Fundamental types.

1. bool
2. char
3. int
4. double
5. void
6. enum
7. struct , Unions & Classes

1.**bool**

A boolean can have one of the two values true or false. True as a value 1 when converted to integer and false a value 0.

```
bool  c = a == b;
                    c .eq. true if a .eq.b
                    c .eq. false if a .neq.b
bool  c = 7;
                    c .eq. 1
bool  c = a + b;
                    c .eq. true if a+b .neq. 0
                    c .eq. false if a + b .eq. 0
int   d = true;
```

2. **char**

Universally a char a 8 bits so that it can hold one of 256 different values.The 256 values represented by an 8 bit can be interpreted as the values 0 to 255 or as values -127 to 128.

There are three character types.

a. char (plain)
b. signed char (-127 to 128)
c. unsigned char (0 to 255)

```
unsigned char c = '¦';        // asci(221)
cout<<int(c)<<endl;           // 166


signed char d = '¦';
cout<<int(d)<<endl;           //-90
```

Its implementation defined whether a plain char is considered sigend or unsigned.Most implementation

defined features relate to differences in hardware used to run a program. This may lead to some nasty surprises.

```
char c = 255;  int i  = c;
```

On a machine where a char unsigned i = 255. On a machine where a char signed i = -1.

The three char types are distinct, You can't mix pointers to different char types.

```
char c;
signed char sc;
unsigned char uc;

char* pc = &us;    // error no pointer
conversion
signed char* psc = pc;   //error
unsigned char* pus = pc; // error
```

Variables of the these three char types can be freely assigned to each other. However, assigning a too large value to a signed char is undefined.

```
uc = c;  // ok

uc = sc; // ok

c = sc;   // ok;

c = uc;  // undefined if c is signed & uc is val
large

c = 255; // is plain chars are signed

sc = uc; // undefined if uc value is large

sc = c; // undefined if c is unsigend & value is
large
```

Trigraph

The ASCII special characters [ ] { } | \ occupy character set positions designated as alphabetic by ISO. In most European national ISO-646 character sets, these positions are occupied by letters not found in english alphabet.

A set of trigraphs is provided to allow national characters to be expressed in a portable way using truly standard minimal character set.

[	??(
]	??)
{	??<
}	??>
|	??!
\	??\
#	??=
~	??-
^	??`

```
if( i == 50 ??!??! i == 100)
??<
          cout<<"hello"<<endl;
??>
```

Technicalities

a. Some standard library functions such as strcmp take plain chars only.

b. Charater types are integral types. Each character constant has an interger value associated with it.So arithmatic and logic operations apply.

c. A type wchar_t is provided to hold characte of a large character set such as unicode.The size of wchar_t is implementatio defined and large enough to hold the largest character set supported by the implemenation's locale.

```
#include<stdlib.h>

wchar_t c;
```

Character Literals

A character literal often called character constant is a character enclosed in a single quotes.

```
char a = 'Z';
```

Wide character literals are of the form

```
wchar_t  a = L'ZA';
```

A few characters have a standard names that use a backslash \ as an escape character.

Name	
Newline	\n
Horizontal Tab	\t
Vertical tab	\v
Backspace	\b
Carrige return	\r
Octal	\ooo
Hex number	\xhhh

3. int

There are three integer types.

a. int (plain)
b. signed int
c. unsigned int

In addition There are three Sizes.

a. int
b. short int
c. long int

Unlike plain chars plain ints are always signed.

```
short signed int  a;
short unsigned int a;
long  signed int a;
long  unsigned int a;
```

Integer literals.

```
unsigned int a = 8U;
unsigned int a = 8u;
long int  a = 999999999L;
long int  a = 999999999l;
short int c = 9999;
```

If no suffix is provided the compiler gives an integer literal a suitable type based on its value.

4. Floating point types.

a. float (single precision)
b. double (double precesion)
c. long double (extended precision)

Floating point literals.

By default a floating point literal is of type double.

```
double d = 1.23
double d = 1.23e-15
```

floationg point literals of type float should be prefixed with F or F.

```
float f = 3.141f;
float f = 2.0F;
```

5. void

```
void x;   // error no void objects
void f(); // function does not return a value
void* p;  // pointer to a object of unknown type
```

6. Enumerations.

An enumeration is a type that can hold a set of values specified by the user.

```
Enum keyword{ASM,AUTO,BREAK};
Keyword key;

Switch(key)
{
     case ASM:

     case AUTO:
}
```

7. struct , Unions & Classes

In C, a **struct** is an agglomeration of data, a way to package data so you can treat it in a clump.

In C++ **class** keyword is identical to the **struct** keyword in absolutely every way except one: **class** defaults to **private**, whereas **struct** defaults to **public**. Here are two structures that produce the same result:

```
#include<iostream.h>

struct CA
{
      private:
            int data;
      public:
            void fun();
};

void CA::fun()
{
      data = 200;
      cout<<data<<endl;
```

```
}

// Identical results are produced with:

class CB
{
        int data;
    public:
        void fun();
};

int CB::fun()
{
    data = 200;
    cout<<data<<endl;
}

void main()
{
    CA a;
    CB b;
    a.fun();
    b.fun();
}
```

A **union** can also have a constructor, destructor, member functions, and even access control.

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

union U {
private: // Access control too!
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
```

```
};

U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
} ///:~
```

You might think from the code above that the only difference between a **union** and a **class** is the way the data is stored (that is, the **int** and **float** are overlaid on the same piece of storage). However, a **union** cannot be used as a base class during inheritance, which is quite limiting from an object-oriented design standpoint

Name space

What is a namespace?

A namespace is a declarative region that can be used to package names,improve program readability, and reduce name clashes in the global namespace.

At the most basic level, the syntax for namespace is shown.

