of this error:

```
int x;
class enclose
{
public:
   int x;
   static int s;
   class inner
```

```
   {
      void f()
      {
         x = 1;      // error; enclose::x is not static
         s = 1;      // ok; enclose::s is static
         ::x = 1;    // ok; ::x refers to global
      }
   };};
```

T320 :"local class cannot use local variables from enclosing function";

From within a nested class, you attempted to access a local variable of the enclosing function. A local class can access/use only the static local variables declared in the enclosing function.
The following is an example of this error:

```
void enclose ()
{
   int x;
   static int s;
   class inner
   {
      void f()
      {
         x = 1;      // error; enclose::x is not static
         s = 1;      // ok; enclose::s is static
      }
   };
};
```

T321 :"Illegal declaration: local class cannot have static members";

The specified member of a class, structure, or union with local scope was declared as **static**.
The following is an example of this error:

```
void func( void )
{
   class A
   {
 static int i;    // error, i is local to func
   };
};
class B
{
   static int i;       // OK
};
```

T322 :"'%s' : use of member as default parameter requires static member";

The specified nonstatic member was used as a default parameter.
The following is an example of this error:

```
class C
{
public:
   int i;
   static int j;
   void func1( int i = i );  // error, i is not static
   void func2( int i = j );  // OK, uses static j
};
```

T323 :"'%s' : illegal use of local variable as default parameter";

A local variable was illegally used as a default parameter.
The following is an example of this error:

```
int i;
void func();
{
   int j;
   extern void func2( int k = j );  // error, local variable
   extern void func2( int k = i );  // OK
}
```

T324 :"Cannot take address of function 'main'";

It is illegal to take the address of the main function.

T325 :"'main' function must return 'int' or 'void'";

You associated a return type to the **main** function, other than **int** or **void.**
The return type of the **main** function can be only **int** or **void.**
The following is an example of this error:

```
float main()
{
     return 1.2;
}
```

T326 :"Name '%s' cannot be used in expression";

The specified name cannot be used in expression because it represents neither a function nor a variable nor an enumerator.
The following is an example of this error:

```
class X
{
private:
       X& operator= (const Y& ) { return *this;}
};

class Y
{
public:
      X x;
};

void main()
{
    int i;
    i = Y(1);     // error
}
```

T327 :"cannot evaluate expression to a constant";

One of the enumerators of the **enum** type is not constant.
The **enum** type requires that all enumerators to be constant integer values.
The following is an example of this error:

```
void main()
{
    int i;
    enum colors {red=i,green,blue,yellow}; // error
    enum colors {red=0,green,blue,yellow}; // OK
}
```

T328 :"constant expression is not integral";

One of the enumerators of the **enum** type is not an integral expression.
All the constant values of an **enum** type must be integral.
The following is an example of this error:

```
void main()
{
    int i;
    enum colors {red=0.5,green,blue,yellow}; // error
    enum colors {red=0,green,blue,yellow};   // OK
}
```

T329 :"Cannot define '%s' inside function argument list";

Class and enumeration types may not be defined in a a function argument type.
You must define the given type before using it in this context.
The following is an example of this error:

```
void f(class C {});
void g(enum days {l,ma,mi} );
void main()
{
}
```

T330 :"Base type must be defined before it is used";

The specified base class was declared but never defined.
This error can be caused by a missing include file or an external base class that was not declared with the **extern** specifier.
The following is an example of this error:

```
class A;                 // error, A is undefined
class A {};              // OK, A is defined
class B : public A {};   // the error is detected here
```

T331 :"Base type must be a class";

The specified class was derived from a type name defined by a **typedef** statement or from an object of other type than class/struct.
The following is an example of this error:

```
int B;
typedef unsigned long ulong;
class C : public ulong {}; // error
class D : public B {};     // error - B is not a class
```

T332 :"Unions cannot have base classes";

A union was derived from a class, structure, or union.
The derived user-defined type must be declared as a class or structure.
The following is an example of this error:

```
class D {};

union UN : public D    // error
{
   char   ch;
   int    i;
   long   l;
   float  f;
   double d;
};
```

T333 :"member '%s' : has non trivial constructor";

The specified union member was declared with a default constructor.
A union member is not allowed to have a default constructor.
The following is an example of this error:

```
class A
{
   A(){}      // A has a default constructor
};
union U
{
   A a;      // error
};
```

T334 :"member '%s' : has non trivial copy constructor";

The specified union member was declared with a copy constructor.
A union member is not allowed to have a copy constructor.
The following is an example of this error:

```
class A
{

   A( const A& );  // A has a copy constructor
};
union U
{
   A a;            // error
};
```

T335 :"member '%s' : has non trivial copy assignment";

The specified union member was declared with an assignment operator, **operator=()**.
A union member is not allowed to have an assignment operator.
The following is an example of this error:

```
class A
{
   A& operator= ( const A& );  // A's assignment operator
};
union U
{
   A a;                        // error
};
```

T336 :"member '%s' : has non trivial destructor";

The specified union member was declared with a destructor, which is not allowed.
The following is an example of this error:

```
class A
{
   ~A();    // A has a destructor
};
union U
{
   A a;     // error
};
```

T337 :"Union '%s' : cannot have static member variable '%s'";

The specified union member was declared as **static**.
A union cannot have a **static** data member.
The following is an example of this error:

```
union UNN
{

   static int j; // error, j is static
   int i;        // OK
};
```

T338 :"'%s' : union can't have virtual methods";

The specified union was declared to have a virtual function.
Virtual functions can only be used with a class or structure but not with a union.
Change the specified union to a class or structure or make it a nonvirtual function.
The following is an example of this error:

```
union UN
{
  int i;
  virtual void f(); // error
  void f();         // OK
};
```

T339 :"'%s' 'identifier' was previously defined";

The specified identifier was already defined as type *type*.
The following is an example of this error:

```
struct S {};
class S {};    // error
```

T340 :"anonymous union defines non public member '%s'";

The specified member was declared with protected or private access.
A member of an anonymous union must have public access.
The following are an examples of this error:

```
void main()
{
   union
   {
   public:
      int i;     // OK, i is public
   protected:
      int j;     // error, j is protected
   private:
      int k;     // error, k is private
   };
}
```

T341 :"anonymous union type cannot have member functions";

The specified function was declared in an anonymous union.
An anonymous union cannot have member functions.
The following is an example of this error:

```
void main()
{
```

```
    union
       int i;
       void func( void );     // error, union is anonymous
    };
    union U
       void func2( void );    // OK
    };
}
```

T342 :"global anonymous unions must be declared static";

The anonymous union had global scope but was not declared as **static**.
The following is an example of this error:

```
union { int i; };          // error, not static
static union { int j; };   // OK
union U { int i; };        // OK, not anonymous
```

T343 :"anonymous class or struct: empty declaration";

The compiler detected an empty declaration using an untagged structure or class.
The following is an example of this error:

```
class C {}; // OK
class {};   // error
```

T344 :"Unable to define default destructor for class '%s'";

The compiler couldn't supply a default destructor for the specified class, because the specifed class contains a member that cannot be destroyed (an object of a class with a private destructor).
The following is an example of this error:

```
class X
{
      int i;
      ~X() {}
};
class Y     // error
{
      X x;
};
```

T345 :"Unable to define default assignment operator for class '%s'";

The compiler couldn't supply an assignment operator for the specified class, because the specifed class contains an object of a class with private assignment operator or because there is an assignment operator for the base class that is not accessible by the derived class.
The following is an example of this error:

```
class X
{
      int i;
      X operator= (X &x){ i=x.i;return *this;}
};
class Y     // error
{
      X x;
};
```

T346 :"Unable to define default constructor for class '%s'";

The compiler couldn't supply a default constructor for the specified class, because the base class has only user-defined constructors, or the default constructor of the base class has private access, or the specifed class contains a member that cannot be intialized (an object of a class without a default constructor).
The following are examples of this error:

```
class C
{
public:
 int i;
 C(int a) {i=a;}
};

class B : public C // error
{
};

class D;
class E : public D //error
{
}
```

T347 :"Unable to define default copy constructor for class '%s'";

The compiler couldn't supply a copy constructor for the specified class, because the specified class contains an object of a class with private copy constructor or because there is a copy constructor for the base class that is not accessible by the derived class.
The following is an example of this error:

```
class X
{
      int i;
      X(X &x) {}
};
class Y  // error
{
      X x;
};
```

T348 :"no assignment for '%s'";

No assignment operator was available for the specified class, structure, or union.
The following is an example of this error:

```
class X
{
public:
      X() {}
      X& operator=(X& ){return *this;}
};

class Y     // error
{
      const X x;
};
```

T349 :"no default constructor for '%s'";

No default constructor was available for the specified class, structure, or union.
The following are examples of this error:

```
class C
{
public:
 int i;
 C(int a) {i=a;}
};

class B
{
 C c; // error
};

class D;
class E : public D //error
{
}
```

T350 :"no copy constructor for '%s'";

No copy constructor was available for the specified class, structure, or union.
The following is an example of this error:

```
class X
{
public:
      X() {}
private:
      X(X&) {}
      X(X&,int=0){}
};
class Y     // error
{
      X x;
      Y() {}
};
```

T351 :"'%s' : must be initialized in constructor base/member initializer list";

The given constant was not initialized with an initializer list in the object constructor. The compiler left the constant undefined.
If a **const** or reference member variable is not given a value when it is initialized, it must be given a value in the object constructor.
The following is an example of this error:

```
class C
{
public:
        const int i;
        const int &ref;
        C(){}                   // error
        C() :i(0),ref(i){}      // OK
};
```

T352 :"Constructors, destructors and operators must be functions";

A constructor, destructor or an overloaded operator was declared with something other than function type.
For example:

```
class A
{
   A& operator +; // error - note missing parenthesis
  ~A;             // error - missing paranthesis
};
```

In the example, the function operator '()' is missing, so the operator does not have function type and generates this error (the same for the destructor).

T353 :"Illegal declaration: nothing is defined";

This declaration doesn't declare anything.
This should be a variable in the declaration. C++ requires that something be declared.
For example:

```
void main()
{
  int ;     // error
  int i;    // OK
}
```

T354 :"Illegal declaration: unknown specifier";

You used an unknown specifier. The specifier is none of the valid specifiers: **auto, register, static, const, volatile, friend, virtual, etc.**

T355 :"Illegal specifier: functions cannot be located at an absolute address.";

You tried to locate a function at an absolute address. Only global/static variables (excepted bit variables) can be located at absolute memory location using the ***_at_*** keyword.
Example:

```
int i _at_ 0x8000;      // OK
void f() _at_ 0x7800;   // error
void main()
{
  static int j _at_ 0xe000;   // OK - i is static
  int k _at_ 0x5000;          // error - j is not static
}
```

T356 :"Illegal parameters or return value types for function with extern alien specifier";

The return value and/or parameters of the function have incorrect types.
Parameters and return values of functions with **extern alien** specifier may be any of the following types: **bit, char, unsigned char, int and unsigned int.**

T357 :"A function definition with 'task' specifier must return void and must have an empty parameters list";

The function was declared with a return type different from void and/or with one or more arguments. Task functions must be declared with a void return type and a void argument list.

T358 :"task ID must be a number from 0 to 255 for RTX51 Full or 0 to 15 for RTX51 Tiny";

The **task id** wasn't in the required range of values. For the RTX51 Full operating system **task id** must be in the range 0-255 and for the RTX51 Tiny operating system it must be in the range 0-15.

T359 :"Register bank value must be in the range %d - %d";

The function uses a non-existing register bank (specifies an invalid value). There are 4 register banks, numbered from 0 to 3, so the register bank value specified after **using** keyword must be in the range 0-3.

T360 : "Priority value must be in the range %d - %d";

The **priority** value wasn't in the range 0-15.
The **priority** must be a value of type char/int/long in the range 0-15.

T361 :"Illegal return value type: function with register bank specifier cannot return bit value";

The function having the **register bank** specifier returned a bit value.
Functions with **register bank** specifer cannot return a bit value or a variable in registers.

T362 :"Illegal interrupt id must be in the range %d - %d ";

The **interrupt id** wasn't in the range 1-255.
Interrupt id must be a constant expression of type char/int/long in the range 1-255

T363 :"Illegal parameters or return value types for interrupt function";

The function was declared with a return type different from void and/or with one or more arguments. Interrupt functions MUST not have any arguments and must not return a value.

T364 :"Illegal parameter types (bit) for reentrant function";

One ore more arguments of the reentrant function have bit type.
The bit type for reentrant function arguments is not allowed.

T365 :"Illegal specifier: variables cannot have extern alien specifier";

The **extern alien** specifier was associated to a variable. Only PL/M-51 functions can be declared with **extern alien** specifier.

T366 :"Illegal specifier: variables cannot have task specifier";

The **task** specifier was associated to a variable. Only functions (with no arguments and **void** return type) can have **task** specifier.

T367 :"Illegal specifier: variables cannot have register bank specifier";

The **register bank** specifier was associated to a variable. Only interrupt functions and **main** function can have **register bank** specifier.

T368 :"Illegal specifier: variables cannot have interrupt specifier";

The **interrupt** specifier was associated to a variable. Only functions (with no arguments and **void** return type) can have **interrupt** specifier.

T369 :"Illegal specifier: variables cannot have memory model specifier";

A memory model specifier (**small, large, compact**) was associated to a variable. Only functions can have memory model specifier.

T370 :"Illegal specifier: variables cannot have reentrant specifier";

The **reentrant** specifier was associated to a variable. Only functions can have **reentrant** specifier.

T371 :"Illegal specifier: variables of type bit cannot be located at an absolute address.";

You tried to locate a variable of type bit at an absolute address.
Functions and variables of type bit cannot be located at an absolute address.

T372 :"Illegal specifier: variables with far memory type specifier cannot be located at an absolute address.";

You tried to locate a variable with **far** memory specifier at an absolute address.
The memory space **far** cannot be used together with the **_at_** keyword.

T373 :"Illegal declaration: interrupt ID specified twice";

The **interrupt** specifier was used more than once in the same declaration or definition.
For example:

```
void f() interrupt (1) interrupt (1); // error
```

```
void main() {}
```

T374 :"Illegal declaration: register bank specified twice";

The **register bank** specifier was used more than once in the same declaration or definition.
For example:

```
void f() interrupt (1) using(1) using(2); // error
void main()
{
}
```

T375 :"Illegal declaration: 'spec' specified twice";

The 'spec' specifier was used more than once in the same declaration or definition.
For example:

```
void f() interrupt (1) interrupt (1);           // error
void g() reentrant reentrant;                   // error
void g() interrupt(2) saveregbank saveregbank;  // error
void g() saveregs saveregs;                     // error
void main() user user                           // error
{}
```

T376 :"Illegal declaration: priority ID specified twice";

The **priority** specifier was used more than once in the same declaration or definition.
For example:

```
void g() interrupt(3) priority(1) priority(3); // error
```

T377 :"Illegal declaration: task ID specified twice";

The **task** specifier was used more than once in the same declaration or definition.
For example:

```
void g() _task_ 1 _task_ 3; // error
```

T378 :"Illegal declaration: memory model specified twice";

The memory model was used more than once in the same declaration or definition.
For example:

```
void g() small large; // error
```

T379 :"Illegal declaration: memory type specified twice";

The memory type specifier was used more than once in the same declaration or definition.
For example:

```
far far int i;    // error
bdata xdata int j;// error
near far int k;   // error
```

T380 :"Argument '%d' must be initialized for function '%s'";

A parameter was missing in a default parameter list.
If a default parameter is supplied anywhere in a parameter list, then all subsequent parameters on the right side of the default parameter must also be defined.
The following is an example of this error:

```
void func( int = 1, int);  // error
void func( int=1, int = 3);      // OK
```

T381 :"Complex constructors only valid for classes";

A function-style type cast of a built-in type can only take one argument. The error is generated when multiple arguments are supplied. For example:

```
void main()
{
  int j= int(1,3);
}
```

T382 :"Call to '%s' is ambiguous";

Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.
The following is an example of this error:

```
class C
{
public:
      int i;
      int operator() (int i) {return i;}
```

```
      int operator() (int i, int j=0) {return i+j;}
};
void main()
{
      C c;
      c(0); // error
}
```

T383 :"Call to '%s' cannot be resolved";

A call was made to a function through an expression that did not evaluate to a function pointer.
This error is probably caused by attempting to call a nonfunction (the name being called is not declared as a function).
The following is an example of this error:

```
class C
{
public:
      int i;
};
void main()
{
      C c;
      c(0); // error - c is the name of an object
}
```

T384 :"Illegal expression: member access to non-structure";

The left side of the specified class member access operator (->) was not a pointer to a class, structure, or union.

T385 :"Illegal expression: member access to undefined type";

You used class member access operator "**->**" on an object of an undefined class/struct (the class/struct was declared, but not defined).
The following is an example of this error:

```
class B;
void main()
{
      B *pb;
      pb->i=90; // error (class is declared but not defined)
}
```

T386 :"Wrong constructor init list";

A class in an initialization list was not a base class or member.
Only a member or base class can be in the initialization list for a class or structure.
The following is an example of this error:

```
class A
{
public:
   int i;
   A( int ia ) : B( i ) {};  // error, B is not a member of A
};
```

T387 :"Wrong initializer for '%s'";

A class member in an initialization list was initialized with more than one value.
Only a value can be assigned to a member in the initialization list for a class or structure.
The following is an example of this error:

```
class X
{
   int i;
   X() : i(1,9) {} // error - you specified more than one value for
i
};
```

T388 :"No matching initializer for '%s'";

A class member in an initialization list was initialized with a value of a type that couldn't be converted to that member type.
The following is an example of this error:

```
class Y {};
class X
{
      Y i;
      X():i(1){}  // error - i expected a value of type Y
};
```

T389 :"Illegal declaration: must initialize address variable here";

A reference was not initialized when it was declared.

The following cases are the only times a reference can be declared without initialization:

- It is declared with the keyword **extern.**
- It is a member of a class, structure, or union and is initialized in the class's constructor function.
- It is declared as a parameter in a function declaration or definition.
- It is declared as the return type of a function.

The following is an example of this error:

```
void main()
{
      int a;
      int &ref; //error
      int &ref=a; //OK
}
```

T390 :"Illegal declaration: bit fields cannot have storage static";

Only ordinary class data members can be declared static, not bit fields.
The following is an example of this error:

```
class C
{
  static int i:2;       // error
};
```

T391 :"Illegal declaration: union members cannot have constructors, destructors, or assignment";

The specified union member was declared with a default constructor or with a destructor or with an assignment operator.
A union member is not allowed to have a default constructor or a destructor or an assignment operator.

T392 :"Illegal declaration: illegal storage class for member";

The specified class member was declared with an illegal storage class. The **auto** and **register** storage class are not allowed in a class/struct/union and **static** storage class is not allowed in a union.
The following is an example of this error:

```
class C
{
  auto int i;     // error
```

```
  register int j; // error
  int k;          // OK
};
```

T393 :"Illegal declaration: illegal storage class for function";

The specified function was declared with an illegal storage class.
Only the **extern, static** and **extern alien** storage class can be used in conjunction with a function.

T394 :"Illegal declaration: cannot initialize class or union members";

Individual members of structs, unions, and classes can't have initializers.