```
namespace  Test
{
     int i = 100;
     int j = 200;

     class CA  <----------------  class Test::CA
     {
     };

     class CB  <----------------  class Test::CB
     {
     };

     void fun()  <----------------  Test::fun()
```

```
    {
    }
};
```

A namespace definition can appear only at global scope, or nested within another namespace. No terminating semicolon is necessary after the closing brace of a namespace definition. Namespace facilitate building large systems by partioning names into logical groupings.

Using Namespace

You can refer to a name within a namespace in three ways: by specifying the name using the scope resolution operator, with a **using** directive to introduce all names in the namespace, or with a **using** declaration to introduce names one at a time.

1.Scope resolution

Any name in a namespace can be explicitly specified using the scope resolution operator in the same way that you can refer to the names within a class:

```
Namespace_name::name

int ans = Test::i + Test::j;
```

2. The using directive

Because it can rapidly get tedious to type the full qualification for an identifier in a namespace, the **using** keyword allows you to import an entire namespace at once. When used in conjunction with the **namespace** keyword this is called a *using directive*.

```
using Test;

int ans = i + j;
```

3.The using declaration

Another approach is to introduce the equivalent of a local name .You can inject names one at a time into the current scope with a *using declaration*. Unlike the using **directive, which treats names as if they were declared globally to the scope, a** using **declaration is a declaration within the current scope. This means it can override names from a** using **directive:**

```
using Test;        // Using directive
using Test::i;     // Using declaration


int ans = i + Test::j;
```

Notice:

Global namespace can be thought of as the name space without an identifier.(eg ::a refers to global a)

Namespace Aliases

A namespace name can be *aliased* to another name, so you don't have to type an unwieldy name created by a library vendor:

```
namespace test
{
     int i= 200;
}

void main()
{
     namespace t=test;

     cout<<t::i<<endl;
}
```

Unnamed Namespace

When we don't want the namespace name to be known outside the local context, it simply becomes a bother to invent a redundant name that might accidentally clash with someone else's names. Each translation unit contains an unnamed namespace that you can add to by saying "**namespace**" without an identifier:

```
namespace
{
     int i= 200;
}

int main()
{
     cout<<i<<endl;
}
```

Namespace Composition

Often, we would like to compose an interface out of an existing interface.

```
namespace test1
{
     int i= 200;
}
```

```
namespace test2
{
     using namespace test2;
     int j = 300;
}


void main()
{
     cout<<test2::i<<endl;
}
```

If an explicitly qualified name is’nt declared in the namespace mentioned, the compiler looks in namespace mentioned in using-directives.

Selection

Occasionally, we want to access only a few names from a namespace.

```
namespace test1
{
     int i= 200;
     int x;
     int y;
}
namespace test2
{
     using test1::i;
     int j = 300;
}

int main()
{
     cout<<test2::i<<endl;
}
```

Resolving Potential Clash

```
namespace test1
{
     int i= 200;
}

namespace test2
{
     int i = 500;
     int j = 300;
}

namespace test3
{
     using namespace test1;
     using namespace test2;

     using test1::i; // resolve potential clash in
                          favour of test1
}
void main()
{
     cout<<test3::i<<endl;
}
```

Name spaces are open

A namespace definition can be "continued" over multiple header files using a syntax that, for a class, would appear to be a redefinition

```
      We can add names to a namespace from several
namespace declarations.

      namespace test
      {
            int i= 200;
      }

      namespace test
      {
            int j = 300;
      }

      void main()
      {
            cout<<test::i<<endl;
            cout<<test::j<<endl;
      }
```

#define, #ifdef, and #endif

The preprocessor directive **#define** can be used to create compile-time flags. You have two choices: you can simply tell the preprocessor that the flag is defined, without specifying a value:

```
#define FLAG
```

or you can give it a value (which is the typical C way to define a constant):

```
#define PI 3.14159
```

In either case, the label can now be tested by the preprocessor to see if it has been defined:

```
#ifdef FLAG
```

This will yield a true result, and the code following the **#ifdef** will be included in the package sent to the compiler. This inclusion stops when the preprocessor encounters the statement

```
#endif
```

or

```
#endif // FLAG
```

Any non-comment after the **#endif** on the same line is illegal, even though some compilers may accept it. The **#ifdef**/**#endif** pairs may be nested within each other.
The complement of **#define** is **#undef** (short for "un-define"), which will make an **#ifdef** statement using the same variable yield a false result. **#undef** will also cause the preprocessor to stop using a macro. The complement of **#ifdef** is **#ifndef**, which will yield a true if the label has not been defined (this is the one we will use in header files).
There are other useful features in the C preprocessor. You should check your local documentation for the full set.

Value substitution

When programming in C, the preprocessor is liberally used to create macros and to substitute values. Because the preprocessor simply does text replacement and has no concept nor facility for type checking, preprocessor value substitution introduces subtle problems that can be avoided in C++ by using **const** values.

The typical use of the preprocessor to substitute values for names in C looks like this:

```
#define BUFSIZE 100
```

but if you decide to change a value, you must perform hand editing, and you have no trail to follow to ensure you don't miss one of your values (or accidentally change one you shouldn't).

Const

The concept of *constant* was created to allow the programmer to provides safety between what changes and what doesn't.

The use of **const** is not limited to replacing **#define**s in constant expressions

const variable

If you initialize a variable with a value that is produced at runtime and you know it will not change for the lifetime of that variable, it is good programming practice to make it a **const** so the compiler will give you an error message if you accidentally try to change it.

```
#include <iostream.h>

int main()
{
        const int i = 100;

        i++;                    // error
}
```

const pointer

When using **const** with pointers, you have two options:

1. **const** can be applied to what the pointer is pointing to.
2. **const** can be applied to the address stored in the pointer itself.

1. **const** to what the pointer is pointing to.

    a) int const* p;

    b) const int* p;

No initialization is required because **P** can point to anything but the value it points to cannot be changed.The two definitions are the same.To prevent confusion, you should probably stick to the first form.