T395 :"Illegal declaration: unbounded array member in class or union";

A class/structure or union contained an array with zero size.
The following is an example of this error:

```
class C { int a[]; };  // error
```

T396 :"Illegal declaration: can't specify storage class for typedef";

There was a storage class inside the **typedef** declaration.
The **typedef** declaration cannot include a storage class.
The following is an example of this error:

```
typedef extern unsigned int UINT; // error
```

T397 :"Illegal declaration: illegal typedef";

The **typedef** declaration was preceded by a storage class.
There cannot be a storage class in a **typedef** declaration.
The following is an example of this error:

```
extern typedef unsigned int UINT; // error
```

T398 :"Illegal declaration: function definition expected";

You used a modifier that request a function declaration/definition not a data declaration.

The **virtual**, **pure specifier**, **inline, explicit** modifiers cannot be used for data declarations.
The following is an example of this error:

```
class C
{
public:
      int i;
      C(int a) {i=a;}
      int f1=0;          // error
      explicit int f2;   // error
      inline int f3;     // error
};
```

T399 :"local class member functions must be defined within the class";

All members of classes declared local to a function must be entirely defined in the class definition.
This means that local classes cannot contain any static data members, and all of their member functions must have bodies defined within the class definition.
The following is an example of this error:

```
void f()
{
      class X
      {
            void f();
      };
}
```

T400 :"Cannot typedef a function type";

A **typedef** was used to define a function type.
For example:

```
typedef int functyp();
functyp func1 {};        // error
```

T401 :"Only non-static member functions can be const or volatile";

The specified **static** member function was declared with a **const** or **volatile** specifier.
The following is an example of this error:

```
class C
{
 public:
```

```
   static void func1() const; // error, func1 is static
   void func2() const;        // OK
};
```

T402 :"Methods can only be external";

A member function was defined at file scope. Member functions should be declared with external linkage.
The following are examples of this error:

```
class C
{


   static void func();
};

static void C::func(){};    // error
//-----------------------
class D
{
   void f() ;
};

static void D::f()      // error
{
}
```

T403 :"Illegal types for operator '%s'";

You used operands of type **bit, sbit** in conjunction with operators such as: +=,-=,*=,/=,&,||,==,!=. These operators can't be used with operands of type **bit** or **sbit.**
The following is an example of this error:

```
void main()
{
      sbit i,j;
      i+=2;       // error
      if (i||j);  // error
}
```

T404 :"Implicit constructor conversion is not allowed here because of 'explicit' specifier";

You tried to initialize an object having an explicit constructor, by using an assignment statement.
Explicit specifier doesn't permit an implicit conversion.
The following is an example of this error:

```
class C
{
public:
      int i;
      explicit C(int a) {i=a;}
};
void main()
{
      C c=9;      // error
      C c(9);     // OK
}
```

T405 :"'explicit' specifier is allowed only for constructors";

The function was declared with an **explicit** specifier. Only constructors can be defined with **explicit** specifier.
The following is an example of this error:

```
class C
{
public:
      int i;
      explicit C(int a) {i=a;}// OK
      explicit void f ();     // error
};

explicit void g();            // error
```

T406 :"Name '%s' must be declared first in his enclosing scope";

You are trying to reference '%s' as a member of 'class/struct', but it is not a member. You must declare it first in the specified class/structure:
The following is an example of this error:

```
class B;
int B::f(){return 0;} // error
```

T407 :"'%s': Structure members cannot be of type bit";

A class/structure member was declared of type **bit.**
Class/structures cannot have members of type **bit.**

The following is an example of this error:

```
class C
{
public:
      int i;
      C(int a) {i=a;}
      bit a;      // error
};
```

T408 :"'%s': Taking the address of a bit variable is illegal";

An attempt was made to take the address of a **bit** variable.
The following is an example of this error:

```
void main()
{
      bit a;
      void *pa=&a; // error
}
```

T409 :"'%s': Taking the address of a sfr variable is illegal";

An attempt was made to take the address of a **sfr** variable.
The following is an example of this error:

```
void main()
{
      sfr a;
      void *pa=&a; // error
}
```

T410 :"Enable use '%s' only for interrupt function ";

The **priority (n), using (n)** and **saveregbank** function modifiers were used in association with non-interrupt functions.
The above modifiers are allowed only on **interrupt** or **main** functions (**saveregbank** will be used only for an interrupt function).
The following is an example of this error:

```
void f() using(3); // error
void g() saveregbank; // error
```

T411 :"'system/user' modifier is allowed only on function 'main'";

The **system/user** modifier was used in other function than **main.**
This modifier is allowed only on **main** function.
The following is an example of this error:

```
void f() system;   // error
void main() system // OK
{
}
```

T412 :"'%s': Memory specifier must be used only in a global/static declaration";

Memory space (**sfr, code, data, bdata, ddata, xdata**) can only be used in a global/static declaration.
The following is an example of this error:

```
void main() user
{
      char code ch;     // error
      sfr int i;        // error
}
```

T413 :"'%s': Address location must be used only in a global/static declaration";

You tried to locate a local or non-static variable at an absolute memory location.
Absolute variable location can be performed for global/static variables only;
The following is an example of this error:

```
char xdata c _at_ 0x5000; // OK
void main()
{
      char c _at_ 0x3000; // error
}
```

T414 :"Pointers to sfr space are illegal";

Sfr space can only be used in a global/static declaration of a simple type variable (bit. char, int, long, float either signed or unsigned).
Every other usage is illegal including: structure, arrays, casting, pointers, typedef, etc.
Pointers to sfr space are illegal.
The following is an example of this error:

```
sfr int *a; // error
void main()
{
}
```

T415 :"Sfr space must be used only for basic types";

Sfr space can only be used in a global/static declaration of a simple type variable (bit. char, int, long, float either signed or unsigned).
Every other usage is illegal including: structure, arrays, casting, pointers, typedef, etc.
The following is an example of this error:

```
sfr int a[5]; // error
void main()
{
}
```

T416 :"'%s' : Sfr variables cannot be initialized";

A sfr variable was initialized.
Sfr variables cannot be initialized.
The following is an example of this error:

```
sfr int a=5; // error
void main() {}
```

T417 :"'%s' : Variable location out of adress space";

The variable was located at an address that isn't in the allowed range.
Here are the allowed ranges of addresses according to the memory space:
sfr = 0x400 - 0x7FF
data = 0 - 0xFFFFFF
code = 0 - 0xFFFFFF
xdata = 0 - 0xFFFF
ddata = 0 - 0x3FF, 0x10000 - 0x103FF ... (offset is 0 - 0x3FF)
bdata = offset is 0x20 - 0x3F
bit data/bdata = offset is 0x100 - 0x1FF
bit sfr = 0x200 - 0x3FF
The following is an example of this error:

```
sfr bit a _at_ 0x100;   // error
void main() user
{
}
```

T500 :"'#pragma C' directive is not ended with '#pragma END_C' ";

The **#pragma C** directive doesn't have its pair **'#pragma END_C** directive.
The above directives work only together.

T501 :"'#pragma END_C' directive was not begun with '#pragma C'";

The **#pragma END_C** directive doesn't have its pair **'#pragma C** directive.
The above directives work only together.
Pragma directives
#pragma PRAGMAC
#pragma ENDPRAGMAC
The code between PRAGMAC and ENDPRAGMAC is unchanged, and remains as C code
#pragma CODE_PRAGMA
#pragma COMPACT_PRAGMA
NOOJ_PRAGMA
#pragma OT_PRAGMA
#pragma OR_PRAGMA
#pragma PL_#pragma DB_PRAGMA
#pragma DISABLE_PRAGMA
#pragma EJECT_PRAGMA
#pragma FF_PRAGMA
#pragma I2_PRAGMA
#pragma IV_PRAGMA

- #pragma NOIV_PRAGMA
- #pragma LARGE_PRAGMA
- #pragma LISTINCLUDE_PRAGMA
- #pragma MAXARGS_PRAGMA
- #pragma NOAM_PRAGMA
- #pragma NOEXTEND_PRAGMA
- #pragma OJ_PRAGMA
- #pragma PRAGMA
- #pragma PW_PRAGMA
- #pragma PARM51_PRAGMA
- #pragma PARM251_PRAGMA
- #pragma PR_PRAGMA
- #pragma NOPR_PRAGMA
- #pragma REGFILE_PRAGMA
- #pragma ROM_PRAGMA
- #pragma SAVE_PRAGMA
- #pragma RESTORE_PRAGMA
- #pragma SMALL_PRAGMA
- #pragma SYMBOLS_PRAGMA
- #pragma WARNINGLEVEL_PRAGMA
- #pragma REGPARMS_PRAGMA
- #pragma NOREGPARMS_PRAGMA

Inline asm code

The syntax is as follows:

asm line:

__asm *assembly-language-instruction*

asm block:

__asm {
assembly-language-instructions
}

CHAPTER 6

EC++ LIBRARIES

CHAPTER 6

EC++ LIBRARIES

EC++ libraries

1. **<cctype>** - based on the standard C header file ctype.h
2. **<cerrno>** - based on the standard C header file errno.h
3. **<cfloat>** - based on the standard C header file float.h
4. **<climits>** - based on the standard C header file limits.h
5. **<cmath>** - based on the standard C header file math.h
6. **<csetjmp>** - based on the standard C header file setjmp.h
7. **<cstdarg>** - based on the standard C header file stdarg.h
8. **<cstddef>** - based on the standard C header file stddef.h
9. **<cstdio>** - based on the standard C header file stdio.h
10. **<cstdlib>** - based on the standard C header file string.h
11. **<cstring>**
12. **<complex>**
13. **<string>**
14. **<streambuf>**
15. **<stdiobuf>**
16. **<ios>**
17. **<istream>**
18. **<ostream>**
19. **<iostream>**
20. **<iomanip>**
21. **<new>**

<double_complex> and <float_complex>

```
class double_complex {
public:
        typedef double value_type;

   friend double_complex operator+(const double_complex&, const
double_complex&);
   friend double_complex operator+(const double_complex&, const double&);
   friend double_complex operator+(const double&, const double_complex&);
   friend double_complex operator-(const double_complex&, const
double_complex&);
   friend double_complex operator-(const double_complex&, const double&);
   friend double_complex operator-(const double&, const double_complex&);
   friend double_complex operator*(const double_complex&, const
double_complex&);
   friend double_complex operator*(const double_complex&, const double&);
   friend double_complex operator*(const double&, const double_complex&);
   friend double_complex operator/(const double_complex&, const
double_complex&);
   friend double_complex operator/(const double_complex&, const double&);
   friend double_complex operator/(const double&, const double_complex&);
   friend bool operator==(const double_complex&, const double_complex&);
   friend bool operator==(const double_complex&, const double&);
   friend bool operator==(const double& lhs, const double_complex& rhs);
   friend bool operator!=(const double_complex&, const double_complex&);
   friend bool operator!=(const double_complex&, const double&);
   friend bool operator!=(const double&, const double_complex&);
   friend istream& operator>>(istream&, double_complex&);
   friend ostream& operator<<(ostream&, const double_complex&);

   friend double real(const double_complex&);
   friend double imag(const double_complex&);
   friend double abs(const double_complex&);
   friend double arg(const double_complex&);
   friend double norm(const double_complex&);
   friend double_complex conj(const double_complex&);
   friend double_complex polar(const double&, const double&);
   friend double_complex cos (const double_complex&);
   friend double_complex cosh (const double_complex&);
   friend double_complex exp (const double_complex&);
   friend double_complex log (const double_complex&);
   friend double_complex log10(const double_complex&);
```

```
   friend double_complex pow(const double_complex&, int);
   friend double_complex pow(const double_complex&, const double&);
   friend double_complex pow(const double_complex&, const double_complex&);
   friend double_complex pow(const double&, const double_complex&);
   friend double_complex sin (const double_complex&);
   friend double_complex sinh (const double_complex&);
   friend double_complex sqrt (const double_complex&);
   friend double_complex tan (const double_complex&);
   friend double_complex tanh (const double_complex&);

   double_complex(double re = 0.0, double im = 0.0);
   double_complex(const float_complex& x);
   double real() const;
   double imag() const;

   double_complex& operator=(double);
   double_complex& operator+=(double);
   double_complex& operator-=(double);
   double_complex& operator*=(double);
   double_complex& operator/=(double);
   double_complex& operator=(const double_complex&);
   double_complex& operator+=(const double_complex&);
   double_complex& operator-=(const double_complex&);
   double_complex& operator*=(const double_complex&);
   double_complex& operator/=(const double_complex&);
private:
        double re, im;
};
```

The **double_complex** class describes an object that stores two objects of type double, one that represents the real part of a complex number and one that represents the imaginary part.
The first constructor initializes the stored real part to re and the stored imaginary part to im. The remaining constructor initializes the stored real part to x.real() and the stored imaginary part to x.imag().

```
class float_complex {
public:
        typedef float value_type;

   friend float_complex operator+(const float_complex&, const float_complex&);
   friend float_complex operator+(const float_complex&, const float&);
```

```
  friend float_complex operator+(const float&, const float_complex&);
  friend float_complex operator-(const float_complex&, const float_complex&);
  friend float_complex operator-(const float_complex&, const float&);
  friend float_complex operator-(const float&, const float_complex&);
  friend float_complex operator*(const float_complex&, const float_complex&);
  friend float_complex operator*(const float_complex&, const float&);
  friend float_complex operator*(const float&, const float_complex&);
  friend float_complex operator/(const float_complex&, const float_complex&);
  friend float_complex operator/(const float_complex&, const float&);
  friend float_complex operator/(const float&, const float_complex&);
  friend bool operator==(const float_complex&, const float_complex&);
  friend bool operator==(const float_complex&, const float&);
  friend bool operator==(const float&, const float_complex&);
  friend bool operator!=(const float_complex&, const float_complex&);
  friend bool operator!=(const float_complex&, const float&);
  friend bool operator!=(const float&, const float_complex&);
  friend istream& operator>>(istream&, float_complex&);
  friend ostream& operator<<(ostream&, const float_complex&);

  friend float real(const float_complex&);
  friend float imag(const float_complex&);
  friend float abs(const float_complex&);
  friend float arg(const float_complex&);
  friend float norm(const float_complex&);
  friend float_complex conj(const float_complex&);
  friend float_complex polar(const float&, const float&);
  friend float_complex cos (const float_complex&);
  friend float_complex cosh (const float_complex&);
  friend float_complex exp (const float_complex&);
  friend float_complex log (const float_complex&);
  friend float_complex log10(const float_complex&);
  friend float_complex pow(const float_complex&, int);
  friend float_complex pow(const float_complex&, const float&);
  friend float_complex pow(const float_complex&, const float_complex&);
  friend float_complex pow(const float&, const float_complex&);
  friend float_complex sin (const float_complex&);
  friend float_complex sinh (const float_complex&);
  friend float_complex sqrt (const float_complex&);
  friend float_complex tan (const float_complex&);
  friend float_complex tanh (const float_complex&);

  float_complex(float re = 0.0f, float im = 0.0f);
  float_complex(const double_complex& x);
  float real() const;
```

```
   float imag() const;
   float_complex& operator=(float);
   float_complex& operator+=(float);
   float_complex& operator-=(float);
   float_complex& operator*=(float);
   float_complex& operator/=(float);
   float_complex& operator=(const float_complex&);
   float_complex& operator+=(const float_complex&);
   float_complex& operator-=(const float_complex&);
   float_complex& operator*=(const float_complex&);
   float_complex& operator/=(const float_complex&);
private:
        float re, im;
};
```

The **float_complex** class describes an object that stores two objects of type float, one that represents the real part of a complex number and one that represents the imaginary part.
The first constructor initializes the stored real part to re and the stored imaginary part to im. The remaining constructor initializes the stored real part to x.real() and the stored imaginary part to x.imag().

double_complex:: double_complex
float_complex:: float_complex

- *Syntax*

double_complex(double `re` = 0.0, double `im` = 0.0);
float_complex(float `re` = 0.0f, float `im` = 0.0f);

- *Description*

The constructor initializes the stored real part to **`re and the imaginary part to im.`**

- **Example**

```
#include <complex>
void main()
{
        double_complex  dc(10,3);
double re=dc.imag();
double im=dc.real();
      }
```

double_complex:: double_complex
float_complex:: float_complex

- *Syntax*

double_complex(const float_complex& `x`);
float_complex(const double_complex& `x`);

- *Description*

The constructor initializes the stored real part to **`x.real()`** and the stored imaginary part to **`x.imag()`**.

- **Example**

```
#include <complex>
void main()
{
        double_complex  dc(10,3);
float_complex fc(dc);
float re=fc.imag();
float im=fc.real();
    }
```

double_complex:: real
float_complex:: real

- *Syntax*

double real() const;
float real() const;

- *Description*

Returns the real part of the complex number.

- **Example**

```
#include <complex>
void main()
{
        double_complex  dc(10,3);
double re=dc.imag();
double im=dc.real();
    }
```

double_complex:: imag
float_complex:: imag

- *Syntax*

double imag() const;
float imag() const;

- *Description*

Returns the imaginary part of the complex number.

- **Example**

```
#include <complex>
void main()
```

```
{
        double_complex  dc(10,3);
double re=dc.imag();
double im=de.real();
    }
```

double_complex::operator=
float_complex::operator=

- *Syntax*

double_complex& operator=(const double _complex& rhs);
double _complex& operator=(double rhs);
float_complex& operator=(const float_complex& rhs);
float_complex& operator=(float rhs);

- *Description*

The first member function replaces the stored real part with `rhs.real()` and the stored imaginary part with `rhs.imag()`.
The second member function replaces the stored real part with `rhs` and the stored imaginary part with zero.

- *Return Value*

Returns `*this`.

- **Example**

```
#include <complex>
void main()
{
        double_complex  c1,c2(3,8);
c1=c2;
c1=3;
    }
```

double_complex::operator+=
float_complex::operator+=

- *Syntax*

double_complex& operator+=(const double_complex& rhs);
double_complex& operator+=(double rhs);
float_complex& operator+=(const float_complex& rhs);
float_complex& operator+=(float rhs);

- *Description*

The first member function replaces the stored real and imaginary parts with those corresponding to the complex sum of `*this` and `rhs`.

The second member function adds `rhs` to the stored real part.

- *Return Value*

Returns `*this`.

- **Example**

```
#include <complex>
void main()
{
        double_complex  c1(10,3),c2(0,1);
c2+=c1;
double x=9.5;
c1+=x;
     }
```

double_complex::operator-=
float_complex::operator-=

- *Syntax*

double_complex& operator-=(const double _complex& `rhs`);
double_complex& operator-=(double `rhs`);
float_complex& operator-=(const float_complex& `rhs`);
float_complex& operator-=(float `rhs`);

- *Description*

The first member function replaces the stored real and imaginary parts with those corresponding to the complex difference of `*this` and `rhs`.

The second member function subtracts `rhs` from the stored real part.

- *Return Value*

Returns `*this`.

- **Example**

```
#include <complex>
void main()
{
        float_complex  c1(10,3),c2(3.5,4.8);
c2==c1;
float x=9.5;
c1-=x;
     }
```

double_complex::operator*=
float_complex::operator*=

- *Syntax*

double_complex& operator*=(const double _complex& `rhs`);
double _complex& operator*=(double `rhs`);

float_complex& operator*=(const float_complex& `rhs`);
float_complex& operator*=(float `rhs`);

- *Description*

The first member function replaces the stored real and imaginary parts with those corresponding to the complex product of `*this` and `rhs`.
The second member function multiplies both the stored real part and the stored imaginary part with `rhs`.