```
#include<iostream.h>

void main()
{
        int data1 = 10;
        int data2 = 20;
```

```
        int const* p;

        p = &data1;

        *p = *p + 1; // error cannot modify value

        cout<<*p<<endl;

        p = &data2;

        cout<<*p<<endl;
}
```

2. **const** applied to the address stored in the pointer itself.

```
int* const p = &data1;
```

Because the pointer itself is now the **const**, the compiler requires that it be given an initial value that will be unchanged for the life of that pointer.

```
#include<iostream.h>

void main()
{
        int data1 = 10;
        int data2 = 20;

        int* const p = &data1;

        *p = *p + 1;

        cout<<*p<<endl;

        p = &data2;         // error cannot store another pointer address

}
```

Passing by const value

You can specify that function arguments are **const** when passing them by value, such as

```
void f1(const int i) {
  i++; // Illegal -- compile-time error
}
```

const member variable

If you declare a member vaiable **const**, you tell the compiler the variable will not change value.

```
const int data;
```

How is **const** data member initialized ?

constatnt variable should be initialized in the construcor or it will give a compilation error.

```
CA( ):data(100), data2(200)


#include<iostream.h>

Class CA
{
        const int data;

public:
        CA( ):data(100)
        {
        }

        void fun()
        {
                ++data;         //error
        }
};
```

const member function

If you declare a member function **const**, you tell the compiler the functions will not change the data.

```
Class CA
{
        int data;
public:
        void fun() const
        {
                ++data;         //error
        }
};
```

What if you want to create a **const** member function, but you'd still like to change some of the data in the object

There are two ways to change a data member from within a **const** member function

1. The first approach is the historical one and is called *casting away constness*.. You take **this** and cast it to a pointer to an object of the current type. It would seem that **this** is *already* such a pointer. However, inside a **const** member function it's actually a **const** pointer, so by casting it to an ordinary pointer, you remove the **const**ness for that operation.

Use const_cast to cast away const and/or volatile.

```
class CA
{
     const int i;
public:
```

```
      CA():i(0)
      {
`
      {
            int* j = const_cast<int*>(&i);

            ++*j;

            cout<<i<<endl;
      }
};

class CA
{
      int i;
public:
      CA():i(0)
      {

      }
      void fun() const
      {
            CA *self = const_cast<CA*>(this);

            ++self->i;

            cout<<i<<endl;
      }
};
```

2. To put everything out in the open, you should use the **mutable** keyword in the class declaration to specify that a particular data member may be changed inside a **const** object:

```
class Z
{
      int i;
      mutable int j;
public:
      Z();
      void f() const;
};
```

Whats Inspector & mutator

An Inspector is a member function that returns information about an object's state with out changig the objects abstract state. A mutator changes the state of an object.

`Differences with C`

In C, a **const** always occupies storage and its name is global. The C compiler cannot treat a **const** as a compile-time constant.

you will get an error.

```
const int bufsize = 100;
char buf[bufsize];
```

Static members in C++

There are times when you need a single storage space to be used by all objects of a class. In C, you would use a global variable, but this is not very safe. Global data can be modified by anyone, and its name can clash with other identical names in a large project. It would be ideal if the data could be stored as if it were global, but be hidden inside a class, and clearly associated with that class.
This is accomplished with **static** data members inside a class. There is a single piece of storage for a **static** data member, regardless of how many objects of that class you create. All objects share the same **static** storage space for that data member, so it is a way for them to "communicate" with each other. But the **static** data belongs to the class; its name is scoped inside the class and it can be **public**, **private**, or **protected**.

```
Class CA
{
        static int i;
public:


};
```

Because **static** data has a single piece of storage regardless of how many objects are created, that storage must be defined in a single place. The compiler will not allocate storage for you. The linker will report an error if a **static** data member is declared but not defined.

you must define storage for that static data member in the definition file like this:

```
int A::i = 1;
```

you cannot have **static** data members inside local classes

```
void fun()
{
        class CB
        {
                static      int j; // error
        };

}
```

static member functions

You can also create **static** member functions that, like **static** data members, work for the class as a whole rather than for a particular object of a class. Instead of making a global function that lives in and "pollutes" the global or local namespace, you bring the function inside the class.
You can call a **static** member function in the ordinary way, with the dot or the arrow, in association with an object. However, it's more typical to call a **static** member function by itself, without any specific object, using the scope-resolution operator, like this:

```
//: C10:SimpleStaticMemberFunction.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

RTTI

RTTI, like exceptions, depends on type information residing in the virtual function table. If you try to use RTTI on a class that has no virtual functions, you'll get unexpected results.

typeid( )

It determines the precise type of an object at runtime.

```
#include <cassert>
#include <typeinfo>
using namespace std;

int main() {
  assert(typeid(47) == typeid(int));
  assert(typeid(0) == typeid(int));
  int i;
  assert(typeid(i) == typeid(int));
  assert(typeid(&i) == typeid(int*));
}
```

Producing the proper type name

typeid() returns a reference to a standard library class called type_info.

type_info has a member function returns the name of the parameter's type in an implementation specific format.

```
#include <iostream>
#include <typeinfo>

void main()
{
     int j;
     cout<<typeid(j).name()<<endl;
}
```

```
#include <iostream>
#include <typeinfo>

using namespace std;

class CA
{
public:
          void fun()
          {
               cout<<"fun"<<endl;
          }
};
int main()
{
     CA obj;
     cout << typeid(obj).name() << endl;
}
```

The following class contains a nested class.

```
#include <iostream>
#include <typeinfo>

using namespace std;

class CA
{
```

```
	class CB{};

	CB *m_inner;
public:

	CA():m_inner(new CB)
	{}

	~CA()
	{
		delete m_inner;
	}

	CB* get_inner()
	{
		return m_inner;
	}

};

int main()
{

	CA obj;
	cout << typeid(obj).name() << endl;
	cout << typeid(*obj.get_inner()).name() << endl;
}
```

Downcast

A downcast is the conversion of a Base* to a Derived*, where class Derived is publicly derived from Base. A downcast is used when the client code thinks that a Base* points to an object of class Derived.

A downcast from a base class pointer to a derived class pointer instructs the compiler to blindly reinterpret the bits of the the pointer. Invalid conversion would cause big, so use type-safe downcassting using **dynamic_cast<T>(x)**

Dynamic_cast<T>(X)

Dynamic_cast is like old style cast (T)x, casts a value of x to the type T. Dynamic_cast<T>(x) has several advantages over the old style cast. It never

performs an invalid conversion, since it checks that the cast is legal at runtime.

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

class Shape
{
public:
     virtual void draw()=0;
     virtual ~Shape() throw()
     {}
};

class Circle:public Shape
{
public:
     virtual void draw()
     {}

};

class Square:public Shape
{
public:
     virtual void draw()
     {}

};

void sample(Shape *p) throw()
{
     Circle * cp = dynamic_cast<Circle*>(obj);
     Square * sp = dynamic_cast<Square*>(p);

     if(cp != NULL)
          cout<<"circle"<<endl;

     if(sp != NULL)
          cout<<"Square"<<endl;
}

int main()
{
     Circle c;
     Square s;
     sample(&c);
     sample(&s);
```

```
}
```

Nonpolymorphic types

Although **typeid( )** works with nonpolymorphic types (those that don't have a virtual function in the base class), the information you get this way is dubious.

```
#include <cassert>
#include <typeinfo>
using namespace std;

class X {
 int i;
public:
  // ...
};

class Y : public X {
  int j;
public:
  // ...
};

int main() {
  X* xp = new Y;
  assert(typeid(*xp) == typeid(X));
  assert(typeid(*xp) != typeid(Y));
}
```

If you create an object of the derived type and upcast it,
X* xp = new Y. The **typeid( )** operator will produce results, but not the ones you might expect. Because there's no polymorphism, the static type information is used.

RTTI is intended for use only with polymorphic classes.

The following table describes the different forms of casting:

static_cast	For an upcast or automatic type conversion.
const_cast	To cast away **const** and/or **volatile**.
dynamic_cast	For type-safe downcasting.
Reinterpret_cast	To cast to a completely different meaning. This is the most dangerous of all the casts.

const_cast

```
class CA
{
	const int i;
public:
	CA():i(0)
	{

	}
	void fun()
	{
		int* j = const_cast<int*>(&i);

		++*j;

		cout<<i<<endl;
	}
};

class CA
{
	int i;
public:
	CA():i(0)
	{
```

```
	}
	void fun() const
	{
		CA *self = const_cast<CA*>(this);

		++self->i;

		cout<<i<<endl;
	}
};
```

Reinterpret_cast

```
class CA
{
public:
	void fun()
	{
		cout<<"have fun"<<endl;
	}
};

class CB
{
public:
	void fun()
	{
		cout<<"have fun"<<endl;
	}
};

void main()
{
	CB *pv = reinterpret_cast<CB*>(new CA);

	reinterpret_cast<CA*>(pv)->fun();
}
```

static_cast

```
void main()
{
	void *pv = static_cast<void*>(new CA);

	reinterpret_cast<CA*>(pv)->fun();
}
```

Exceptions

Throwing an exception

If you encounter an exceptional situation in your code – that is, one where you don't have enough information in the current context to decide what to do – you can send information about the error into a larger context by creating an object containing that information and "throwing" it out of your current context. This is called *throwing an exception*.

```
throw myerror("something bad happened");
```

The try block

If you're inside a function and you throw an exception (or a called function throws an exception), that function will exit in the process of throwing. If you don't want a **throw** to leave a function, you can set up a special block within the function. This is called the *try block* because you try your various function calls there. The try block is an ordinary scope, preceded by the keyword **try**:

```
try {
  // Code that may generate exceptions
}
```

Exception handlers

The thrown exception ends up in *exception handlers*, and there's one for every exception type you want to catch. Exception handlers immediately follow the try block and are denoted by the keyword **catch**:

```
try {
// code that may generate exceptions
} catch(type1 id1) {
  // handle exceptions of type1
} catch(type2 id2) {
  // handle exceptions of type2
}
// etc...
```

Catching any exception

As mentioned, if your function has no exception specification, *any* type of exception can be thrown. One solution to this problem is to create a handler that

catches any type of exception. You do this using the ellipses in the argument list.

```
catch(...) {
  cout << "an exception was thrown" << endl;
}
```

This will catch any exception, so you'll want to put it at the *end* of your list of handlers to avoid pre-empting any that follow it.

There are two basic models in exception-handling theory.

1.Termination
2. resumption

In *termination* (which is what C++ supports) you assume the error is so critical there's no way to get back to where the exception occurred. Whoever threw the exception decided there was no way to salvage the situation, and they don't *want* to come back

In *resumption* the exception handler is expected to do something to rectify the situation, and then the faulting function is retried, presuming success the second time.
To achieve resumption don't throw an exception; call a function that fixes the problem. Alternatively, place your **try** block inside a **while** loop that keeps reentering the **try** block until the result is satisfactory.

It is easy to terminate to a handler that's far away, but to jump to that handler and then back again may be too conceptually difficult for large systems where the exception can be generated from many points.

The exception specification

C++ provides a syntax to allow you to politely tell the user what exceptions this function throws, so the user may handle them. This is the *exception specification* and it's part of the function declaration, appearing after the argument list.