- *Return Value*

Returns `*this`.

- **Example**

```
#include <complex>
void main()
{
        float_complex  c1(10,3),c2(1,5);
c2*=c1;
float x=10;
c1*=x;
     }
```

double_complex::operator/=
float_complex::operator/=

- *Syntax*

double_complex& operator/=(const double _complex& `rhs`);
double _complex& operator/=(double `rhs`);
float_complex& operator/=(const float_complex& `rhs`);
float_complex& operator/=(float `rhs`);

- *Description*

The first member function replaces the stored real and imaginary parts with those corresponding to the complex quotient of `*this` and `rhs`.
The second member function multiplies both the stored real part and the stored imaginary part with `rhs`.

- *Return Value*

Returns `*this`.

- **Example**

```
#include <complex>
void main()
{
        float_complex  c1(10,3),c2(1,5);
c2/=c1;
float x=10;
c1/=x;
```

```
}
```

abs

- *Syntax*

double abs(const double_complex& x);
float abs(const float_complex& x);

- *Description*

The function returns the magnitude of x.

- **Example**

```
#include <complex>
void main()
{
        double_complex  dc(10,3);
        double x;
        x=abs(dc);
    }
```

arg

- *Syntax*

double arg(const double_complex& x);
float arg(const float_complex& x);

- *Description*

The function returns the phase angle of x.

- **Example**

```
#include <complex>
void main()
{
        double_complex  dc(10,3);
        double x;
        x=arg(dc);
     }
```

conj

- *Syntax*

double_complex conj(const double_complex& x);
float_complex conj(const float_complex& x);

- *Description*

The function returns the conjugate of x.

- **Example**

```
#include <complex>
```

```
void main()
{
        double_complex  x(10,3),xc(0,0);
        xc=conj(x);
    }
```

cos

- *Syntax*

double_complex cos (const double_complex& x);
float_complex cos (const float_complex& x);

- *Description*

The function returns the cosine of x.

- **Example**

```
#include <complex>
void main()
{
        float_complex  x(10,3),xc(0,0);
        xc=cos(x);
    }
```

cosh

- *Syntax*

double_complex cosh (const double_complex& x);
float_complex cosh (const float_complex& x);

- *Description*

The function returns the hyperbolic cosine of x.

- **Example**

```
#include <complex>
void main()
    {
        float_complex  x(2,3),xc(0,0);
        xc=cosh(x);
    }
```

exp

- *Syntax*

double_complex exp (const double_complex& x);
float_complex exp (const float_complex& x);

- *Description*

The function returns the exponential of x.

♦ **Example**

```
#include <complex>
void main()
    {
       float_complex  x(1,2),xexp(0,0);
       xexp=exp(x);
     }
```

imag

♦ *Syntax*

double imag(const double_complex& x);
float imag(const float_complex& x);

♦ *Description*

The function returns the imaginary part of x.

♦ **Example**

```
#include <complex>
void main()
    {
       float_complex  x(3,8);
       float im;
       im=imag(x);
    }
```

log

♦ *Syntax*

double_complex log (const double_complex& x);
float_complex log (const float_complex& x);

♦ *Description*

The function returns the logarithm of x. The branch cuts are along the negative real axis.

♦ **Example**

```
#include <complex>
void main()
    {
       double_complex  x(2,3),xl(0,0);
       xl=log(x);
    }
```

log10

♦ *Synatx*

double_complex log10(const double_complex& x);
float_complex log10(const float_complex& x);

- *Description*

The function returns the base 10 logarithm of `x`. The branch cuts are along the negative real axis.

- **Example**

```
#include <complex>
void main()
    {
       double_complex  x(2,3),xl(0,0);
       xl=log10(x);
    }
```

norm

- *Syntax*

double norm(const double_complex& x);
float norm(const float_complex& x);

- *Description*

The function returns the squared magnitude of `x`.

- **Example**

```
#include <complex>
void main()
    {
       double_complex  x(2,3);
       double n;
       n=norm(x);
    }
```

polar

- *Syntax*

double_complex polar(const double& rho, const double& theta);
float_complex polar(const float& rho, const float& theta);

- *Description*

The function returns the complex value whose magnitude is `rho` and whose phase angle is `theta`.

- **Example**

```
#include <complex>
void main()
    {
       double_complex  x(0,0);
       double rho=7;
```

```
    double theta=9;
    x=polar(rho,theta);
}
```

pow

- *Syntax*

double_complex pow(const double_complex& x, int y);
double_complex pow(const double_complex& x, const double& y);
double_complex pow(const double_complex& x, const double_complex& y);
double_complex pow(const double& x, const double_complex& y);

float_complex pow(const float_complex& x, int y);
float_complex pow(const float_complex& x, const float& y);
float_complex pow(const float_complex& x, const float_complex& y);
float_complex pow(const float& x, const float_complex& y);

- *Description*

The functions each effectively convert both operands to the return type, then return the converted `x` to the power `y`. The branch cut for `x` is along the negative real axis.

- *Return Value*

Returns the converted **x** to the power **y.**

- **Example**

```
#include <complex>
void main()
{
    float_complex  x(2,3),y(1,2),r(0,0);
    float a=3.9;int n=8;
    r=pow(x,n);
    r=pow(x,a);
    r=pow(x,y);
}
```

real

- *Syntax*

double real(const double_complex& x);
float real(const float_complex& x);

- *Description*

The function returns the real part of `x`.

- **Example**

```
#include <complex>
void main()
{
```

```
    float_complex  x(3,8);
    float re;
    re=real(x);
}
```

sin

- *Syntax*

double_complex sin (const double_complex& x);
float_complex sin (const float_complex& x);

- *Description*

The function returns the imaginary sine of `x`.

- **Example**

```
#include <complex>
void main()
{
    double_complex  x(1,2),xsin(0,0);
    xsin=sin(x);
}
```

sinh

- *Syntax*

double_complex sinh (const double_complex& x);
float_complex sinh (const float_complex& x);

- *Description*

The function returns the hyperbolic sine of `x`.

- **Example**

```
#include <complex>
void main()
{
    double_complex  x(1,2),xsinh(0,0);
    xsinh=sinh(x);
}
```

sqrt

- *Syntax*

double_complex sqrt (const double_complex& x);
float_complex sqrt (const float_complex& x);

- *Description*

The function returns the square root of `x`, with phase angle in the half-open interval `(-pi/2, pi/2]`. The branch cuts are along the negative real axis.

♦ **Example**

```
#include <complex>
void main()
{
        double_complex  x(1,2),xsqrt(0,0);
        xsqrt=sqrt(x);
      }
```

operator!=

♦ *Syntax*

bool operator!=(const double_complex& lhs, const double_complex& rhs);
bool operator!=(const double_complex& lhs, const double& rhs);
bool operator!=(const double& lhs, const double_complex& rhs);
bool operator!=(const float_complex& lhs, const float_complex& rhs);
bool operator!=(const float_complex& lhs, const float& rhs);
bool operator!=(const float& lhs, const float_complex& rhs);

♦ *Description*

The operators each return true only if **real**`(lhs) != real(rhs) ||` **imag**`(lhs) != ` **imag**`(rhs).`

♦ **Example**

```
#include <complex>
#include <iostream>
void main()
{
        double_complex  lhs(1,2),rhs(1,5);
        if (lhs!=rhs)
          cout<<"Different"<<endl;
else
  cout<<"Equal"<<endl;

      }
```

operator*

♦ *Syntax*

double_complex operator*(const double_complex& lhs, const double_complex& rhs);
double_complex operator*(const double_complex& lhs, const double& rhs);
double_complex operator*(const double& lhs, const double_complex& rhs);
float_complex operator*(const float_complex& lhs, const float_complex& rhs);
float_complex operator*(const float_complex& lhs, const float& rhs);
float_complex operator*(const float& lhs, const float_complex& rhs);

♦ *Description*

The operators each convert both operands to the return type, then return the complex product of the converted `lhs` and `rhs`.

♦ **Example**

```
#include <complex>
void main()
{
        double_complex  lhs(1,2),rhs(3,5.78),p(0,0);
        double x=3.25;
        p=lhs*rhs;
        p=x*rhs;
    }
```

operator+

♦ *Syntax*

double_complex operator+(const double_complex& lhs, const double_complex& rhs);
double_complex operator+(const double_complex& lhs, const double& rhs);
double_complex operator+(const double& lhs, const double_complex& rhs);
float_complex operator+(const float_complex& lhs, const float_complex& rhs);
float_complex operator+(const float_complex& lhs, const float& rhs);
float_complex operator+(const float& lhs, const float_complex& rhs);

♦ *Description*

The binary operators each convert both operands to the return type, then return the complex sum of the converted `lhs` and `rhs`.

The unary operator returns `lhs`.

♦ **Example**

```
#include <complex>
void main()
{
        double_complex  lhs(1,2),rhs(3,5.78),s(0,0);
        double x=3.25;
        s=lhs+rhs;
        s=lhs+x;
    }
```

operator-

♦ *Syntax*

double_complex operator-(const double_complex& lhs, const double_complex& rhs);
double_complex operator-(const double_complex& lhs, const double& rhs);

double_complex operator-(const double& lhs, const double_complex& rhs);
float_complex operator-(const float_complex& lhs, const float_complex& rhs);
float_complex operator-(const float_complex& lhs, const float& rhs);
float_complex operator-(const float& lhs, const float_complex& rhs);

- *Description*

The binary operators each convert both operands to the return type, then return the complex difference of the converted `lhs` and `rhs`.

The unary operator returns a value whose real part is `-real(lhs)` and whose imaginary part is `-imag(lhs)`.

- *Return Value*

Returns the complex difference of **lhs** and **rhs.**

- **Example**

```
#include <complex>
void main()
{
        double_complex  lhs(1,2),rhs(3,5.78),d(0,0);
        double x=3.25;
        d=lhs-rhs;
        d=lhs-x;
    }
```

operator/

- *Syntax*

double_complex operator/(const double_complex& lhs, const double_complex& rhs);
double_complex operator/(const double_complex& lhs, const double& rhs);
double_complex operator/(const double& lhs, const double_complex& rhs);
float_complex operator/(const float_complex& lhs, const float_complex& rhs);
float_complex operator/(const float_complex& lhs, const float& rhs);
float_complex operator/(const float& lhs, const float_complex& rhs);

- *Description*

The operators each convert both operands to the return type, then return the complex quotient of the converted `lhs` and `rhs`.

- *Return Value*

Returns the complex quotient of **lhs** and **rhs.**

- **Example**

```
#include <complex>
void main()
{
        double_complex  lhs(1,2),rhs(3,5.78),r(0,0);
        double x=3.25;
        r=lhs/rhs;
```

```
        r=lhs/x;
r=x/rhs;
    }
```

operator<<

- *Syntax*

ostream& operator<<(ostream& os, const double_complex& x);
ostream& operator<<(ostream& os, const float_complex& x);

- *Description*

The template function inserts the complex value `x` into the output stream `os`, effectively by executing:

```
ostringstream ostr;
ostr.flags(os.flags());
ostr.imbue(os.imbue());
ostr.precision(os.precision());
ostr << '(' << real(x) << ','
    << imag(x) << ')';
os << ostr.str().c_str();
```

Thus, if `os.width()` is greater than zero, any padding occurs either before or after the parenthesized pair of values, which itself contains no padding.

- *Return Value*

The function returns `os`.

- **Example**

```
#include <complex>
#include <iostream>
void main()
{
        double_complex  lhs(1,2),rhs(3,5.78);
        cout<<"First complex number: "<<lhs<<endl;
cout<<"Second complex number:"<<rhs<<endl;
    }
```

operator==

- *Syntax*

bool operator==(const double_complex& lhs, const double_complex& rhs);
bool operator==(const double_complex& lhs, const double& rhs);
bool operator==(const double& lhs, const double_complex& rhs);
bool operator==(const float_complex& lhs, const float_complex& rhs);
bool operator==(const float_complex& lhs, const float& rhs);
bool operator==(const float& lhs, const float_complex& rhs);

- *Description*

The operators each return true only if `real(lhs) == real(rhs) && imag(lhs) == imag(rhs).`

♦ **Example**

```
#include <complex>
#include <iostream>
void main()
{
        double_complex  lhs(1,2),rhs(3,5.78),s(1,0);
        double x=1;
        if (lhs==rhs)
          cout<<"lhs==rhs";
        if (s==x)
          cout<<Equality between s and x!";
    }
```

operator>>

♦ *Syntax*

istream& operator>>(istream& is, double_complex& x);
istream& operator>>(istream& is, double_complex& x);

♦ *Description*

The template function attempts to extract a complex value from the input stream `is`, effectively by executing:

```
is >> ch && ch == '('
is >> re >> ch && ch == ','
is >> im >> ch && ch == ')'
```

Here, `ch` is an object of type `char`, and `re` and `im` are objects of type `double/float`.

If the result of this expression is true, the function stores `re` in the real part and `im` in the imaginary part of `x`.

♦ *Return Value*

Returns `is`.

♦ **Example**

```
#include <complex>
#include <iostream>
void main()
{
        double_complex s(0,0);
        cout<<"Type in the complex number:"<<endl;
        cin>>s;
        cout<<"The complex number you typed is:"<<s;
    }
```

<string>

```
class string {
        friend string operator + (const string &lhs,const string &rhs);
        friend string operator + (const char *lhs,const string &rhs);
        friend string operator + (char lhs,const string &rhs);
        friend string operator + (const string &lhs,const char *rhs);
        friend string operator + (const string &lhs,char rhs);
        friend bool operator == (const string &lhs,const string &rhs);
        friend bool operator == (const char *lhs,const string &rhs);
        friend bool operator == (const string &lhs,const char *rhs);
        friend bool operator != (const string &lhs,const string &rhs);
        friend bool operator != (const char *lhs,const string &rhs);
        friend bool operator != (const string &lhs,const char *rhs);
        friend bool operator < (const string &lhs,const string &rhs);
        friend bool operator < (const char *lhs,const string &rhs);
        friend bool operator < (const string &lhs,const char *rhs);
        friend bool operator > (const string &lhs,const string &rhs);
        friend bool operator > (const char *lhs,const string &rhs);
        friend bool operator > (const string &lhs,const char *rhs);
        friend bool operator <= (const string &lhs,const string &rhs);
        friend bool operator <= (const char *lhs,const string &rhs);
        friend bool operator <= (const string &lhs,const char *rhs);
        friend bool operator >= (const string &lhs,const string &rhs);
        friend bool operator >= (const char *lhs,const string &rhs);
        friend bool operator >= (const string &lhs,const char *rhs);
        friend void swap(string &lhs, string &rhs);
        friend istream & operator >> (istream &is,string &str);
        friend ostream & operator << (ostream &os,const string &str);
        friend istream & getline (istream &is,string &str,char delim);
        friend istream & getline (istream &is,string &str);
private:
        char *string_ptr;
        size_t string_size;
public:
        typedef char* iterator;
        typedef const char* const_iterator;
        static const size_t npos;
        string();
        string(const string& str, size_t pos = 0, size_t n = npos);
        string(const char* s, size_t n);
        string(const char* s);
        string(size_t n, char c);
```

```
~string();
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
iterator begin();
const_iterator begin() const;
iterator end();
const_iterator end() const;
size_t size() const;
size_t length() const;
size_t max_size() const;
void resize(size_t n, char c);
void resize(size_t n);
size_t capacity() const;
void reserve(size_t res_arg = 0);
void clear();
bool empty() const;
const char & operator[](size_t pos) const;
char & operator[](size_t pos);
const char & at(size_t n) const;
char & at(size_t n);
string& operator+=(const string& str);
string& operator+=(const char* s);
string& operator+=(char c);
string& append(const string& str);
string& append(const string& str, size_t pos, size_t n);
string& append(const char* s, size_t n);
string& append(const char* s);
string& append(size_t n, char c);
string& assign(const string&);
string& assign(const string& str, size_t pos, size_t n);
string& assign(const char* s, size_t n);
string& assign(const char* s);
string& assign(size_t n, char c);
string& insert(size_t pos1, const string& str);
string& insert(size_t pos1, const string& str, size_t pos2, size_t n);
string& insert(size_t pos, const char* s, size_t n);
string& insert(size_t pos, const char* s);
string& insert(size_t pos, size_t n, char c);
iterator insert(iterator p, char c = char());
void insert(iterator p, size_t n, char c);
string& erase(size_t pos = 0, size_t n = npos);
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

```
string& replace(size_t pos1, size_t n1, const string& str);
string& replace(size_t pos1, size_t n1, const string& str,size_t pos2,
size_t n2);
string& replace(size_t pos, size_t n1, const char* s, size_t n2);
string& replace(size_t pos, size_t n1, const char* s);
string& replace(size_t pos, size_t n1, size_t n2, char c);
string& replace(iterator i1, iterator i2, const string& str);
string& replace(iterator i1, iterator i2, const char* s, size_t n);
string& replace(iterator i1, iterator i2, const char* s);
string& replace(iterator i1, iterator i2, size_t n, char c);
size_t copy(char* s, size_t n, size_t pos = 0) const;
void swap(string&);
const char* c_str() const; // explicit
const char* data() const;
size_t find (const string& str, size_t pos = 0) const;
size_t find (const char* s, size_t pos, size_t n) const;
size_t find (const char* s, size_t pos = 0) const;
size_t find (char c, size_t pos = 0) const;
size_t rfind(const string& str, size_t pos = npos) const;
size_t rfind(const char* s, size_t pos, size_t n) const;
size_t rfind(const char* s, size_t pos = npos) const;
size_t rfind(char c, size_t pos = npos) const;
size_t find_first_of(const string& str, size_t pos = 0) const;
size_t find_first_of(const char* s, size_t pos, size_t n) const;
size_t find_first_of(const char* s, size_t pos = 0) const;
size_t find_first_of(char c, size_t pos = 0) const;
size_t find_last_of (const string& str, size_t pos = npos) const;
size_t find_last_of (const char* s, size_t pos, size_t n) const;
size_t find_last_of (const char* s, size_t pos = npos) const;
size_t find_last_of (char c, size_t pos = npos) const;
size_t find_first_not_of(const string& str, size_t pos = 0) const;
size_t find_first_not_of(const char* s, size_t pos, size_t n) const;
size_t find_first_not_of(const char* s, size_t pos = 0) const;
size_t find_first_not_of(char c, size_t pos = 0) const;
size_t find_last_not_of (const string& str, size_t pos = npos) const;
size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
size_t find_last_not_of (const char* s, size_t pos = npos) const;
size_t find_last_not_of (char c, size_t pos = npos) const;
string substr(size_t pos = 0, size_t n = npos) const;
int compare(const string& str) const;
int compare(size_t pos1, size_t n1, const string& str) const;
int compare(size_t pos1, size_t n1, const string& str,size_t pos2, size_t
n2) const;
int compare(const char* s) const;
```

```
        int compare(size_t pos1, size_t n1,const char* s, size_t n2 = npos) const;
};
```

The **string** class describes an object that controls a varying-length sequence of elements of **type char**.

string::string

- *Syntax:*

#include <string>
explicit string();

- *Description:*

Constructs an empty string.

- **Example**

```
#include <string>
void main()
{
        string s();
    }
```

string::string

- *Syntax:*

#include <string>
string(const string& str, size_t pos = 0, size_t n = npos);

- *Description:*

Constructs a string from `n` characters of the sequence specified by `str`, beginning at position `pos`.

- **Example**

```
#include <string>
void main()
{
        string s1("abcdefg");
        string s2(s1,0,3);
    }
```

string::string

- *Syntax:*

#include <string>
string(const char* s, size_t n);

- *Description:*

Constructs a string from `n` characters of the sequence specified by `*s`.

♦ **Example**

```
#include <string>
void main()
{
        string s1("abcdefg",3);
    }
```

string::string

♦ *Syntax:*

#include <string>
string(const char* s);

♦ *Description:*

Constructs a string from the sequence specified by `*s`.

♦ **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG");
    }
```

string::string

♦ *Syntax:*

#include <string>
string(size_t n, char c);

♦ *Description:*

Constructs a string filled with `n` characters specified by `c`.