```
void f() throw(toobig, toosmall, divzero);

void f() throw();   it means that no exceptions are
                    thrown from a function.
```

Rethrowing an exception

Rethrowing is throwing the exception to go to the exception handlers in the next-higher context. In addition, everything about the exception object is preserved, so the handler at the higher context that catches the specific exception type is able to extract all the information from that object.

```
catch(...) {
  cout << "an exception was thrown" << endl;
  throw;
}
```

```
#include<iostream.h>

void main()
{
     try{
          try{
               throw 100;
          }catch(int)
          {
               cout<<"hai 1";
               throw;
          }
     }catch(int)
     {
          cout<<"hai 2";
     }
}
```

Uncaught exceptions

terminate()

If an exception is uncaught, the special function **terminate( )** is automatically called. Its default value is the Standard C library function **abort( ).**

set_terminate()

You can install your own **terminate( )** function using the standard **set_terminate( )** function.

1. **terminate( )** must take no arguments
2. **void** must be the return value.
3. **terminate( )** must not return or throw an exception

```
#include <exception>
#include <iostream.h>
#include <cstdlib>

void terminator()
{
     cout << "Error occured " << endl;
     abort();
}

class CA
{
public:

     void test()
     {
          throw "error";
     }
};

void main()
{
     set_terminate(terminator);
     CA obj;
     obj.test();
}
```

Function-level try blocks

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

int main() try {

    throw "main";

} catch(const char* msg)
{
  cout << msg << endl;
}
```

Catch by reference, not by value

If you throw an object of a derived class and it is caught *by value* in a handler for an object of the base class, that object is "sliced" – that is, the derived-class elements are cut off and you'll end up with the base-class object being passed.

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

class Base
{
public:
     virtual void what()
     {
            cout << "Base" << endl;
     }
};

class Derived : public Base
{
public:
      void what()
     {
            cout << "Derived" << endl;
     }
};

void f() { throw Derived(); }

int main()
{
     try
     {
            f();
     }
     catch(Base b)
     {
            b.what();
     }
     try
     {
            f();
     }
     catch(Base& b)
     {
            b.what();
```

```
        }
    }
```

Don't cause exceptions in destructors.

If a destructor is called while unwinding the stack caused by another exception, and that destructor throws an exception, the destructor must catch the second (nested) excpetion. Otherwise the handling mechanism calls terminate function.

```
#include<memory>
#include<stdexcept>
#include<iostream>

using namespace std;

class CA
{
public:
      ~CA()
      {
            cout<<"dest"<<endl;
            try{
                  term();
            }
            catch(bad_cast)
            {
                  //....
            }
      }

      void term()
      {
            // uninitializing goes here
            throw bad_cast();
      }
};

typedef auto_ptr<CA> CAPtr;

void main()
{
      CAPtr p(new CA);
}
```

Avoid naked pointers

A naked pointer usually means vulnerability in the constructor if resources are allocated for that pointer. A pointer doesn't have a destructor, so those resources won't be released if an exception is thrown in the constructor.

```
#include<memory>
#include<iostream>

using namespace std;

class CA
{
public:
	CA()
	{
		cout<<"const"<<endl;
	}
	~CA()
	{
		cout<<"dest"<<endl;
	}
	void fun()
	{
		cout<<"fun"<<endl;
	}
};

typedef auto_ptr<CA> CAPtr;

void main()
{
	CAPtr p(new CA);
	p->fun();
}
```

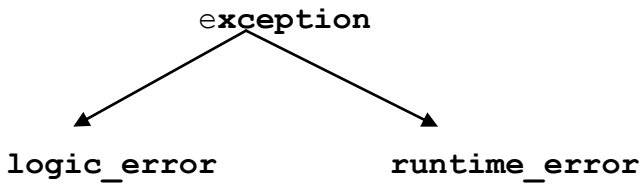
Zombie Objects.

Zoombie objects are C++'s version of the living dead.Objects that are'nt quiet alive but are'nt quiet dead either.

When an environment doesn't support throw or when a programmer decides to avoid throwing exceptions from constructors, an object that can't finish its constructor can set an internal flag to indicate the object is unusable. Then class provides a query (inspector) member function so that user can see whether the object is usable or zombie.

Zombie technique is inferior to throwing an exception.

Standard exceptions



Exception classes derived from logic_error

domain_error	Reports violations of a precondition.
invalid_argument	Indicates an invalid argument to the function it's thrown from.
length_error	
out_of_range	Reports an out-of-range argument.
bad_cast	Thrown for executing an invalid **dynamic_cast** expression in run-time type identification .
bad_typeid	Reports a null pointer **p** in an expression **typeid(*p)**.

Exception classes derived from runtime_error

range_error	Reports violation of a postcondition.
overflow_error	Reports an arithmetic overflow.
bad_alloc	Reports a failure to allocate storage.

Constructors & Destructors

A constructor is automatically called at the moment an object is created. Developers of a class provide a set of constructors that define how objects of that class can be initialized. For a class named X, the destructor is a member function named x.

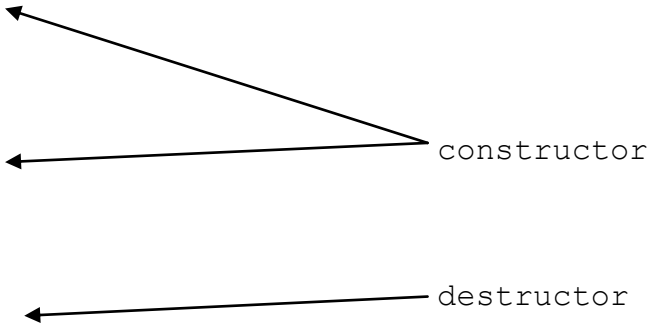
The destructor is the last member function ever called for an object. The destructor’s typical purpose is to release resources that the object is holding. A class can have at most one destructor. For a class named X, the destructor is a member function named ~x.

```
class CA
{
      CA()
      {

      }

      CA(int a)
      {

      }
      ~CA()
      {
            // do something
      }
};
```




```
int main()
{
      CA obj1;
      CA obj2(678);

      CA *obj3 = new CA;
      delete obj3.
}
```

Object construction using **new** is a two step process.

1. sizeof(CA) bytes of memory are allocated.
2. The pointer returned from the first step is passed as the constructor's this parameter. The call to the constructor is wrapped in a **try** block. If an exception occurs in the constructor all bytes allocated are automatically released back to the heap.

A local object dies when it is popped from the runtime stack. It is popped from the stack at the close of the block {...}.

A Object allocated via new(dynamically) dies when the object is deleted.

What happens when a destructor is executed ?

1. First the destructor's body {...} is executed.

2. Then, destructors for member objects are called in the reverse orderthat the member objects appear in the class body.

3. Then, runtime system calls the destructors for immediate base classes.

4. Virtual base classes are special their destructors are Called at the end of the most derived class's destructor.

Default Constructor.
Its the constructor that can be called with no arguments. It is posiible for a default constructor to take arguments, provided they are given default values.

```
CA(int i = 3, int j = 5);
```

Copy Constructor

Copy constructor initializes an object by copying the state from another object of the same class.If the class of object being copied is X, the copy constrouctor's signature is usually X::X(const X&).

```
class CA
{
     int data;
public:
     CA()
     {
          cout<<"const"<<endl;
     }
     CA(const CA &obj)
     {
          cout<<"copy const"<<endl;
          data = obj.data;
     }
     ~CA()
     {
          cout<<"dest"<<endl;
     }
};

void main()
{
     CA obj3;
     CA obj4 = obj3;
```

```
    CA *obj= new CA;
    CA *obj2 = new CA(*obj);

    delete obj;
    delete obj2;
}
```

Aggregate initialization

An *aggregate* is just what it sounds like: a bunch of things clumped together. An array is an aggregate of a single type.Initializing aggregates can be error-prone and tedious. C++ *aggregate initialization* makes it much safer. When you create an object that's an aggregate, all you must do is make an assignment, and the initialization will be taken care of by the compiler.

For an array of built-in types this is quite simple:

```
int a[5] = { 1, 2, 3, 4, 5 };
```

If you try to give more initializers than there are array elements, the compiler gives an error message. But what happens if you give *fewer* initializers? For example:

```
int b[6] = {0};
```

Here, the compiler will use the first initializer for the first array element, and then use zero for all the elements without initializers.

A second shorthand for arrays is *automatic counting*, in which you let the compiler determine the size of the array based on the number of initializers:

```
int c[] = { 1, 2, 3, 4 };
```

Because structures are also aggregates, they can be initialized in a similar fashion. Because a C-style **struct** has all of its members **public**, they can be assigned directly:

```
struct X {
  int i;
  float f;
  char c;
};

X x1 = { 1, 2.2, 'c' };
```

If you have an array of such objects, you can initialize them by using a nested set of curly braces for each object:

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

Here, the third object is initialized to zero.

If any of the data members are **private** or even if everything's **public** but there's a constructor, things are different. The constructors must be called to perform the initialization.

```
struct Y {
  float f;
```

```
  int i;
  Y(int a);
};

Y y1[] = { Y(1), Y(2), Y(3) };
```

Here's a second example showing multiple constructor arguments:

```
//: C06:Multiarg.cpp
// Multiple constructor arguments
// with aggregate initialization
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
  for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```

What are Virtual Destructors.

A destructor is called whenever an object is deleted, but there are some cases when users code doesn't know which destructor should be called.

For eg. If a base class pointer is pointing to derived class object, when the object dies only the base

class destructor is called. This could be a serious error, especially if the derived destructor is supposed to release some resources.

The solution is to make the base class destructor virtual.once that is done, the compiler dynamically binds to the destructor, and thus the right destructor is always called.

```
class CA
{
public:
	virtual ~CA()
	{
		cout<<"dest CA"<<endl;
	}
};

class CB:public CA
{
public:
	~CB()
	{
		cout<<"dest cb"<<endl;
	}
};

void main()
{
	CA *obj = new CB;
	delete obj;
}
```

Virtual destructors are extremely valuable when some derived classes have some specified cleanup code.If any code anywhere deletes an derived class object via a base class pointer, then the base class's destructor needs to be virtual.

A class can have a pure virtual destructor provided it gives an explicit definition else where.

```
class CA
{
public:
	virtual ~CA() = 0;
};

	inline CA::~CA()
```

```
    {
        cout<<"dest CA"<<endl;
    }
```

What are Virtual Constructors.

The virtual keyword cannot be applied to a constructor since constructor turns raw bits into a living object, and until there is a living object against which to invoke a member function, the member function cannot possibly work correctly.

When the constructor of a base class calls a virtual function, why is'nt the override called ?

Objects of a derived class mature during construction. While the bas class's constructor is executing, the object is merely a base class object. Later when the derived class's contructor begins executing, the object matures into a derived class object.

If a virtual function is invoked while the object is still immature, the immature version of the virtual function is called.

When the destructor of a base class calls a virtual function, why is'nt the override called ?

Just as object of derived class matures into a derived class object during construction, it reverts back into a base class object during destruction.

```
class CA
{
public:
    CA()
    {
        fun();
    }

    ~CA()
    {
        fun();
    }

    virtual void fun()
```

```
	{
		cout<<"fun ca"<<endl;
	}
};

class CB:public CA
{
public:
	void fun()
	{
		cout<<"fun cb"<<endl;
	}
};


void main()
{
	CB obj;
}
```

New & Delete

In C++, new & delete create and destroy objects. malloc() & free()allocate and deallocate memory.