♦ **Example**

```
#include <string>
void main()
{
        string s1(3,'A');
    }
```

string::~string

♦ *Syntax:*

#include <string>
~string();

♦ *Description:*

Frees the memory allocated during the constructor call.

string::append

- *Syntax:*

#include <string>
string& append(const string& str);
string& append(const string& str, size_t pos, size_t n);
string& append(const char* s, size_t n);
string& append(const char* s);
string& append(size_t n, char c);

- *Description:*

The first function appends the sequence specified by `str` to the end of the sequence controlled by `*this`, then returns `*this`.

The second function appends *n* characters of the sequence specified by `str` starting at position `pos` to the end of the sequence controlled by `*this`, then returns `*this`.

The third function appends `n` characters of the sequence specified by `*s` to the end of the sequence controlled by `*this`, then returns `*this`.

The fourth function appends the sequence specified by `*s` to the end of the sequence controlled by `*this`, then returns `*this`.

The fifth function appends `n` copies of the character specified by `c` to the end of the sequence controlled by `*this,` then returns `*this`.

- **Return Value**

Returns ***this.**

- **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2;
        s2.append(s1);
s1.append("Sample",4);
s2.append(5,'0');
    }
```

string::assign

- *Syntax:*

#include <string>
string& assign(const string&);
string& assign(const string& str, size_t pos, size_t n);
string& assign(const char* s, size_t n);
string& assign(const char* s);
string& assign(size_t n, char c);

- *Description:*

The first function replaces the sequence controlled by `*this` with the sequence specified by `str`, then returns `*this`.

The second function replaces the sequence controlled by `*this` with `n` characters of the sequence specified by `str`, starting at position `pos`, then returns `*this`.

The third function replaces the sequence controlled by `*this` with `n` characters of the sequence specified by `*s,` then returns `*this`.

The fourth function replaces the sequence controlled by `*this` with the sequence specified by `*s`, then returns `*this`.

The fifth function replaces the sequence controlled by ***this*** with `n` copies of the character specified by `c`, then returns `*this`.

- **Return Value**

Returns ***this.**

- **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2("123"),s3;
        s2.assign(s1);
s3.assign("Sample",4);
s1.assign(5,'0');}
```

string::at

- *Syntax:*

```
#include <string>
const char & at(size_t n) const;
char & at(size_t n);
```

- *Description:*

Each member function returns a reference to the element of the controlled sequence at position *pos*, or reports an out-of-range error.

- **Example**

```
#include <string>
#include <iostream>
void main()
{
        string s1("ABCDEFG");
        char c=s1.at(3);
cout<<s1.at(5);
    }
```

string::begin

- *Syntax:*

```
#include <string>
```

iterator begin();
const_iterator begin() const;

- *Description:*

Each member function returns a random-access iterator that points at the first element of the sequence (or just beyond the end of an empty sequence).

- **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        char *s=s1.begin();

    }
```

string::c_str

- *Syntax:*

#include <string>
const char* c_str() const;

- *Description:*

The member function returns a pointer to a non-modifiable C string constructed by adding a terminating null element (`char(0)`) to the controlled sequence. Calling any non-const member function for `*this` can invalidate the pointer.

- **Return Value**

Returns a pointer to a non-modifiable C string.

- **Example**

```
#include <string>
#include <iostream>
void main()
{
        string s1("ABCDEFG");
cout<<s1.c_str();
    }
```

string::capacity

- *Syntax:*

#include <string>
size_t capacity() const;

- *Description:*

The member function returns the storage currently allocated to hold the controlled sequence, a value at least as large as `size()`.

- **Example**

```
#include <string>
void main()
```

```
{

        string s1("ABCDEFG");
        size_t n=s1.capacity();
    }
```

string::clear

- *Syntax:*

#include <string>
void clear();

- *Description:*

Actions like the destructor.

string::compare

- *Syntax:*

#include <string>
int compare(const string& str) const;
int compare(size_t pos1, size_t n1, const string& str) const;
int compare(size_t pos1, size_t n1, const string& str,size_t pos2, size_t n2) const;
int compare(const char* s) const;
int compare(size_t pos1, size_t n1,const char* s, size_t n2 = npos) const;

- *Description:*

The first function compares elements of the controlled sequence, if these arguments are not supplied, to the sequence specified by `str`.

The second function compares up to `n1` elements of the controlled sequence, beginning with position `p1`, to the sequence specified by `str`.

The third function compares up to `n1` elements of the controlled sequence beginning with position `p1`, to `n2` elements of `str` beginning with position `pos2`.

The fourth function compares elements of the controlled sequence, if these arguments are not supplied, to the sequence specified by `*s`.

The fifth function compares up to `n1` elements of the controlled sequence beginning with position `p1`, to `n2` elements of `*s` beginning with position `pos2`.

- **Return Value**

Each function returns:

- A negative value if the first differing element in the controlled sequence compares less than the corresponding element in operand sequence (`str or *s`), or if the two have a common prefix but the operand sequence (`str or *s`) is longer.
- Zero if the two compare equal element by element and are the same length.

- A positive value otherwise.

♦ **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2("ABCD");
        int c=s1. compare(s2);
        if (c==0)
          cout <<"Equality!";
        else
           if (c<0)
              cout<<"s1<s2";
           else
              cout <<"s1>s2";
     }
```

string::copy

♦ *Syntax:*

#include <string>
size_t copy(char* s, size_t n, size_t pos = 0) const;

♦ *Description:*

The member function copies up to `n` elements from the controlled sequence, beginning at position `pos`, to the array of `char`, beginning at `s`.

♦ *Return Value:*

Returns the number of elements actually copied

♦ *Example:*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        char s[10];
s1.copy(s,5,0);
     }
```

string::data

♦ *Syntax:*

#include <string>
const char* data() const;

♦ *Description:*

The member function returns a pointer to the first element of the sequence (or, for an empty sequence, a non-null pointer that cannot be derefferenced).

- *Example:*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        const char* s=s1.data();
    }
```

string::empty

- *Syntax:*

#include <string>
bool empty() const;

- *Description:*

The member function returns true for an empty controlled sequence.

- *Example:*

```
#include <string>
#include <iostream>
void main()
{
        string s1;
        if (s1.empty())
          cout<<"Empty string!";
}
```

string::end

- *Syntax:*

#include <string>
iterator end();
const_iterator end() const;

- *Description:*

Each member function returns a random-access iterator that points just beyond the end of the sequence.

- *Example:*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        char *s=s1.end();

    }
```

string::erase

- *Syntax:*

#include <string>
iterator erase(iterator first, iterator last);
iterator erase(iterator position);
string& erase(size_t pos = 0, size_t n = npos);

- *Description:*

The first member function removes the elements of the controlled sequence in the range `[first, last)`. The second member function removes the element of the controlled sequence pointed to by `it`. Both return an iterator that designates the first element remaining beyond any elements removed, or `end()` if no such element exists.

The third member function removes up to `n` elements of the controlled sequence beginning at position `p`, then returns `*this`.

- *Example:*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        s1.erase(1,2);
    }
```

string::find

- *Syntax:*

#include <string>
size_t find (const string& str, size_t pos = 0) const;
size_t find (const char* s, size_t pos, size_t n) const;
size_t find (const char* s, size_t pos = 0) const;
size_t find (char c, size_t pos = 0) const;

- *Description:*

The first function finds the first subsequence in the controlled sequence, beginning on or after position `pos`, that matches the sequence specified by `str`.

The second function finds the first subsequence in the controlled sequence, beginning on or after position `pos`, that matches the first `n` characters of sequence `*s`.

The third function finds the first subsequence in the controlled sequence, beginning on or after position `pos`, that matches the sequence specified by `*s`.

The fourth function finds the first character in the controlled sequence, beginning on or after position *pos*, that matches the character specified by `c`.

- *RETURN VALUE*

If it succeeds, it returns the position where the matching character was found. Otherwise, the function returns `npos`.

- *Example:*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        size_t pos;
        pos=s1.find("BCD",0);
    }
```

string::find_first_not_of

- *Syntax:*

#include <string>
size_t find_first_not_of(const string& str, size_t pos = 0) const;
size_t find_first_not_of(const char* s, size_t pos, size_t n) const;
size_t find_first_not_of(const char* s, size_t pos = 0) const;
size_t find_first_not_of(char c, size_t pos = 0) const;

- *Description:*

The first function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches none of the elements in the sequence specified by `str`.

The second function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches none of the elements in the sequence specified by the first `n` characters of `*s`.

The third function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches none of the elements in the sequence specified by `*s`.

The fourth function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that does not match with the character `c`.

- *RETURN VALUE*

If it succeeds, it returns the position. Otherwise, the function returns `npos`.

- *Example*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        size_t pos;
        pos=s1.find_first_not_of("CD",0);
    }
```

string::find_first_of

- *Syntax:*

#include <string>
size_t find_first_of(const string& str, size_t pos = 0) const;
size_t find_first_of(const char* s, size_t pos, size_t n) const;
size_t find_first_of(const char* s, size_t pos = 0) const;
size_t find_first_of(char c, size_t pos = 0) const;

- *Description*

The first function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches any of the elements in the sequence specified by `str`.

The second function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches any of the elements in the sequence specified by the first *n* characters of `*s`.

The third function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that matches any of the elements in the sequence specified by `*s`.

The fourth function finds the first (lowest position) element of the controlled sequence, at or after position `pos`, that does matches with the character `c`.

- *RETURN VALUE*

If it succeeds, it returns the position. Otherwise, the function returns `npos`.

- *Example*

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        size_t pos;
        pos=s1.find_first_not_of("CD",0);
    }
```

string::find_last_not_of

- *Syntax:*

#include <string>
size_t find_last_not_of (const string& str, size_t pos = npos) const;
size_t find_last_not_of (const char* s, size_t pos, size_t n) const;
size_t find_last_not_of (const char* s, size_t pos = npos) const;
size_t find_last_not_of (char c, size_t pos = npos) const;

- *Description*

The first function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that matches none of the elements in the sequence specified by `str`.

The second function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that matches none of the elements in the sequence specified by last `n` characters of `*s`.

The third function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that matches none of the elements in the sequence specified by `*s`.

The fourth function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that does not match with the character `c`.

- *RETURN VALUE*

If it succeeds, it returns the position. Otherwise, the function returns `npos`.

- **Example**

See the **find_first_of** example.

string::find_last_of

- *Syntax:*

#include <string>
size_t find_last_of (const string& str, size_t pos = npos) const;
size_t find_last_of (const char* s, size_t pos, size_t n) const;
size_t find_last_of (const char* s, size_t pos = npos) const;
size_t find_last_of (char c, size_t pos = npos) const;

- *Description*

The first function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that matches any of the elements in the sequence specified by `str`.

The second function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that matches any of the elements in the sequence specified by the last `n` characters of `*s`.

The third function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that matches any of the elements in the sequence specified by `*s`.

The fourth function finds the last (highest position) element of the controlled sequence, at or after position `pos`, that does matches with the character `c`.

- *RETURN VALUE*

If it succeeds, it returns the position. Otherwise, the function returns `npos`.

- **Example**

See the **find_first_of** example.

string::insert

- *Syntax:*

#include <string>
string& insert(size_t pos, const string& str);
string& insert(size_t pos, const string& str, size_t pos2, size_t n);
string& insert(size_t pos, const char* s, size_t n);
string& insert(size_t pos, const char* s);
string& insert(size_t pos, size_t n, char c);
iterator insert(iterator p, char c = char());

- *Description*

The first function inserts, after position `pos` or after the element it points to in the controlled sequence, the sequence specified by `str`.

The second function inserts, after position `pos` or after the element itpoints to in the controlled sequence, `n` characters of the sequence specified by `str`, beginning at position `pos`.

The third function inserts, after position `pos` or after the element it points to in the controlled sequence, the first `n` characters of the sequence specified by `*s`.

The fourth function inserts, after position `pos` or after the element it points to in the controlled sequence, the sequence specified by `*s`.

The fifth function inserts, after position `pos` or after the element it points to in the controlled sequence, `n` copies of character `c`.

The sixth function inserts, after position `p` in the controlled sequence, the character specified by `c`.

- *RETURN VALUE*

Returns `*this`.

- *Example*

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2("OK");
        char* s="123";
        s1.insert(3,s2);
s2.insert(1,s,2);
s2.insert(2,3,'c');
    }
```

string::length

- *Syntax:*

#include <string>
size_t length() const;

- *Description*

The member function returns the length of the controlled sequence (same as `size()`).

- **Example**

```
#include <string>
void main()
      {
         string s("ABCDEFG");
size_t n=s.length();
      }
```

string::max_size

- *Syntax:*

#include <string>
string::max_size
size_t max_size() const;

- *Description*

The member function returns the length of the longest sequence that the object can control.

- **Example**

```
#include <string>
void main()
     {
         string s("ABCDEFG");
size_t maxsize=s.max_size();
      }
```

`static const size_t` **`npos`** `= -1;`

The constant is the largest representing value of type `size_t`. It is assuredly larger than `max_size()`; hence it serves as either a very large value or as a special code.

string::operator+=

- *Syntax:*

#include <string>
string& operator+=(const string& str);
string& operator+=(const char* s);
string& operator+=(char c);

- *Description*

Each operator appends the *operand sequence* (the sequence specified by `str`, or by `*s`, or by `c`) to the end of the sequence controlled by `*this`, then returns `*this`.

- **Example**

```
#include <string>
void main()
     {
        string s("ABCDEFG");
s+='A';
s+="OK";
     }
```

string::operator=

- *Syntax:*

```
#include <string>
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char c);
```

- *Description*

Each operator replaces the sequence controlled by `*this` with the *operand sequence* (the sequence specified by `str`, or by `*s`, or by `c`), then returns `*this`.

- **Example**

```
#include <string>
void main()
     {
        string s("ABCDEFG");
s='A';
s="OK";
     }
```

string::operator[]

- *Syntax:*

```
#include <string>
onst char & operator[](size_t pos) const;
char & operator[](size_t pos);
```

- *Description*

Each member function returns a reference to the element of the controlled sequence at position `pos`. If that position is invalid, the behavior is undefined.

- **Example**

```
#include <string>
```

```
#include <iostream>
void main()
{
        string s("ABCDEFG");
        for (int i=0;i<s.size();i++)
          cout<<s[i];
}
```

string::replace

- *Syntax:*

#include <string>
string& replace(size_t pos1, size_t n1, const string& str);
string& replace(size_t pos1, size_t n1, const string& str,size_t pos2, size_t n2);
string& replace(size_t pos1, size_t n1, const char* s, size_t n2);
string& replace(size_t pos, size_t n1, const char* s);
string& replace(size_t pos, size_t n1, size_t n2, char c);
string& replace(iterator i1, iterator i2, const string& str);
string& replace(iterator i1, iterator i2, const char* s, size_t n);
string& replace(iterator i1, iterator i2, const char* s);
string& replace(iterator i1, iterator i2, size_t n, char c);

- *Description*

The first function replaces up to `n1` elements of the controlled sequence beginning with position `p1`. The replacement is the first `n1` characters of `str`.

The second function replaces up to `n1` elements of the controlled sequence beginning with position `p1`. The replacement is `n2` characters of `str`, beginning at position `pos2`.

The third function replaces up to `n1` elements of the controlled sequence beginning with position `pos1`. The replacement is the first `n2` specified by `*s`.

The fourth function replaces up to `n1` elements of the controlled sequence beginning with position `p1`. The replacement is the sequence specified by `*s`.

The fifth function replaces up to `n1` elements of the controlled sequence beginning with position `p1`. The replacement is `n2` copies of character `c`.

The sixth function replaces the elements of the controlled sequence beginning with the one pointed to by **i1**, up to but not including **i2**. The replacement is the sequence specified by `str`.

The seventh function replaces the elements of the controlled sequence beginning with the one pointed to by **i1**, up to but not including **i2**. The replacement is first `n` characters of the sequence specified by `*s`.

The eighth function replaces the elements of the controlled sequence beginning with the one pointed to by **i1**, up to but not including **i2**. The replacement is the sequence specified by `*s`.

The ninth function replaces the elements of the controlled sequence beginning with the one pointed to by **i1**, up to but not including **i2**. The replacement is `n` copies of the character `c`.

- *RETURN VALUE*

The function then returns `*this`.

- **Example**

```
#include <string>
void main()
    {
      string s1("ABCDEFG"),s2("Sample");
      s1.replace(2,3,"123");
      s1.replace(0,3,s2);
    }
```

string::reserve

- *Syntax:*

#include <string>
void reserve(size_t n = 0);

- *Description*

The member function ensures that `capacity()` henceforth returns at least `n`.

string::resize

- *Syntax:*

#include <string>
void resize(size_t n, char c);

- *Description*

The member function ensures that `size()` henceforth returns `n`. If it must lengthen the controlled sequence, it appends elements with value `c`.

- **Example**

```
#include <string>
void main()
    {
      string s1("ABCDEFG");
      s1.resize(20,"c");
    }
```

string::rfind

- *Syntax:*

#include <string>

size_t rfind(const string& str, size_t pos = npos) const;
size_t rfind(const char* s, size_t pos, size_t n) const;
size_t rfind(const char* s, size_t pos = npos) const;
size_t rfind(char c, size_t pos = npos) const;

- *Description*

The first function finds the last subsequence in the controlled sequence, beginning on or after position `pos`, that matches the sequence specified by `str`.

The second function finds the last subsequence in the controlled sequence, beginning on or after position `pos`, that matches the first `n` characters of sequence `*s`.

The third function finds the last subsequence in the controlled sequence, beginning on or after position `pos`, that matches the sequence specified by `*s`.

The fourth function finds the last character in the controlled sequence, beginning on or after position `pos`, that matches the character specified by `c`.

- *RETURN VALUE*

If it succeeds, it returns the position where the matching subsequence begins or the matching character was found. Otherwise, the function returns `npos`.

- **Example**

```
#include <string>
void main()
{
        string s1("ABCABFG");
        size_t pos=s1.rfind("AB",0);
    }
```

string::size

- *Syntax:*

#include <string>
size_t size() const;

- *Description*

The member function returns the length of the controlled sequence.

- **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG");
        size_t ln=s1.size();
    }
```

string::substr

- *Syntax:*

#include <string>
string substr(size_t pos = 0, size_t n = npos) const;

- *Description*

The member function returns an object whose controlled sequence is a copy of up to `n` elements of the controlled sequence beginning at position `pos`.

- **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2;
        s2=s1.substr(0,5);
     }
```

string::swap

- *Syntax:*

#include <string>
void swap(string & str);

- *Description*

Swaps the controlled sequences between `*this` and `str`.