New is a two step operation.
a. The first step is to allocate memory.
b. The second step is to call the appropriate constructor of the class.

Delete is a two step operation.
a. It frist calls the destructor on the object.
b. Then it releases the memory.

```
class CA
```

```
{
	CA()
	{
		cout<<"const"<<endl;
	}
	~CA()
	{
		cout<<"dest"<<endl;
	}
};

void main()
{
	CA *obj = new CA;

	delete obj;
}
```

new never ever returns NULL. Instead, if **new** runs out of memory, it throws an exception of type **bad_alloc**.

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

void fun() throw(bad_alloc)
{
	CA *obj = new CA;
	obj->fun();
	delete obj;
}
```

By replacing new by new(nothrow) , new can return NULL rather than throwing an exception.

```
void fun() throw()
{
	CA *obj = new(nothrow) CA();

	If(obj == NULL)
	{

	}
	else
	{
		obj->fun();
		delete obj;
	}
}
```

Flexibility:

The new operator can be overloaded by a class.

```
class CA
{
public:
      static void * operator new( unsigned int cb )
      {
            cout<<"new "<<endl;
            return malloc(cb);
      }


      static void  operator delete(void * obj)
      {
            cout<<"delete "<<endl;
            free(obj);
      }
};

void main()
{
      CA *obj = new CA;

      delete obj;
}
```

Creating Array of objects on the heap.

```
int main()
{
      CA *obj = new CA[10];

      delete[] obj;  <------  [] is required when deallocating the
                              array
}
```

when the array is deallocated, the [] must appear just after the delete keyword. The real solution is not to use pointers at all but instead to use a good container class library.

After **CA* obj = new CA[n]** how does compiler know that there are n objects to be destroyed during **delete[] obj**. The compiler can use any technique it wants to use, but there is one popular ones.

The code generated by **CA* obj = new CA[n]** mightallocate an extra sizeof(size_t) bytes of memory and put the value n just before the first **CA** object. The **delete[] obj** would find n by looking at the fixed offset before the first **CA** object and would deallocate the memory starting at the begining of the allocation.

Placement new

It's a way to pass parameters to allocator rather than just to the constructor.

```
#include<iostream.h>
#include<new>

using namespace std;

class CA
{

public:
      CA()
      {
            cout<<"const"<<endl;
      }
      ~CA()
      {
            cout<<"dest"<<endl;
      }
      virtual void fun()
      {
            cout<<"fun ca"<<endl;
      }
};


void main()
{
      char memory[sizeof(CA)];
      void *p = memory;
      CA *obj = new(p) CA;
      obj->fun();
      obj->~CA();  <-------- Explicitily call the destructor
}
```

The programmer takes sole responsibility to deallocate the object.This is done by explicitly calling the destructor, which is one of the few times a destructor should be called explicitly.

Warning:

Pointers returned from **new** should not be deallocated with **free()** & pointers returned from **malloc()** should not be deallocated with **delete.**

Even if it appears to work on your particular compiler, please don't do it. Corrupting the heap is very subtle and disastrous thing. It's just not worth the trouble.

When a pointer is deleted twice, first time the object is safely destructed and the memory pointed to by pis safely returned to the heap. The second time, the remains of what used to be an object at *
Are passed to the destructor which could be disastrous, and the memory is handed back to the heap second time. This is likely to corrupt the heap and its list of free memory.

```
int main()
{
      CA *obj     = new CA;
      CA *obj1    = obj;
      delete obj;
      delete obj1;
}
```

Assignment:

1. How can class CA guarantee the CA objects are created only with new and not on the stack.

2. Create a global pool of recycled objects.

   When a dynamically allocated object is no longer needed, rather than delete, using a member function add the object to a static pool of objects.

   When a object is needed rather than new, using a static member function return the first entry from the list or use new is list is empty.

   When memory runs low, the pool of objects need to be flushed to free up available memory.

```
#include<iostream.h>
#include<new>
#include<stdlib.h>
using namespace std;


class CA
{
      CA()
      {
            cout<<"const"<<endl;
            next = 0;
      }
      ~CA()
      {
            cout<<"dest"<<endl;
```

```
	}

	static CA* head;
		  CA* next;
public:
	static CA* create()
	{
		if (head==NULL)
			return new CA;

		CA *p = head;
		head = head->next;
		return p;
	}

	void discard()
	{
		next = head;
		head = this;
	}

	static void flushpool()
	{
		while(head != 0)
			delete create();
	}

	virtual void fun()
	{
		cout<<"fun ca"<<endl;
	}

};

CA* CA::head = 0;

void flushallpools()
{
	CA::flushpool();

}
int main()
{
	set_new_handler(flushallpools);

	CA *obj = CA::create();

	obj->fun();
	obj->discard();

	CA *obj1 = CA::create();
	CA *obj2 = CA::create();

	obj1->fun ();
	obj1->discard();

}
```

RTTI

RTTI, like exceptions, depends on type information residing in the virtual function table. If you try to use RTTI on a class that has no virtual functions, you'll get unexpected results.

typeid( )

It determines the precise type of an object at runtime.

```
#include <cassert>
#include <typeinfo>
using namespace std;

int main() {
  assert(typeid(47) == typeid(int));
  assert(typeid(0) == typeid(int));
  int i;
  assert(typeid(i) == typeid(int));
  assert(typeid(&i) == typeid(int*));
}
```

Producing the proper type name

typeid() returns a reference to a standard library class called type_info.

type_info has a member function returns the name of the parameter's type in an implementation specific format.