- **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2("123");
        cout<<s1<<endl<<s2;
        s1.swap(s2);
        cout<<s1<<endl<<s2;
      }
```

string::iterator

The type describes an object that can serve as a random-access iterator for the controlled sequence. It is described here as a synonym for the type `char*`.

operator>>

- *Syntax:*

#include <string>
istream & operator >> (istream &is,string &str);

- *Description*

The function overloads `operator>>` to replace the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. Extraction stops:

- At end of file.
- After the function extracts `is.width ()` elements, if that value is nonzero.
- After the function extracts `is.max_size ()` elements.

After the function extracts a **space character** (0x09 - 0x0D or 0x20) in which case the character is put back.

If the function extracts no elements, it calls `setstate(ios_base::failbit)`. In any case, it calls `width(0)` and returns `*this`.

♦ **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2;
        cout<<s1<<endl<<s2;
        cin>>s2;
        cout<<s1<<endl<<s2;
     }
```

operator<<

♦ *Syntax:*

#include <string>
ostream & operator << (ostream &os,const string &str);

♦ *Description*

The function overloads `operator<<` to insert an object `str` of class `string` into the stream `os` The function effectively returns `os.write ( str.c_str (), str.size ())`.

♦ **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2;
        cout<<s1<<endl<<s2;
        cin>>s2;
        cout<<s1<<endl<<s2;
     }
```

getline

♦ *Syntax:*

#include <string>
istream & getline (istream &is,string &str);
istream & getline (istream &is,string &str,char delim);

♦ *Description*

The first function returns `getline(is, str, '\n')`.

The second function replaces the sequence controlled by `str` with a sequence of elements extracted from the stream `is`. In order of testing, extraction stops:

3. At end of file.
4. After the function extracts an element that compares equal to `delim`, in which case the element is neither put back nor appended to the controlled sequence.
5. After the function extracts `is.max_size ()` elements, in which case the function calls `setstate(ios_base::failbit)`.

If the function extracts no elements, it calls `setstate(failbit)`.

♦ *Return Value*

Returns `*this`.

♦ **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2("123");
        cout<<s1<<endl<<s2;
        getline(cin,s2);
        cout<<s1<<endl<<s2;
    }
```

operator+

♦ *Syntax:*

#include <string>
string operator + (const string &lhs,const string &rhs);
string operator + (const char *lhs,const string &rhs);
string operator + (char lhs,const string &rhs);
string operator + (const string &lhs,const char *rhs);
string operator + (const string &lhs,char rhs);

♦ *Description*

Each function overloads `operator+` to concatenate two objects of class `string`.

♦ *RETURN VALUE*

All effectively return `string(lhs).append(rhs)`.

♦ **Example**

```
#include <string>
void main()
{
        string s1("ABCDEFG"),s2("123"),s3;
        char *s="Sample";
```

```
        s3=s1+s2;
        s3=s2+s;
        s1='c'+s2;
    }
```

operator==

- *Syntax:*

#include <string>
bool operator == (const string &lhs,const string &rhs);
bool operator == (const char *lhs,const string &rhs);
bool operator == (const string &lhs,const char *rhs);

- *Description*

Each function overloads `operator==` to compare two objects of template class `string`.

- *RETURN VALUE*

All effectively return `string(lhs).compare(rhs) == 0`.

- **Example**

```
#include <string>
void main()
{
        string s1("123"),s2("123"),s3;
        if (s1==s2)
          cout<<"The strings are equal";
        if (s3=="OK")
          cout<<"OK";
    }
```

operator!=

- *Syntax:*

#include <string>
bool operator != (const string &lhs,const string &rhs);
bool operator != (const char *lhs,const string &rhs);
bool operator != (const string &lhs,const char *rhs);

- *Description*

Each function overloads `operator!=` to compare two objects of template class `string`. All effectively return `string(lhs).compare(rhs) != 0`.

- **Example**

```
#include <string>
void main()
    {
```

```
    string s1("123"),s2("213"),s3;
    if (s1!=s2)
      cout<<"Not equal";
    if (s3=="OK")
      cout<<"OK";
}
```

operator<

- *Syntax:*

#include <string>
bool operator < (const string &lhs,const string &rhs);
bool operator < (const char *lhs,const string &rhs);
bool operator < (const string &lhs,const char *rhs);

- *Description*

Each function overloads `operator<` to compare two objects of template class `string`.

- *RETURN VALUE*

All effectively return `string(lhs).compare(rhs) < 0`.

- **Example**

```
#include <string>
void main()
  {
    string s1("123"),s2("1423"),s3;
    if (s1<s2)
      cout<<"s1<s2";
    else
     if (s1>s2)
       cout<<"s1>s2";
}
```

operator>

- *Syntax:*

#include <string>
bool operator > (const string &lhs,const string &rhs);
bool operator > (const char *lhs,const string &rhs);
bool operator > (const string &lhs,const char *rhs);

- *Description*

Each function overloads `operator<` to compare two objects of template class `string`.

- *RETURN VALUE*

All effectively return `string(lhs).compare(rhs) > 0`.

- **Example**

```
#include <string>
void main()
    {
       string s1("123"),s2("213"),s3;
       if (s1<s2)
         cout<<"s1<s2";
       else
        if (s1>s2)
          cout<<"s1>s2";
    }
```

operator<=

- *Syntax:*

#include <string>
bool operator <= (const string &lhs,const string &rhs);
bool operator <= (const char *lhs,const string &rhs);
bool operator <= (const string &lhs,const char *rhs);

- *Description*

Each function overloads `operator<` to compare two objects of template class `string`.

- *RETURN VALUE*

All effectively return `string(lhs).compare(rhs) <= 0`.

- **Example**

```
#include <string>
void main()
    {
       string s1("123"),s2("123"),s3;
       if (s1<=s2)
        if (s1==s2)
          cout<<"s1=s2";
        else
          cout<<"s1<s2";
    }
```

operator>=

- *Syntax:*

#include <string>
bool operator >= (const string &lhs,const string &rhs);
bool operator >= (const char *lhs,const string &rhs);
bool operator >= (const string &lhs,const char *rhs);

- *Description*

Each function overloads `operator<` to compare two objects of template class `string`.

- *RETURN VALUE*

All effectively return `string(lhs).compare(rhs) >= 0`.

- **Example**

```
#include <string>
void main()
{
        string s1("1230"),s2("123"),s3;
        if (s1>=s2)
          if (s1==s2)
            cout<<"s1=s2";
          else
            cout<<"s1<s2";
     }
```

swap

- *Syntax:*

#include <string>
void swap(string &lhs, string &rhs);

- *Description*

Swaps the controlled sequences between `lhs` and `rhs`.

- **Example**

```
#include <string>
void main()
{
        string s1("123"),s2("321");
        cout<<s1<<endl<<s2<<endl;
        swap(s1,s2);
        cout<<s1<<endl<<s2<<endl;
    }
```

<stdiobuf>

class stdiobuf : public streambuf {
public:

```
stdiobuf(filedesc);
~stdiobuf();
 stdiobuf*       close();
virtual int          overflow(int=EOF);
virtual int          underflow();
virtual int          sync();
private:
filedesc       x_fd;
};
```

The **stdiobuf** class is a derived class of **streambuf** that is specialized for buffering to and from the standard I/O system.

stdiobuf::stdiobuf

- *Syntax:*

#include <stdiobuf>
stdiobuf(filedesc fd**);**

- *Description*

Constructs a **stdiobuf** object from a file descriptor. Objects of class **stdiobuf** are constructed from open standard I/O files, including **stdin**, **stdout**, and **stderr**. The object is unbuffered by default.

stdiobuf::~stdiobuf

- *Syntax:*

#include <stdiobuf>
~stdiobuf();

- *Description*

Destroys a **stdiobuf** object and, in the process, flushes the put area. The destructor does not close the attached file.

stdiobuf::close

- *Syntax:*

#include <stdiobuf>
stdiobuf* close();

- *Description*

Flushes any waiting output .

- *Return Value*

Return the address of the **stdiobuf** object.

stdiobuf::overflow

- *Syntax:*

#include <stdiobuf>
int overflow(int=EOF);

- *Description*

Empties the put area.

- *Return Value*

If the function cannot succeed, it returns **EOF.** Otherwise, it returns **nCh.**

stdiobuf::underflow

- *Syntax:*

#include <stdiobuf>
int underflow();

- *Description*

Fills the get area if necessary.

- Return Value

If the get area is empty, it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then **underflow** returns **EOF** and leaves the get area empty.

stdiobuf::sync

- *Syntax:*

#include <stdiobuf>
int sync();

- *Description*

Empties the get area and the put area and sends any unprocessed characters back to the source, if necessary.

- *Return Value*

EOF if an error occurs.

<streambuf>

```
class streambuf {
public:

   virtual ~streambuf();

   inline int in_avail() const;
   inline int out_waiting() const;
   int sgetc();
   int snextc();
   int sbumpc();
   void stossc();
   int sungetc();

   inline int sputbackc(char);

   inline int sputc(int);
   inline int sputn(const char *,int);
   inline int sgetn(char *,int);

   streampos pubseekoff(streamoff _o, ios::seekdir _w,int _m = ios::in | ios::out);
   streampos pubseekpos(streampos  _p, int _m = ios::in | ios::out);
   inline streambuf* pubsetbuf(char *, int);
   inline int pubsync();

   virtual int sync();

   virtual streambuf* setbuf(char *, int);
   virtual streampos seekoff(streamoff,ios::seekdir,int =ios::in|ios::out);
   virtual streampos seekpos(streampos,int =ios::in|ios::out);

   virtual int xsputn(const char *,int);
   virtual int xsgetn(char *,int);

   virtual int overflow(int =EOF) = 0; // pure virtual function
   virtual int underflow() = 0;        // pure virtual function
   virtual int uflow();

   virtual int pbackfail(int);

   void dbp();
```

```
    streambuf();
    streambuf(char *,int);

protected:

    inline char * base() const;
    inline char * ebuf() const;
    inline char * pbase() const;
    inline char * pptr() const;
    inline char * epptr() const;
    inline char * eback() const;
    inline char * gptr() const;
    inline char * egptr() const;
    inline int blen() const;
    inline void setp(char *,char *);
    inline void setp(char *,char *,char *);
    inline void setg(char *,char *,char *);
    inline void pbump(int);
    inline void gbump(int);

    void setb(char *,char *,int =0);
    inline int unbuffered() const;
    inline void unbuffered(int);
    int allocate();
    virtual int doallocate();
    virtual int showmanyc()      ;
private:
    int _fAlloc;
    int _fUnbuf;
    int x_lastc;
    char * _base;
    char * _ebuf;
    char * _pbase;
    char * _pptr;
    char * _epptr;
    char * _eback;
    char * _gptr;
    char * _egptr;
};
```

The class describes an abstract base class for deriving a **stream buffer**, which controls the transmission of elements (characters) to and from a specific representation of a stream.

Every stream buffer conceptually controls two independent streams, in fact, one for extractions (input) and one for insertions (output). It typically maintains some relationship between the two streams. What you insert into the output stream object, for example, is what you later extract from its input stream. When you position one stream of `filebuf` object, you position the other stream in tandem.

The public interface to template class `streambuf` supplies the operations common to all stream buffers, however specialized. The protected interface supplies the operations needed for a specific representation of a stream to do its work. The protected virtual member functions let you tailor the behavior of a derived stream buffer for a specific representation of a stream. The remaining protected member functions control copying to and from any storage supplied to buffer transmissions to and from streams. An **input buffer**, for example, is characterized by:

- `eback()`, a pointer to the beginning of the buffer
- `gptr()`, a pointer to the next element to read
- `egptr()`, a pointer just past the end of the buffer

Similarly, an **output buffer** is characterized by:

- `pbase()`, a pointer to the beginning of the buffer
- `pptr()`, a pointer to the next element to write
- `epptr()`, a pointer just past the end of the buffer

For any buffer, the protocol is:

- If the next pointer is null, no buffer exists. Otherwise, all three pointers point into the same sequence. (They can be safely compared for order.)
- For an output buffer, if the next pointer compares less than the end pointer, you can store an element at the **write position** designated by the next pointer.
- For an input buffer, if the next pointer compares less than the end pointer, you can read an element at the **read position** designated by the next pointer.
- For an input buffer, if the beginning pointer compares less than the next pointer, you can put back an element at the **putback position** designated by the decremented next pointer.

An object of class `streambuf` stores the six pointers described above.

streambuf::in_avail

♦ *Syntax:*

#include <streambuf>
int in_avail() const;

♦ *Description*

Returns the number of characters in the get area.

♦ *Return Value*

Returns the number of characters in the get area that are available for fetching. These characters are between the **gptr** and **egptr** pointers and may be fetched with a guarantee of no errors.

streambuf::streambuf

- *Syntax:*

#include <streambuf>
streambuf();

- *Description*

Makes an uninitialized **streambuf** object. This object is not suitable for use until a **setbuf** call is made. A derived class constructor usually calls **setbuf** or uses the second constructor.

streambuf::streambuf

- *Syntax:*

#include <streambuf>
streambuf(char ***pr,int** nLength**);**

- *Description*

Initializes the **streambuf** object with the specified reserve area or marks it as unbuffered.

streambuf::~streambuf

- *Syntax:*

#include <streambuf>
virtual ~streambuf();

- *Description*

The **streambuf** destructor flushes the buffer if the stream is being used for output.

streambuf:: out_waiting

- *Syntax:*

#include <streambuf>
int out_waiting() const;

- *Description*

Returns the number of characters in the put area.

- *Return Value*

Returns the number of characters in the put area that have not been sent to the final output destination. These characters are between the **pbase** and **pptr** pointers.

streambuf::sgetc

- *Syntax:*

#include <streambuf>
int sgetc();

- *Description*

Returns the character at the get pointer. The **sgetc** function does not move the get pointer. Returns **EOF** if there is no character available.

- **Example**

```
#include <iostream>
void main()
{
          int i;
 i=cin.rdbuf()->sgetc();
cout<<i;
      }
```

streambuf::snextc

- *Syntax:*

#include <streambuf>
int snextc();

- *Description*

Advances the get pointer, then returns the next character.

- *Return Value*

First tests the get pointer, then returns **EOF** if it is already at the end of the get area. Otherwise, it moves the get pointer forward one character and returns the character that follows the new position. It returns **EOF** if the pointer has been moved to the end of the get area.

- **Example**

```
#include <iostream>
void main()
{
    char s[3];
    for (int i=0;i<3;i++)
         s[i]=cin.rdbuf()->snextc();
        cout<<s;
     }
```

streambuf::sbumpc

- *Syntax:*

#include <streambuf>
int sbumpc();

- *Description*

Returns the current character, and then advances the get pointer.

- *Return Value*

Returns the current character, then advances the get pointer. Returns **EOF** if the get pointer is currently at the end of the sequence (equal to the **egptr** pointer).

- **Example**

```
#include <iostream>
void main()
{
        char s[3];
 for (int i=0;i<3;i++)
     s[i]=cin.rdbuf()->sbumpc();
          cout<<s;
    }
```

streambuf::stossc

- *Syntax:*

#include <streambuf>
void stossc();

- *Description*

Moves the get pointer forward one position, but does not return a character. If the pointer is already at the end of the get area, the function has no effect.

streambuf::stossc

- *Syntax:*

#include <streambuf>
int sungetc();

- *Description*

If a **putback position** is available, the member function decrements the next pointer for the input buffer and returns the current character. Otherwise it returns `pbackfail()`.

streambuf::stossc

- *Syntax:*

#include <streambuf>
int sputbackc(char ch**);**

- *Description*

Attempts to move the get pointer back one position. The *ch* character must match the character just before the get pointer.

- *RETURN VALUE*

EOF on failure.

streambuf::sputc

- *Syntax:*

#include <streambuf>
int sputc(int nCh**);**

- *Description*

Stores a character in the put area and advances the put pointer.

- *Return Value*

The number of characters successfully stored; **EOF** on error.

- **Example**

```
#include <iostream>
void main()
{
    cout.rdbuf()->sputc('P');
     }
```

streambuf::sputn

- *Syntax:*

#include <streambuf>
int sputn(const char *pCh**,int** nCount**);**

- *Description*

Copies *nCount* characters from *pch* to the **streambuf** buffer following the put pointer. The function repositions the put pointer to follow the stored characters.

- *RETURN VALUE*

The number of characters stored. This number is usually *nCount* but could be less if an error occurs.

- **Example**

```
#include <iostream>
void main()
{
        char *s="Example";
cout.rdbuf()->sputn(s,3);
     }
```

streambuf::sgettn

- *Syntax:*

#include <streambuf>
int sgetn(char *pCh**,int** nCount**);**

- *Description*

Gets the *nCount* characters that follow the get pointer and stores them in the area starting at *pch*. When fewer than *nCount* characters remain in the **streambuf**

object, **sgetn** fetches whatever characters remain. The function repositions the get pointer to follow the fetched characters.

- *Return Value*

The number of characters fetched.

- **Example**

```
#include <iostream>
void main()
{
        char s[5];
        cin.rdbuf()->sgetn(s,3);
    }
```

streambuf::pubseekoff

- *Syntax:*

#include <streambuf>
streampos pubseekoff(streamoff off**, ios::seekdir** way**,int** which **= ios::in | ios::out);**

- *RETURN VALUE*

The member function returns `seekoff(off, way, which).`

streambuf::pubseekpos

- *Syntax:*

#include <streambuf>
streampos pubseekpos(streampos sp**, int** which **= ios::in | ios::out);**

- *RETURN VALUE*

The member function returns `seekpos(sp,which).`

streambuf::pubsetbuf

- *Syntax:*

#include <streambuf>
streambuf* pubsetbuf(char ***s, int** n**);**

- *RETURN VALUE*

The member function returns `setbuf(s, n).`

streambuf::pubsync

- *Syntax:*

#include <streambuf>
int pubsync();

- *Return Value*

The member function returns `sync()`.

streambuf::sync

- *Syntax:*

#include <streambuf>
virtual int sync();

- *Description*

The **sync** function flushes the put area. It also empties the get area and, in the process, sends any unprocessed characters back to the source, if necessary.

- *Return Value*

EOF if an error occurs.

streambuf::setbuf

- *Syntax:*

#include <streambuf>
virtual streambuf* setbuf(char *pr**, int** nLength**);**

- *Description*

Attaches the specified reserve area to the **streambuf** object. Derived classes may or may not use this area.

- *Return Value*

A **streambuf** pointer if the buffer is accepted; otherwise **NULL**.

streambuf::seekoff

- *Syntax:*

#include <streambuf>
virtual streampos seekoff(streamoff off**,ios::seekdir** dir**,int** nMode**=ios::in|ios::out);**

- *Description*

Seeks to a specified offset (changes the position for the **streambuf** object).

- *Return Value*

The new position value. This is the byte offset from the start of the file (or string). If both **ios::in** and **ios::out** are specified, the function returns the output position. If the derived class does not support positioning, the function returns **EOF**.

streambuf::seekoff

- *Syntax:*

#include <streambuf>
virtual streampos seekpos(streampos pos**,int** nMode **=ios::in|ios::out);**

♦ *Description*

Seeks to a specified position (changes the position, relative to the beginning of the stream, for the **streambuf** object).

♦ *Return Value*

The new position value. If both **ios::in** and **ios::out** are specified, the function returns the output position. If the derived class does not support positioning, the function returns **EOF**.

streambuf::xsputn

♦ *Syntax:*

#include <streambuf>
virtual int xsputn(const char `*s`**,int** `n`**);**

♦ *Description*

Inserts up to `n` elements into the output stream, as if by repeated calls to `sputc`, from the array beginning at `s`.

♦ *Return Value*

It returns the number of elements actually inserted.

streambuf::xsgetn

♦ *Syntax:*

#include <streambuf>
virtual int xsgetn(char `*s`**,int** `n`**);**

♦ *Description*

The protected virtual member function extracts up to `n` elements from the input stream, as if by repeated calls to `sbumpc`, and stores them in the array beginning at `s`.

♦ *Return Value*

It returns the number of elements actually extracted.

streambuf::overflow

♦ *Syntax:*

#include <streambuf>
virtual int overflow(int nCh**=EOF) = 0;**

♦ *Description*

Empties the put area.

The **overflow** function is most frequently called by public **streambuf** functions like **sputc** and **sputn** when the put area is full, but other classes, including the stream classes, can call **overflow** anytime.

The function "consumes" the characters (writes these characters to a file) in the put area between the **pbase** and **pptr** pointers and then reinitializes the put area.

The **overflow** function must also consume *nCh* (if *nCh* is not **EOF**), or it might choose to put that character in the new put area so that it will be consumed on the next call.

♦ *Return Value*

EOF to indicate an error.

streambuf::underflow

♦ *Syntax:*

#include <streambuf>
virtual int underflow() = 0;

♦ *Description*

Fills the get area if necessary.

The **underflow** function is most frequently called by public **streambuf** functions like **sgetc** and **sgetn** when the get area is empty, but other classes, including the stream classes, can call **underflow** anytime.

The **underflow** function supplies the get area with characters from the input source. If the get area contains characters, **underflow** returns the first character.

♦ *Return Value*

If the get area is empty, it fills the get area and returns the next character (which it leaves in the get area). If there are no more characters available, then **underflow** returns **EOF** and leaves the get area empty.

streambuf::uflow

♦ *Syntax:*

#include <streambuf>
virtual int uflow();

♦ *Description*

The protected virtual member function endeavors to extract the current element ⊂ from the input stream, then advance the current stream position.

♦ *Return Value*

If the function cannot succeed, it returns **EOF**. Otherwise, it returns the current element ⊂ in the input stream.

streambuf::pbackfail

♦ *Syntax:*

#include <streambuf>
virtual int pbackfail(int nCh**);**

♦ *Description*

Augments the **sputbackc** function. This function is called by **sputbackc** if it fails, usually because the **eback** pointer equals the **gptr** pointer. The **pbackfail** function

should deal with the situation, if possible, by such means as repositioning the external file pointer.

- *Return Value*

The *nCh* parameter if successful; otherwise **EOF**.

streambuf::dbp

- *Syntax:*

#include <streambuf>
void dbp();

- *Description*

Prints buffer statistics and pointer values (writes ASCII debugging information directly on **stdout**).

streambuf::base

- *Syntax:*

#include <streambuf>
char * base() const;

- *Description*

Returns a pointer to the start of the reserve area.

- Return Value

  Returns a pointer to the first byte of the reserve area. The reserve area consists of space between the pointers returned by **base** and **ebuf**.

streambuf::ebuf

- *Syntax:*

#include <streambuf>
char * ebuf() const;

- *Description*

Returns a pointer to the end of the reserve area.

- *Return Value*

  Returns a pointer to the byte after the last byte of the reserve area. The reserve area consists of space between the pointers returned by **base** and **ebuf**.

streambuf::pbase

- *Syntax:*

#include <streambuf>
char * pbase() const;

- *Description*

Returns a pointer to the start of the put area.

- *Return Value*

Returns a pointer to the start of the put area. Characters between the **pbase** pointer and the **pptr** pointer have been stored in the buffer but not flushed to the final output destination.

streambuf::pptr

- *Syntax:*

#include <streambuf>
char * pptr() const;

- *Description*

Returns the put pointer.

- *Return Value*

Returns a pointer to the first byte of the put area. This pointer is known as the put pointer and is the destination for the next character(s) sent to the **streambuf** object.

streambuf::epptr

- *Syntax:*

#include <streambuf>
char * epptr() const;

- *Description*

Returns a pointer to the end of the put area.

- *Return Value*

Returns a pointer to the byte after the last byte of the put area.

streambuf::eback

- *Syntax:*

#include <streambuf>
char * eback() const;

- *Description*

Returns the lower bound of the get area.

- *Return Value*

Returns the lower bound of the get area. Space between the **eback** and **gptr** pointers is available for putting a character back into the stream.

streambuf::gptr

- *Syntax:*

#include <streambuf>
char * gptr() const;

- *Description*

Returns the get pointer.

- *Return Value*

Returns a pointer to the next character to be fetched from the **streambuf** buffer. This pointer is known as the get pointer.

streambuf::egptr

- *Syntax:*

#include <streambuf>
char * egptr() const;

- *Description*

Returns a pointer to the end of the get area.

- *Return Value*

Returns a pointer to the byte after the last byte of the get area.

streambuf::blen

- *Syntax:*

#include <streambuf>
int blen() const;

- *Description*

Returns the size of the reserve area.

- *Return Value*

Returns the size, in bytes, of the reserve area.

streambuf::setp

- *Syntax:*

#include <streambuf>
void setp(char *pp,char *pep);

- *Description*

Sets the values for the put area pointers (`pbase=pptr=pp; epptr=pep`).

streambuf::setp

- *Syntax:*

#include <streambuf>
void setp(char *pb,char *pp,char *pep);

- *Description*

Sets the values for the put area pointers (`pbase=pb; pptr=pp; epptr=pep`).

streambuf::setg

- *Syntax:*

#include <streambuf>
void setg(char *peb**,char** *pg**,char** *peg**);**

- *Description*

Sets the values for the get area pointers.

streambuf::pbump

- *Syntax:*

#include <streambuf>
void pbump(int nCount**);**

- *Description*

Increments the put pointer. No bounds checks are made on the result.

streambuf::gbump

- *Syntax:*

#include <streambuf>
void gbump(int nCount**);**

- *Description*

Increments the get pointer. No bounds checks are made on the result.

streambuf::setb

- *Syntax:*

#include <streambuf>
void setb(char *pb**,char** *peb**,int** nDelete**=0);**

- *Description*

  Sets the values of the reserve area pointers. If both *pb* and *peb* are **NULL**, there is no reserve area. If *pb* is not **NULL** and *peb* is **NULL**, the reserve area has a length of 0.

streambuf::unbuffered

- *Syntax:*

#include <streambuf>
int unbuffered() const;

- *Description*

Tests the **streambuf** buffer state variable.

- *Return Value*

Returns the current buffering state variable.

streambuf::unbuffered

- *Syntax:*

#include <streambuf>
void unbuffered(int nState**);**

- *Description*

Sets the value of the **streambuf** object's buffering state. This variable's primary purpose is to control whether the **allocate** function automatically allocates a reserve area.

streambuf::alllocate

- *Syntax:*

#include <streambuf>
int allocate();

- *Description*

Allocates a buffer, if needed, by calling **doalloc**.

- *Return Value*

If a reserve area already exists or if the **streambuf** object is unbuffered, **allocate** returns 0. If the space allocation fails, **allocate** returns **EOF**.

streambuf::doalllocate

- *Syntax:*

#include <streambuf>
virtual int doallocate();

- *Description*

By default, this function attempts to allocate a reserve area using operator **new**.

- *Return Value*

If the reserve area allocation fails, **doallocate** returns **EOF**.

streambuf::showmanyc

- *Syntax:*

#include <streambuf>
virtual int showmanyc();

- *Description*

The protected virtual member function returns a count of the number of characters that can be extracted from the input stream without the program experiencing an indefinite wait. The default behavior is to return zero.

<ios>

class ios_base {

public:

```
        enum fmtflags {
                skipws = 0x0001,
                unitbuf = 0x0002,
                uppercase = 0x0004,
                showbase = 0x0008,
        showpoint = 0x0010,
                showpos = 0x0020,
                left = 0x0040,
                right = 0x0080,
                internal = 0x0100,
                dec = 0x0200,
                oct = 0x0400,
                hex = 0x0800,
                scientific = 0x1000,
                fixed = 0x2000,
                boolalpha = 0x4000,
                adjustfield = 0x01c0,
                basefield = 0x0e00,
                floatfield = 0x3000
        };


   enum iostate  {  goodbit = 0x00,
                eofbit  = 0x01,
                failbit = 0x02,
                badbit  = 0x04 };

enum event {erase_event, imbue_event, copyfmt_event};

        class Init;

        long flags() const ;
        long flags(long fmtfl);
        long setf(long fmtfl);
        long setf(long fmtfl, long mask);
        void unsetf(long mask);
        streamsize precision() const ;
```

```
    streamsize precision(streamsize prec);
        streamsize width() const ;
        streamsize width(streamsize wide);
        ~ios_base();
        ios_base();

        static bool sync_with_stdio(bool sync = true);

        protected:
            long x_flags;
            int x_precision, x_width;
            int x_except, x_state;
            static bool _Sync;
            static int index;
            static long* iarray;
            static void** parray;
};
```

class ios_base::Init {

```
private:
        static int _Init_cnt;
protected:
        void cout_flush();
public:
        Init();
        ~Init();
};
```

The nested class describes an object whose construction ensures that the standard iostreams objects are properly constructed, even during the execution of a constructor for an arbitrary static object.

An object of class `ios_base` stores **formatting information**, which consists of:

- **Format flags** in an object of type `fmtflags`**.**
- An **exception mask** in an object of type `iostate`**.**
- A **field width** in an object of type *int* .
- A **display precision** in an object of type *int*.
- Two **extensible arrays**, with elements of type *long* and *void* pointer.

An object of class `ios_base` also stores **stream state information**, in an object of type `iostate`, and a **callback stack**.

```
class ios : public ios_base {

public:

        int delbuf() const ;
        void delbuf(int i) ;
        operator void*() const;
        bool operator!() const;
        int rdstate() const ;
void clear(int st = goodbit);
        void setstate(int st);
        bool good() const ;
        bool eof() const ;
        bool fail() const ;
        bool bad() const ;
int exceptions() const ;
        void exceptions(iostate except) ;
ios(streambuf* sb);
        ios();
        virtual ~ios();

        ostream* tie() const ;
        ostream* tie(ostream* _os) ;
        streambuf* rdbuf() const ;
        streambuf* rdbuf(streambuf* sb);
        ios& copyfmt(const ios& rhs);
        char fill() const ;
        char fill(char ch);

protected:

        char    x_fill;
        int     x_delbuf;   // if set, rdbuf() deleted by ~ios
        streambuf*  bp;
        ostream* x_tie;
        void init(streambuf* sb);

};
```

The class describes the storage and member functions common to both input streams (of class `istream`) and output streams (of class `ostream`). An object of class `ios` helps control a stream with characters.
An object of class `ios` stores:

- **`Formatting information`** and **`stream state information`** in a base object of type `ios_base;`
- a fill character in an object of type `char`;
- a tie pointer to an object of type `ostream` ;
- a stream buffer pointer to an object of type `streambuf;`

ios_base::fmtflags

enum **fmtflags** { skipws, unitbuf, uppercase, showbase, showpoint, showpos, left, right, internal, dec, oct, hex, scientific, fixed, boolalpha, adjustfield, basefield, floatfield };

The enumerated type **fmtflags** describes an object that can store **format flags**. The distinct flag values are:

- **skipws**, to skip leading **white space** (0x09 - 0x0D or 0x20) before certain extractions.
- **unitbuf**, to flush output after each insertion.
- **uppercase**, to insert uppercase equivalents of lowercase letters in certain insertions.
- **showbase**, to insert a prefix that reveals the base of a generated integer field.
- **showpoint**, to insert a decimal point unconditionally in a generated floating-point field.
- **showpos**, to insert a plus sign in a non-negative generated numeric field.
- **left**, to pad to a **field width** as needed by inserting **fill characters** at the end of a generated field (left justification).
- **right**, to pad to a **field width** as needed by inserting **fill characters** at the beginning of a generated field (right justification).
- **internal**, to pad to a **field width** as needed by inserting **fill characters** at a point internal to a generated numeric field.
- **dec**, to insert or extract integer values in decimal format.
- **oct**, to insert or extract integer values in octal format.
- **hex**, to insert or extract integer values in hexadecimal format.
- **scientific**, to insert floating-point values in scientific format (with an exponent field).
- **fixed**, to insert floating-point values in fixed-point format (with no exponent field).
- **boolalpha**, to insert or extract objects of type *bool* as names (such as `true` and `false`) rather than as numeric values.
- **adjustfield**, `internal | left | right`
- **basefield**, `dec | hex | oct`
- **floatfield**, `fixed | scientific`

ios_base::iostate

The enumerated type **iostate** describes an object that can store **stream state information**. The distinct flag values are:

- **goodbit**, no bits set
- **eofbit**, to record end-of-file while extracting from a stream
- **failbit**, to record a failure to extract a valid field from a stream
- **badbit**, to record a loss of integrity of the stream buffer

ios_base::event

The enumerated type **`event`** describes an object that can store the **callback event** used as an argument to a function registered with `register_callback`. The distinct event values are:

- **copyfmt_event**, to identify a callback that occurs near the end of a call to `copyfmt`, just before the **exception mask** is copied.
- **erase_event**, to identify a callback that occurs at the beginning of a call to `copyfmt`, or at the beginning of a call to the destructor for `*this`.
- **imbue_event**, to identify a callback that occurs at the end of a call to `imbue`, just before the function returns.

ios_base::flags

♦ *Syntax:*

#include <ios>
long flags() const ;
long flags(long `fmtfl`**);**

♦ *Description*

The first function reads the stream's format flags.
The second function sets the stream's internal flags variable to *fmtfl* and returns the previous value.

♦ *Return Value*

The first function returns the current value of the stream's format flags.
The second function returns the previous value of the stream's format flags.

♦ *Example*

```
#include <iostream>
void main()
{
  long f,i;
  int j;
  char
indic[15][12]={"skipws","left","right","internal","dec","oct","hex","showbase","sho
wpoint",    "uppercase", "showpos","scientific","fixed","unitbuf"};
```

```
  f=cout.flags();
  for (i=1,j=0;i<0x2000;i=i<<1,j++)
    if (i & f) cout<<indic[i][j]<<" is enabled"<<endl;
    else cout<<indic[i][j]<<" is disabled"<<endl;
  f=0x04a4;
  cout.flags(f);//enables all flags
   }
```

ios_base::setf

- *Syntax:*

#include <ios>
long setf(long `fmtfl`**);**
long setf(long `fmtfl`**, long** `mask`**);**

- *Description*

The first function manipulates the stream's format flags
The second function manipulates the stream's format flags

- *Return Value*

The first function turns on only those format bits that are specified by 1s in *fmtfl*. It returns a **long** that contains the previous value of all the flags.
The second function alters those format bits specified by 1s in *mask*. The new values of those format bits are determined by the corresponding bits in *fmtfl*. It returns a **long** that contains the previous value of all the flags.

- *Example*

```
#include <iostream>
void main()
{
   cout.setf(ios::showbase | ios::hex);
   cout<<100;//displays 0x64
   cout.setf(ios::oct,ios::hex | ios::oct);
   cout<<endl<<100;//displays 0144
    }
```

ios_base::unsetf

- *Syntax:*

#include <ios>
void unsetf(long `mask`**);**

- *Description*

Clears the format flags specified by 1s in *mask*.

- *Return Value*

It returns a **long** that contains the previous value of all the flags.

- *Example*

```
#include <iostream>
void main()
{
   cout.setf(ios::uppercase | ios::scientific);
   cout<<100.12;//displays 1.0012E+02
   cout.unsetf(ios::uppercase);
   cout<<endl<<100.12;//displays 1.0012e+02
    }
```

ios_base::precision

- *Syntax:*

#include <ios>
streamsize precision() const ;
streamsize precision(streamsize `prec`**);**

- *Description*

The first function reads the stream's floating-point format display precision.

The second function sets the stream's internal floating-point precision variable to *prec* . The default precision is six digits. If the display format is scientific or fixed, the precision indicates the number of digits after the decimal point. If the format is automatic (neither floating point nor fixed), the precision indicates the total number of significant digits.

- *Return Value*

The first function returns the stream's current precision value.
The second function returns the stream's previous precision value.

- *Example*

```
#include <iostream>
void main()
{
   int prec=cout.precision();
   cout.precision(4);
   cout.width(10);
   cout <<10.12345;//displays 10.12
    }
```

ios_base::width

- *Syntax:*

#include <ios>
streamsize width() const ;
streamsize width(streamsize `wide`**);**

- *Description*

The first function reads the stream's output field width.

The second function sets the stream's internal field width variable to *nw*. When the width is 0 (the default), inserters insert only the number of characters necessary to represent the inserted value. When the width is not 0, the inserters pad the field with the stream's fill character, up to *wide*. If the unpadded representation of the field is larger than *wide*, the field is not truncated. Thus, *wide* is a minimum field width.

The internal width value is reset to 0 after each insertion or extraction.

- *Return Value*

The first function returns the current value of the stream's width variable.
The second function returns the previous value of the stream's width variable.

- *Example*

```
#include <iostream>
void main()
{
        int w=cout.width();
  cout.precision(4);
  cout.width(10);
  cout <<10.12345;//displays 10.12
   }
```

ios_base::~ios_base

- *Syntax:*

#include <ios>
~ios_base();

- *Description*

Frees the memory allocated for the internal arrays (`extensible arrays`).

ios_base::ios_base

- *Syntax:*

#include <ios>
ios_base();

- *Description*

The (protected) constructor does nothing. A later call to `ios::init` must initialize the object before it can be safely destroyed. Thus, the only safe use for class `ios_base` is as a base class `ios`.

ios_base:: sync_with_stdio

- *Syntax:*

#include <ios>

static bool sync_with_stdio(bool `sync` **= true);**

- *Description*

The static member function stores a stdio sync flag, which is initially true. When true, this flag ensures that operations on the same file are properly synchronized between the **iostream** functions and those defined in the **Standard C library**. Otherwise, synchronization may or may not be guaranteed, but performance may be improved. The function stores `sync` in the stdio sync flag and returns its previous stored value. You can call it reliably only before performing any operations on the standard streams.

ios:: delbuf

- *Syntax:*

#include <ios>
int delbuf() const ;

- *Description*

Controls the connection of **streambuf** deletion with **ios** destruction

- *Return Value*

Returns the current value of the buffer deletion flag.

ios:: delbuf

- *Syntax:*

#include <ios>
void delbuf(int i) ;

- *Description*

Controls the connection of **streambuf** deletion with **ios** destruction
Assigns a value to the stream's buffer-deletion flag.

ios:: operator void*

- *Syntax:*

#include <ios>
operator void*() const;

- *Description*

Converts a stream to a pointer that can be used only for error checking.

- *Return Value*

The conversion returns 0 if either **failbit** or **badbit** is set in the stream's error state. See `rdstate` for a description of the error state masks. A nonzero pointer is not meant to be derefferenced.

ios:: operator!

♦ *Syntax:*

#include <ios>
bool operator!() const;

♦ *Description*

Returns a nonzero value if a stream I/O error occurs.

♦ *Return Value*

Returns a nonzero value if either **failbit** or **badbit** is set in the stream's error state. See `rdstate` for a description of the error state masks.

ios:: rdstate

♦ *Syntax:*

#include <ios>
int rdstate() const ;

♦ *Description*

Returns the stream's error flags.

♦ *Return Value*

Returns the current error state as specified by the following masks (**ios** enumerators):

- **ios::goodbit** No error condition.
- **ios::eofbit** End of file reached.
- **ios::failbit** A possibly recoverable formatting or conversion error.
- **ios::badbit** A severe I/O error or unknown state.

The returned value can be tested against a mask with the AND (**&**) operator

♦ *Example*

```
#include <iostream>
void main()
{
   int i; char ch; float f;
   cin>>i>>ch>>f;
   if (cin.rdtstate()!=ios::goodbit)
     { cin.clear();exit(1);}
      }
```

ios::clear

♦ *Syntax:*

#include <ios>
void clear(int st = goodbit);

♦ *Description*

Sets or clears the error-state flags. The **rdstate** function can be used to read the current error state.

- Example

```
#include <iostream>
void main()
{
   int i; char ch; float f;
   cin>>i>>ch>>f;
   if (cin.rdtstate()!=ios::goodbit)
     { cin.clear();exit(1);}
     }
```

ios::setstate

- *Syntax:*

#include <ios>
void setstate(int st);

- *Description*

The member function effectively calls `clear(st|rdstate()).`

- *Example*

```
#include <iostream>
void main()
{
int i; char ch; float f;
   cin>>i>>ch>>f;
   if (cin.rdtstate()!=ios::goodbit)
     { cin.setstate(ios::goodbit);exit(1);}
      }
```

ios::good

- *Syntax:*

#include <ios>
bool good() const ;

- *Description*

Indicates good stream status.

- *Return Value*

Returns a nonzero value if all error bits are clear. Note that the **good** member function is not simply the inverse of the **bad** function.

- *Example*

```
#include <iostream>
void main()
{
```

```
int i; char ch; float f;
cin>>i>>ch>>f;
if (!cin.good())
  { cin.setstate(ios::goodbit);exit(1);}
   }
```

ios::eof

- *Syntax:*

#include <ios>
bool eof() const ;

- *Description*

Indicates end of file.

- *Return Value*

Returns a nonzero value if end of file has been reached. This is the same as setting the **eofbit** error flag.

- *Example*

```
#include <iostream>
void main()
{
          char ch[5];
         cin>>ch;
          if (cin.eof())
          cin.clear();
    }
```

ios::fail

- *Syntax:*

#include <ios>
bool fail() const ;

- *Description*

Indicates a serious I/O error or a possibly recoverable I/O formatting error.

- *Return Value*

Returns a nonzero value if any I/O error (not end of file) has occurred. This condition corresponds to either the **badbit** or **failbit** error flag being set. If a call to **bad** returns 0, you can assume that the error condition is nonfatal and that you can probably continue processing after you clear the flags.

- *Example*

```
#include <iostream>
void main()
{
  char ch[5];
```

```
   cin>>ch;
   if (cin.fail())
     if (cin.bad())
          {cin.clear();exit(1);}
        //other code lines
      }
```

ios::bad

- *Syntax:*

#include <ios>
bool bad() const ;

- *Description*

Indicates a serious I/O error.

- *Return Value*

Returns a nonzero value to indicate a serious I/O error. This is the same as setting the **badbit** error state. Do not continue I/O operations on the stream in this situation.

- *Example*

```
#include <iostream>
void main()
{
   char ch[5];
   cin>>ch;
   if (cin.fail())
     if (cin.bad())
      {cin.clear();exit(1);}
       //other code lines
}
```

ios::exceptions

- *Syntax:*

#include <ios>
int exceptions() const ;
void exceptions(iostate `except`**) ;**

- *Description*

The first function returns the stored exception mask.
The second function stores `except` in the exception mask and returns its previous stored value.

- *Return Value*

The first function returns the stored **exception mask**.
The second function returns the previous stored value

ios::ios

- *Syntax:*

#include <ios>
ios(streambuf* `sb`**);**

- *Description*

Constructor for **ios**. Sets the stream buffer pointer to `sb` value.

ios::ios

- *Syntax:*

#include <ios>
ios();

- *Description*

Constructor for **ios**.

ios::~ios

- *Syntax:*

#include <ios>
virtual ~ios();

- *Description*

Virtual destructor for **ios**.

ios::tie

- *Syntax:*

#include <ios>
ostream* tie() const ;
ostream* tie(ostream* `os`**) ;**

- *Description*

The first function ties a specified **ostream** to this stream.
The second function ties a specified **ostream** to this stream.

- *Return Value*

Returns the value of the previous tie pointer or **NULL** if this stream was not previously tied.

ios::rdbuf

- *Syntax:*

#include <ios>
streambuf* rdbuf() const ;

streambuf* rdbuf(streambuf* `sb`);

- *Description*

The first function gets the stream's **streambuf** object.
The second function sets the stream's **streambuf** object to `sb`.

- *Return Value*

The first function returns a pointer to the **streambuf** object that is associated with this stream.
The second function returns a pointer to the old **streambuf** object that is associated with this stream.

- **Example**

```
#include <iostream>
void main()
{
        int i;
 i=cin.rdbuf()->sgetc();
cout<<i;
    }
```

ios::copyfmt

- *Syntax:*

#include <ios>
ios& copyfmt(const ios& `rhs`**);**

- *Description*

The member function reports the **callback event** `erase event`. It then copies the **fill character**, the **tie pointer**, and the **formatting information** from `rhs` into `*this`. Before altering the **exception mask**, it reports the callback event `copyfmt event`. If, after the copy is complete, `state & exceptions()` is nonzero, the function effectively calls `clear` with the argument `rdstate()`.

- *Return Value*

Returns `*this`.

ios::fill

- *Syntax:*

#include <ios>
char fill() const ;
char fill(char `ch`**);**

- *Description*

The first function reads the stream's fill character.
The second function sets the stream's internal fill character variable to *ch.*

- *Return Value*

The first function returns the stream's fill character.
The second function returns the previous value.

♦ **Example**

```
#include <iostream>
void main()
{
  cout.width(10);
  cout.fill('*');
  cout<<10.123<<endl;
    }
```

ios::init

♦ *Syntax:*

#include <ios>
void init(streambuf* `sb`**);**

♦ *Description*

The member function stores values in all member objects, so that:

- `rdbuf ()` returns `sb`
- `tie ()` returns a null pointer
- `rdstate ()` returns `goodbit` if `sb` is nonzero; otherwise, it returns `badbit`
- `exceptions ()` returns `goodbit`
- `flags ()` returns `skipws| dec`
- `width ()` returns zero
- `precision ()` returns 6
- `fill ()` returns the **space character** (0x09 - 0x0D or 0x20)
- D:\Manuals\Cpp-8051\IOS_ios_baseCCiword.htm - ios_base::iword`iword` returns zero and `pword` returns a null pointer for all argument values

boolalpha

♦ *Syntax:*

#include <ios>
ios_base& boolalpha (ios_base& str);

♦ *Description*

The manipulator effectively calls `str.setf (ios_base::boolalpha).`

♦ *Return Value*

Returns `str`.

♦ **Example**

```
#include <iostream>
```

```
void main()
{
  bool b=true;
  boolalpha(cout);
  cout<<b<<endl;
     }
```

noboolalpha

- *Syntax:*

#include <ios>
ios_base& noboolalpha(ios_base& str);

- *Description*

The manipulator effectively calls `str.unsetf (ios_base::boolalpha).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  bool b=true;
  noboolalpha(cout);
  cout<<b<<endl;
     }
```

showbase

- *Syntax:*

#include <ios>
ios_base& showbase (ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::showbase).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  int n=0x2d;
  showbase(cout);
  cout<<n<<endl;
     }
```

noshowbase

- *Syntax:*

#include <ios>
ios_base& noshowbase (ios_base& str);

- *Description*

The manipulator effectively calls `str.unsetf (ios_base::showbase)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  int n=0x2d;
        noshowbase(cout);
        cout<<n<<endl;
      }
```

showpoint

- *Syntax:*

#include <ios>
ios_base& showpoint (ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::showpoint)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  float n=1.00;
  showpoint(cout);
  cout<<n<<endl;
      }
```

noshowpoint

- *Syntax:*

#include <ios>
ios_base& noshowpoint (ios_base& str);

- *Description*

The manipulator effectively calls `str.unsetf (ios_base::showpoint)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
          float n=1.00;
          noshowpoint(cout);
          cout<<n<<endl;
      }
```

showpos

- *Syntax:*

#include <ios>
ios_base& showpos (ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::showpos).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  float n=1.00;
        showpos(cout);
        cout<<n<<endl;
      }
```

noshowpos

- *Syntax:*

#include <ios>
ios_base& noshowpos (ios_base& str);

- *Description*

The manipulator effectively calls `str.unsetf (ios_base::showpos).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  float n=1.00;
```

```
    noshowpos(cout);
    cout<<n<<endl;
}
```

skipws

- *Syntax:*

#include <ios>
ios_base& skipws(ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::skipws).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
        char  s[10];
        skipws(cin);
        cin>>s;
        cout<<s<<endl;
    }
```

noskipws

- *Syntax:*

#include <ios>
ios_base& noskipws(ios_base& str);

- *Description*

The manipulator effectively calls `str.unsetf (ios_base::skipws).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
        char  s[10];
        noskipws(cin);
        cin>>s;
        cout<<s<<endl;
    }
```

uppercase

♦ *Syntax:*

#include <ios>
ios_base& uppercase(ios_base& str);

♦ *Description*

The manipulator effectively calls `str.setf (ios_base::uppercase).`

♦ *Return Value*

Returns `str`.

♦ **Example**

```
#include <iostream>
void main()
{
  cout.setf(ios::scientific);
  uppercase(cout);
  cout<<100.12<<endl;
    }
```

nouppercase

♦ *Syntax:*

#include <ios>
ios_base& nouppercase(ios_base& str);

♦ *Description*

The manipulator effectively calls `str.unsetf (ios_base::uppercase).`

♦ *Return Value*

Returns `str`.

♦ **Example**

```
#include <iostream>
void main()
{
  cout.setf(ios::scientific);
  uppercase(cout);
  cout<<100.12<<endl;
  nouppercase(cout);
  cout<<100.12<<endl;
    }
```

internal

♦ *Syntax:*

#include <ios>
ios_base& internal(ios_base& str);

♦ *Description*

The manipulator effectively calls `str.setf (ios_base::internal, ios_base::adjustfield)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
        cout.width(10);
       cout.fill('#');
       internal(cout);
       cout<<100.12<<endl;
    }
```

left

- *Syntax:*

#include <ios>
ios_base& left(ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::left, ios_base::adjustfield)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
 cout.width(10);
       cout.fill('#');
       left(cout);
       cout<<100.12<<endl;
    }
```

right

- *Syntax:*

#include <ios>
ios_base& right(ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::right, ios_base::adjustfield)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  cout.width(10);
  cout.fill('#');
  right(cout);
  cout<<100.12<<endl;
    }
```

dec

- *Syntax:*

#include <ios>
ios_base& dec(ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::dec, ios_base::basefield)`.

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
  dec(cout);
  cout<<100<<endl;
  hex(cout);
  cout<<100<<endl;
  oct(cout);
  cout<<100<<endl;
    }
```

hex

- *Syntax:*

#include <ios>
ios_base& hex(ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base::hex, ios_base::basefield)`.

- *Return Value*

Returns `str`.

♦ **Example**

```
#include <iostream>
void main()
{
     dec(cout);
         cout<<100<<endl;
         hex(cout);
         cout<<100<<endl;
         oct(cout);
         cout<<100<<endl;
      }
```

oct

♦ *Syntax:*

#include <ios>
ios_base& oct(ios_base& str);

♦ *Description*

The manipulator effectively calls `str.setf (ios_base::oct, ios_base::basefield)`.

♦ *Return Value*

Returns `str`.

♦ **Example**

```
#include <iostream>
void main()
{
  dec(cout);
  cout<<100<<endl;
  hex(cout);
  cout<<100<<endl;
  oct(cout);
  cout<<100<<endl;
     }
```

fixed

♦ *Syntax:*

#include <ios>
ios_base& fixed(ios_base& str);

♦ *Description*

The manipulator effectively calls `str.setf (ios_base::fixed, ios_base::floatfield)`.

♦ *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
          fixed(cout);
  cout<<1.235<<endl;
    }
```

scientific

- *Syntax:*

#include <ios>
ios_base& scientific(ios_base& str);

- *Description*

The manipulator effectively calls `str.setf (ios_base:: scientific, ios_base::floatfield).`

- *Return Value*

Returns `str`.

- **Example**

```
#include <iostream>
void main()
{
        scientific(cout);
        cout<<1.235<<endl;
    }
```

<istream>

class istream : public ios {

public:

```
        bool  ipfx(bool _Noskip=false);
        void isfx();
        istream(stdiobuf* sb);
        virtual ~istream() ;
        class sentry;
```

```
        istream& operator>> (istream& (*pf)(istream&));
        istream& operator>> (ios& (*pf)(ios&));
        istream& operator>> (ios_base& (*pf)(ios_base&));
        istream& operator>>(bool& n);
        istream& operator>>(short& n);
        istream& operator>>(unsigned short& n);
        istream& operator>>(int& n);
        istream& operator>>(unsigned int& n);
        istream& operator>>(long& n);
        istream& operator>>(unsigned long& n);
        istream& operator>>(float& f);
        istream& operator>>(double& f);
        istream& operator>>(long double& f);
        istream& operator>>(void*& p);
        istream& operator>>(stdiobuf* sb);
        streamsize gcount() const ;
        int get();
        istream& get(char& c);
        istream& get(char* s, streamsize n);
        istream& get(char* s, streamsize n, char delim);
        istream& get(stdiobuf& sb);
        istream& get(stdiobuf& sb, char delim);
        istream& getline(char* s, streamsize n);
        istream& getline(char* s, streamsize n, char delim);
        istream& ignore(streamsize n = 1, int delim = EOF);
        int peek();
        istream& read (char* s, streamsize n);
        streamsize readsome(char* s, streamsize n);
        istream& putback(char c);
        istream& unget();
        int sync() ;
        void eatwhite();
        protected:
                istream& operator=(stdiobuf* _isb);
        private:
                streamsize x_gcount;
                int _fGline;
};
```

class istream::sentry {

```
        bool _Ok;
        public:
```

```
        sentry(istream& is, bool noskipws = false);
        ~sentry();
        operator bool();
};

// character extraction templates:

istream& operator>>(istream&, char&);
istream& operator>>(istream&, unsigned char&);
istream& operator>>(istream&, signed char&);

istream& operator>>(istream&, char*);
istream& operator>>(istream&, unsigned char*);
istream& operator>>(istream&, signed char*);


inline istream& ws(istream &is);
```

This header defines **istream** class, a manipulator and several character extraction functions (**formatted input functions**).

The **istream** class provides the basic capability for sequential and random-access input. An **istream** object has a **streambuf**-derived object attached, and the two classes work together; the **istream** class does the formatting, and the **streambuf** class does the low-level buffered input.

istream::ipfx

- *Syntax:*

#include <istream>
bool ipfx(bool `_Noskip`**=false);**

- *Description*

Check for error conditions prior to extraction operations (input prefix function).

- *Return Value*

A nonzero return value if the operation was successful; 0 if the stream's error state is nonzero, in which case the function does nothing.

istream::isfx

- *Syntax:*

#include <istream>
void isfx();

♦ *Description*

This input suffix function is called at the end of every extraction operation (input suffix function).

istream::istream

♦ *Syntax:*

#include <istream>
istream(stdiobuf* `sb`**);**

♦ *Description*

Constructs an **istream** object attached to an existing object of a **streambuf**-derived class.

istream::~istream

♦ *Syntax:*

#include <istream>
virtual ~istream() ;

♦ *Description*

Destroys an **istream** object.

istream::operator>>

♦ *Syntax:*

#include <istream>
istream::operator>>
istream& operator>> (istream& (*pf)(istream&));
istream& operator>> (ios& (*pf)(ios&));
istream& operator>> (ios_base& (*pf)(ios_base&));
istream& operator>>(bool& n);
istream& operator>>(short& n);
istream& operator>>(unsigned short& n);
istream& operator>>(int& n);
istream& operator>>(unsigned int& n);
istream& operator>>(long& n);
istream& operator>>(unsigned long& n);
istream& operator>>(float& f);
istream& operator>>(double& f);
istream& operator>>(long double& f);
istream& operator>>(void*& p);
istream& operator>>(stdiobuf* sb);

♦ *Description*

These overloaded operators extract their argument from the stream.

- *RETURN VALUE*

Returns the current istream object (*this);

- **Example**

```
#include <iostream>
void main()
{
 bool b;
 float f;
 long n;
 cin>>b>>f>>n;
 cout << b<<"  "<<f<<"  "<<n;
     }
```

istream::gcount

- *Syntax:*

#include <istream>
streamsize gcount() const ;

- *Description*

Counts the characters extracted in the last unformatted operation.

- *RETURN VALUE*

Returns the number of characters extracted by the last unformatted input function.

- **Example**

```
#include <iostream>
void main()
{
 char *name;
 int buf_size = 100;
 int count = 0;          // Character counter.
 name = new char[buf_size];
// Notice that the output buffer is flushed.
       cout << "\n Enter your name:" << endl;
      cin.getline(name, buf_size);
      count = cin.gcount();
  // Since getline() retains the linefeed, gcount()
  // will count it as input.
      cout << "\nName character count: " << count - 1;
    }
```

istream::get

- *Syntax:*

#include <istream>
int get();

- *Description*

Extracts a single character from the stream and returns it.

- **Return Value**

Returns the extracted character.

- **Example**

```
#include <iostream>
void main()
{
 char ch;
 ch=cin.get ();
 cout << ch;
     }
```

istream::get

- *Syntax:*

#include <istream>
istream& get(char& `c`**);**

- *Description*

Extracts a single character from the stream and stores it as specified by the reference argument.

- *RETURN VALUE*

Returns the current istream object (*this);

- **Example**

```
#include <iostream>
void main()
{
 char ch;
 cin.get (ch);
 cout << ch;
     }
```

istream::get

- *Syntax:*

#include <istream>
istream& get(char* `s`**, streamsize** `n`**);**

- *Description*

Effectively calls `get(s, n, '\n').`

- *RETURN VALUE*

Returns the current istream object (*this);

♦ **Example**

```
#include <iostream>
void main()
{
 char s[10];
 cin.get (s,5);
 cout << s;
     }
```

istream::get

♦ *Syntax:*

#include <istream>
istream& get(char* `s`**, streamsize** `n`**, char** `delim`**);**

♦ *Description*

Extracts characters from the stream until either *delim* is found, the limit *n* is reached, or the end of file is reached. The characters are stored in the array followed by a null terminator.

♦ *RETURN VALUE*

Returns the current istream object (*this);

♦ **Example**

```
#include <iostream>
void main()
{
 char s[10];
 cin.get (s,5,'\n');
 cout << s;
     }
```

istream::get

♦ *Syntax:*

#include <istream>
istream& get(stdiobuf& `sb`**);**

♦ *Description*

Effectively calls `get(sb,'\n').`

♦ *RETURN VALUE*

Returns the current istream object (*this);

istream::get

♦ *Syntax:*

#include <istream>

istream& get(stdiobuf& sb, char delim);

- *Description*

Gets characters from the stream and stores them in a **streambuf** object until the delimiter is found or the end of the file is reached. The **ios::failbit** flag is set if the **streambuf** output operation fails.

- *RETURN VALUE*

Returns the current istream object (*this);

istream::getline

- *Syntax:*

#include <istream>
istream& getline(char* s, streamsize n);

- *Description*

EFFECTIVELY CALLS GETLINE(s, N, '\N').

- *RETURN VALUE*

Returns the current istream object (*this);

- **Example**

See the **gcount** example.

istream::getline

- *Syntax:*

#include <istream>
istream& getline(char* s, streamsize n, char delim);

- *Description*

Extracts characters from the stream until either the delimiter *delim* is found, the limit *n*-1 is reached, or end of file is reached. The characters are stored in the specified array followed by a null terminator. If the delimiter is found, it is extracted but not stored.

- *RETURN VALUE*

Returns the current istream object (*this);

- **Example**

See the **gcount** example.

istream::ignore

- *Syntax:*

#include <istream>
istream& ignore(streamsize n = 1, int delim = EOF);

- *Description*

Extracts and discards up to *n* characters. Extraction stops if the delimiter *delim* is extracted or the end of file is reached. If *delim* = **EOF** (the default), then only the end of file condition causes termination. The delimiter character is extracted.

- *RETURN VALUE*

Returns the current istream object (*this);

- **Example**

```
#include <iostream>
void main()
{
  char s[10];
  cin.ignore(2);
  cin.get (s,5,'\n');
  cout << s;
      }
```

istream::peek

- *Syntax:*

#include <istream>
int peek();

- *Description*

Returns the next character without extracting it from the stream. Returns **EOF** if the stream is at end of file or if the **ipfx** function indicates an error.

- **Example**

```
#include <iostream>
void main()
{
  char ch;
  cin.peek();
  cout << ch;
      }
```

istream::read

- *Syntax:*

#include <istream>
istream& read (char* `s`**, streamsize** `n`**);**

- *Description*

Extracts bytes from the stream until the limit *n* is reached or until the end of file is reached. The bytes are stored in the array beginning at `s` **.** The **read** function is useful for binary stream input.

- *RETURN VALUE*

Returns the current istream object (*this);

♦ **Example**

```
#include <iostream>
void main()
{
 char s[10];
 cin.read(s,5);
 cout << s;
     }
```

istream::readsome

♦ *Syntax:*

#include <istream>
streamsize readsome(char* `s`**, streamsize** `n`**);**

♦ *Description*

The member function extracts up to `n` elements and stores them in the array beginning at `s`.

♦ **Example**

```
#include <iostream>
void main()
{
 char s[10];
 cin.readsome(s,5);
 cout << s;
     }
```

istream::putback

♦ *Syntax:*

#include <istream>
istream& putback(char `c`**);**

♦ *Description*

Puts a character back into the input stream. The **putback** function may fail and set the error state. If *c* does not match the character that was previously extracted, the result is undefined.

♦ *RETURN VALUE*

Returns the current istream object (*this);

istream::ungetc

♦ *Syntax:*

#include <istream>

istream& unget();

- *Description*

Puts back the previous element in the stream, if possible , as if by calling `rdbuf()->sungetc()`. If `rdbuf()` is a null pointer, or if the call to `sungetc` returns **EOF**, the function calls `setstate(badbit.`

- *RETURN VALUE*

Returns the current istream object (*this);

istream::sync

- *Syntax:*

#include <istream>
int sync() ;

- *Description*

Synchronizes the stream buffer with the external source of characters.

- *Return Value*

EOF to indicate errors.

istream::eatwhite

- *Syntax:*

#include <istream>
void eatwhite();

- *Description*

Extracts leading **white space** (0x09 - 0x0D or 0x20) from the stream by advancing the get pointer past spaces and tabs.

istream::operator=

- *Syntax:*

#include <istream>
istream& operator=(stdiobuf* _isb);

- *Description*

The operator initializes **ios** and **istream** members and sets the **`streambuf object`** to `_isb`.

operator>>

- *Syntax:*

#include <istream>
istream& operator>>(istream& `is`, char& `c`);
istream& operator>>(istream& `is`, unsigned char& `c`);

istream& operator>>(istream& is, signed char& c);
istream& operator>>(istream& is, char* s);
istream& operator>>(istream& is, unsigned char* s);
istream& operator>>(istream& is, signed char* s);

- *Description*

The first function extracts an element, if possible, and stores it in `c`. Otherwise, it calls `is.setstate(failbit)`.

The second function effectively calls `is>>(char&)c`.

The third function effectively calls `is>>(char&)c`.

The fourth function extracts up to `n-1` elements and stores them in the array beginning at `s`. If `is.width()` is greater than zero, `n` is `is.width()`; otherwise it is the largest array of characters that can be declared. The function always stores `char(0)` after any extracted elements it stores. Extraction stops early on end-of-file or on any element (which is not extracted) that would be discarded by `ws`. If the function extracts no elements, it calls `is.setstate(failbit)`. In any case, it calls `is.width(0)`.

The fifth function effectively calls `is>>(char *)s`.

The sixth function effectively calls `is>>(char *)s`.

- *RETURN VALUE*

Returns `is`.

- **Example**

```
#include <iostream>
void main()
{
 char s[10],ch;
 cin>>s>>ch;
 cout << s<<"  "<<ch;
     }
```

ws

- *Syntax:*

#include <istream>
istream& ws(istream& is);

- *Description*

Extracts leading **white space** from the stream by calling the **eatwhite** function.

- *RETURN VALUE*

Returns `is`.

- **Example**

```
#include <iostream>
void main()
```

```
{
 char s[10];
 ws(cin);
 cin.read(s,3);
 cout << s;
     }
```

<ostream>

```
class ostream : public ios {
public:
        ostream(stdiobuf*);
        virtual ~ostream();
class sentry;
        int  opfx();
        void osfx();
        ostream& operator<<(ostream& (* _f)(ostream&));
        ostream& operator<<(ios& (* _f)(ios&));
        ostream& operator<< (ios_base& (*pf)(ios_base&));
        ostream& operator<<(bool n);
        ostream& operator<<(short);
        ostream& operator<<(unsigned short);
        ostream& operator<<(int);
        ostream& operator<<(unsigned int);
        ostream& operator<<(long);
        ostream& operator<<(unsigned long);
        ostream& operator<<(float);
        ostream& operator<<(double);
        ostream& operator<<(long double);
        ostream& operator<<(const void *);
        ostream& operator<<(stdiobuf*);
        ostream& put(char);
        ostream& put(unsigned char);
        ostream& put(signed char);
        ostream& write(const char *,int);
        ostream& write(const unsigned char *,int);
        ostream& write(const signed char *,int);
        ostream& flush();
};
```

class ostream::sentry {

```
        bool _Ok;
        ostream _Ostr;
public:
        sentry(ostream& _Os);
        ~sentry();
        operator bool();
};

ostream& operator<<(ostream&, char);
ostream& operator<<(ostream&, signed char);
ostream& operator<<(ostream&, unsigned char);
ostream& operator<<(ostream&, const char*);
ostream& operator<<(ostream&, const signed char*);
ostream& operator<<(ostream&, const unsigned char*);
ostream& flush(ostream& os);
ostream& endl(ostream& os);
ostream& ends(ostream& os);
```

This header defines o**stream** class, several related manipulators and character insertion functions (**formatted output functions**).
The **ostream** class provides the basic capability for sequential and random-access output. An **ostream** object has a **streambuf**-derived object attached, and the two classes work together; the **ostream** class does the formatting, and the **streambuf** class does the low-level buffered output.

ostream::ostream

- *Syntax:*

#include <ostream>
ostream(stdiobuf*);

- *Description*

Constructs an **ostream** object that is attached to an existing **streambuf** object.

ostream::~ostream

- *Syntax:*

#include <ostream>
virtual ~ostream();

- *Description*

  Destroys an **ostream** object. The output buffer is flushed as appropriate. The attached **streambuf** object is destroyed only if it was allocated internally within the **ostream** constructor.

ostream::opfx

- *Syntax:*

#include <ostream>
int opfx();

- *Description*

Output prefix function, called prior to insertion operations to check for error conditions, and so forth.

- *Return Value*

If the **ostream** object's error state is not 0, **opfx** returns 0 immediately; otherwise it returns a nonzero value.

ostream::osfx

- *Syntax:*

#include <ostream>
void osfx();

- *Description*

Output suffix function, called after insertion operations; flushes the stream's buffer if it is unit buffered.

ostream::operator<<

- *Syntax:*

#include <ostream>
ostream& operator<<(ostream& (* _f)(ostream&));
ostream& operator<<(ios& (* _f)(ios&));
ostream& operator<< (ios_base& (*pf)(ios_base&));
ostream& operator<<(bool n);
ostream& operator<<(short);
ostream& operator<<(unsigned short);
ostream& operator<<(int);
ostream& operator<<(unsigned int);
ostream& operator<<(long);
ostream& operator<<(unsigned long);
ostream& operator<<(float);
ostream& operator<<(double);
ostream& operator<<(long double);
ostream& operator<<(const void *);
ostream& operator<<(stdiobuf*);

- *Description*

These overloaded operators insert their argument into the stream.

- *RETURN VALUE*

Returns the current output stream `(*this)`.

- **Example**

```
#include <iostream>
void main()
{
  bool b=true;
  short i=30;
  float f=0.78
  cout << b<<"  "<<i<<"  "<<f;
      }
```

ostream::put

- *Syntax:*

#include <ostream>
ostream& put(char c);
ostream& put(unsigned char c);
ostream& put(signed char c);

- *Description*

Inserts a single character into the output stream.

- *RETURN VALUE*

Returns the current output stream `(*this)`.

- **Example**

```
#include <iostream>
void main()
{
  char ch='A';
  cout .put(ch);
      }
```

ostream::write

- *Syntax:*

#include <ostream>
ostream& write(const char *s,int n);
ostream& write(const unsigned char *s,int n);
ostream& write(const signed char *s,int n);

- *Description*

Inserts a specified number of bytes from a buffer into the stream. If the underlying file was opened in text mode, additional carriage return characters may be inserted. The **write** function is useful for binary stream output.

- *RETURN VALUE*

Returns the current output stream `(*this)`.

- **Example**

```
#include <iostream>
void main()
{
  char *s="This is an example";
  cout .write(s,5);
}
```

ostream::flush

- *Syntax:*

#include <ostream>
ostream& flush();

- *Description*

Flushes the buffer associated with this stream.

- *RETURN VALUE*

Returns the current output stream `(*this)`.

operator<<

- *Syntax:*

#include <ostream>
ostream& operator<<(ostream& os, char c);
ostream& operator<<(ostream& os, signed char c);
ostream& operator<<(ostream& os, unsigned char c);
ostream& operator<<(ostream& os, const char* s);
ostream& operator<<(ostream& os, const signed char* s);
ostream& operator<<(ostream& os, const unsigned char* s);

- *Description*

The first function inserts the character **`c into the output stream os`**.
The second function effectively calls `os << (char)c`.
The third function effectively calls `os << (char)c`.
The fourth function determines the length `n` of the sequence beginning at `s`, and inserts the sequence. If `n < os.width()`, then the function also inserts a repetition of `width() - n` **`fill characters`**. The repetition precedes the sequence if `(os.flags() & adjustfield!=left`. Otherwise, the repetition follows the sequence.
The fifth function effectively calls `os << (const char *)s`.
The sixth function effectively calls `os << (const char *)s`.

- *RETURN VALUE*

The function returns `os`.

- **Example**

```
#include <iostream>
void main()
{
  char *s="This is an example";
  char ch='x';
  cout<<s;
       cout<<ch;
     }
```

flush

- *Syntax:*

#include <ostream>
ostream& flush(ostream& os);

- *Description*

This manipulator, when inserted into an output stream, flushes the buffer.

- *Return Value*

Returns the output stream **os**.

endl

- *Syntax:*

#include <ostream>
ostream& endl(ostream& os);

- *Description*

This manipulator, when inserted into an output stream, inserts a newline character and then flushes the buffer.

- *Return Value*

Returns the output stream **os**.

- **Example**

```
#include <iostream>
void main()
{
  cout<<"This is a sample!";
  endl(cout);
     }
```

ends

- *Syntax:*

#include <ostream>
ostream& ends(ostream& os);

- *Description*

This manipulator, when inserted into an output stream, inserts a null-terminator character.

- *Return Value*

Returns the output stream **os**.

- **Example**

```
#include <iostream>
void main()
{
  cout<<"This is a sample!";
  ends(cout);
     }
```

<iostream>

extern class istream cin;
extern class ostream cout;
extern class ostream cerr, clog;

Declare objects that control reading from and writing to the standard streams: **cin, cout, cerr, clog**.

<iomanip>

```
T1 resetiosflags(ios_base::fmtflags mask);
T2 setiosflags(ios_base::fmtflags mask);
T3 setbase(int base);
T4 setfill(char c);
T5 setprecision(int n);
T6 setw(int n);
```

Defines several **manipulators** that each take a single argument. Each of these manipulators returns an unspecified type, called `T1` through `T6` here, that overloads both `istream::operator>>` and `ostream::operator<<`.

resetiosflags

T1 resetiosflags(ios_base::fmtflags **mask**);

The manipulator returns an object that, when extracted from or inserted into the stream str, calls str.setf(ios_base::fmtflags(), mask), then returns str.

Example

```
#include <iostream>
#include <iomanip>
void main()
{
  cout<<resetiosflags(ios::showbase);
  cout<<123<<"  "<<hex<<123;
}
```

setiosflags

T2 setiosflags(ios_base::fmtflags **mask**);

The manipulator returns an object that, when extracted from or inserted into the stream str, calls str.setf(mask), then returns str.

Example

```
#include <iostream>
#include <iomanip>
void main()
{
  cout<<setiosflags(ios::showbase);
  cout<<setiosflags(ios::showpos);
  cout<<123<<"  "<<hex<<123;
}
```

setbase

T3 setbase(int **base**);

The manipulator returns an object that, when extracted from or inserted into the stream str, calls str.setf(mask, ios_base::basefield), then returns str. Here, mask is determined as follows:

- If base is 8, then mask is ios_base::oct.
- If base is 10, then mask is ios_base::dec.
- If base is 16, then mask is ios_base::hex.
- If base is any other value, then mask is ios_base::fmtflags(0).

Example

```
#include <iostream>
```

```
#include <iomanip>
void main()
{
  cout<<"123"<<"  "<<setbase(16)<<123;
}
```

T4 setfill(char **fillch**);

The template manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.fill(`**`fillch`**`)`, then returns `str`.

Example

```
#include <iostream>
#include <iomanip>
void main()
{
  cout<<setfill('*')<<setw(10)<<123.5;
}
```

T5 setprecision(int **prec**);

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.precision(prec)`, then returns `str`.

Example

```
#include <iostream>
#include <iomanip>
void main()
{
  cout<<setprecision(8)<<123.5345786;
}
```

T6 setw(int **wide**);

The manipulator returns an object that, when extracted from or inserted into the stream `str`, calls `str.width(`**`wide`**`)`, then returns `str`.

Example

```
#include <iostream>
#include <iomanip>
void main()
{
  cout<< setw(10) <<setfill('*')<<123.5; cout<<setprecision(8)<<123.5345786
}
```

<new>

void *operator new(size_t size);
void operator delete(void *ptr);
void *operator new[] (size_t size);
void operator delete[](void * ptr);
void *operator new(size_t size,void * ptr);
void *operator new[](size_t size,void * ptr);
void operator delete(void * ptr,void *);
void operator delete[](void * ptr,void *);

void *operator new(size_t size);
void *operator new[] (size_t size);

- *Description*

Attempts to allocate memory for an object or for an array of objects of type **size_t .**

- *Return Value*

A pointer to the allocated memory or NULL if it fails.

- **Example**

```
#include <new>
void main()
{
  int *p;
  p=new int;
  *p=17;
  int *n= new int[3];
  for (int i=0;i<3;i++)
    *(n+i)=10+i;
}
```

void operator delete(void *ptr);
void operator delete[](void * ptr);

- *Description*

The **delete** keyword deallocates a block of memory. The argument *ptr* must point to a block of memory previously allocated by the **new** operator. If *ptr* points to an array, place empty brackets before *pointer*.

- **Example**

```
#include <new>
```

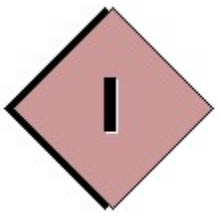
```
void main()
{
  int *p;
  p=new int;
  *p=17;
  if (!p)
  {  cout<<"Allocation error!";exit(1);}
  int *n= new int[3];
  if (!n)
  {  cout<<"Allocation error!";exit(1);}
  for (int i=0;i<3;i++)
    *(n+i)=10+i;
  delete p;
  delete []n;
}
```

void *operator new(size_t `size`,void * `ptr`);
void *operator new[](size_t `size`,void * `ptr`);

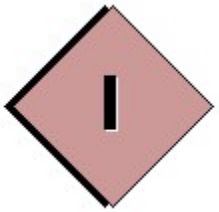
- *Description*

Returns **`ptr`.**

INDEX



INDEX



A

Abs, 6-9
Arg, 6-10
Auto, 4-3

B

Bool, 4-3
Break, 4-4

C

Case, 4-6
Char, 4-6
Class, 4-7
Conj, 6-10
Const, 4-12
Continue, 4-14
Cos, 6-10
Cosh, 6-11

D

Default, 4-15
Delete, 4-15
Directories, 2-1
Do, 4-19
Double, 4-20, 6-2

E

Else, 4-20
Enum, 4-21
Errors, 5-3
Exception, 1-2
Exp, 6-11
Extern, 4-22

F

False, 4-24
Float, 4-24,
For, 4-25
Friend, 4-27

G

Goto, 4-29

H

Handlaing, 1-2

I

If, 4-31
Imag, 6-11
Inline, 4-32, 5-68
Installation, 2-1
Int, 4-33
Ios, 6-63
Istream, 6-86
Iostream, 6-102
Iomanip, 6-103

K

Keywords, 4-1

L

Library, 1-4, 3-1, 6-1
Log, 6-11
Log10, 6-12
Long, 4-34

M

Mutable, 1-2
Multiple Inheritance, 1-4

N

Namespace, 1-3
New, 4-34, 6-105
Norm, 6-12

O

Operator, 4-35, 6-15 to 6-19
ostream, 6-97

P

Polar, 6-13
Pow, 6-13
Private, 4-38
Projects, 3-1
Protected, 4-41
Public, 4-44

R

Real, 6-14
Register, 4-45
Return, 4-46
RTX51, 4-63
Runtime, 1-3

S

Short, 4-47
Signed, 4-47
Sin, 6-14
Sinh, 6-14
Sizeof, 4-48
Software, P-1
Sqrt, 6-15
Static, 4-49
Stdiobuf, 6-46
Streambuf, 6-48
String, 6-20
Struct, 4-50
Swap, 6-2, 6-45
Switch, 4-52

T

Template, 1-3
This, 4-54
True, 4-56
Typedef, 4-56

U

Union, 4-57
Unsigned, 4-59

V

Virtual, 4-59
Void, 4-61
Volatile, 4-61

W

Warnings, 5-1
While, 4-62