```
#include <iostream>
#include <typeinfo>

void main()
{
     int j;
     cout<<typeid(j).name()<<endl;
}
```

```
#include <iostream>
#include <typeinfo>

using namespace std;
```

```
class CA
{
public:
          void fun()
          {
               cout<<"fun"<<endl;
          }
};
int main()
{
     CA obj;
     cout << typeid(obj).name() << endl;
}
```

The following class contains a nested class.

```
#include <iostream>
#include <typeinfo>

using namespace std;

class CA
{
     class CB{};

     CB *m_inner;
public:

     CA():m_inner(new CB)
     {}

     ~CA()
     {
          delete m_inner;
     }

     CB* get_inner()
     {
          return m_inner;
     }

};

int main()
{

     CA obj;
     cout << typeid(obj).name() << endl;
```

```
     cout << typeid(*obj.get_inner()).name() << endl;
}
```

Downcast

A downcast is the conversion of a Base* to a Derived*, where class Derived is publicly derived from Base. A downcast is used when the client code thinks that a Base* points to an object of class Derived.

A downcast from a base class pointer to a derived class pointer instructs the compiler to blindly reinterpret the bits of the the pointer. Invalid conversion would cause big, so use type-safe downcassting using **dynamic_cast<T>(x)**

Dynamic_cast<T>(X)

Dynamic_cast is like old style cast (T)x, casts a value of x to the type T. Dynamic_cast<T>(x) has several advantages over the old style cast. It never performs an invalid conversion, since it checks that the cast is legal at runtime.

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

class Shape
{
public:
     virtual void draw()=0;
     virtual ~Shape() throw()
     {}
};

class Circle:public Shape
{
public:
     virtual void draw()
     {}

};

class Square:public Shape
{
public:
     virtual void draw()
```

```
      {}

};

void sample(Shape *p) throw()
{
      Circle * cp = dynamic_cast<Circle*>(obj);
      Square * sp = dynamic_cast<Square*>(p);

      if(cp != NULL)
            cout<<"circle"<<endl;

      if(sp != NULL)
            cout<<"Square"<<endl;
}

int main()
{
      Circle c;
      Square s;
      sample(&c);
      sample(&s);
}
```

Nonpolymorphic types

Although **typeid( )** works with nonpolymorphic types (those that don't have a virtual function in the base class), the information you get this way is dubious.

```
#include <cassert>
#include <typeinfo>
using namespace std;

class X {
 int i;
public:
  // ...
};

class Y : public X {
  int j;
public:
  // ...
};

int main() {
  X* xp = new Y;
```

```
    assert(typeid(*xp) == typeid(X));
    assert(typeid(*xp) != typeid(Y));
  }
```

If you create an object of the derived type and upcast it,
X* xp = new Y. The **typeid( )** operator will produce results, but not the ones you might expect. Because there's no polymorphism, the static type information is used.

RTTI is intended for use only with polymorphic classes.

The following table describes the different forms of casting:

static_cast	For an upcast or automatic type conversion.
const_cast	To cast away **const** and/or **volatile**.
dynamic_cast	For type-safe downcasting.
Reinterpret_cast	To cast to a completely different meaning. This is the most dangerous of all the casts.

const_cast

```
class CA
{
     const int i;
public:
     CA():i(0)
     {

     }
     void fun()
     {
          int* j = const_cast<int*>(&i);

          ++*j;
```

```
            cout<<i<<endl;
      }
};

class CA
{
      int i;
public:
      CA():i(0)
      {

      }
      void fun() const
      {
            CA *self = const_cast<CA*>(this);

            ++self->i;

            cout<<i<<endl;
      }
};
```

Reinterpret_cast

```
class CA
{
public:
      void fun()
      {
            cout<<"have fun"<<endl;
      }
};

class CB
{
public:
      void fun()
      {
            cout<<"have fun"<<endl;
      }
};

void main()
{
      CB *pv = reinterpret_cast<CB*>(new CA);

      reinterpret_cast<CA*>(pv)->fun();
```

```
}
```

static_cast

```
void main()
{
	void *pv = static_cast<void*>(new CA);

	reinterpret_cast<CA*>(pv)->fun();
}
```

Operator Overloading

operator=

```
#include<iostream.h>

class CA
{
	int data;
public:
	CA(int t):data(t){}

	CA& operator=(CA& obj)
	{
		data = obj.data;
		return obj;
	}
};
void main()
{
	CA obj1(10);
	CA obj2(20);
	CA obj3(30);

	obj2 = obj1;		// obj2.operator=(obj1)

	obj3 = obj2 = obj1;	// obj2.operator=(obj1)
				// obj3.operator=(obj1)

}
```

operator*

```
#include<iostream.h>

class CA
{
	int data;
public:
	CA(int t):data(t){}

	int operator*()
	{
```

```
		return data;
	}

};

void main()
{
	CA obj1(10);

	cout<<*obj1<<endl;
}
```

operator==

```
#include<iostream.h>

class CA
{
	int data;
public:
	CA(int t):data(t){}

	bool operator==(CA& obj)
	{
		if( data == obj.data)
			return true;
		else
			return false;
	}
};

void main()
{
	CA obj1(10);
	CA obj2(10);

	if( obj1 == obj2)
		cout<<"equal"<<endl;
}
```

How are Postfix & Prefix versions of operator++ & operator—distinguished ?

The prefix version takes no parameters and the postfix version takes a single parameter of type int. Note that users should avoid obj++ in their code unless the old state of data is needed.

```
#include<iostream.h>

class CA
{
	int data;
public:
	CA(int t):data(t){}

	CA& operator++()	// prefix
	{
```

```
                ++data;
                return *this;
        }

        CA operator++(int)      // postfix
        {
                CA old = *this;
                ++(*this);
                return old;
        }

};

void main()
{
        CA obj1(10);

        obj1++;
        ++obj1;
}
```

Why do subscript operators usually come in pairs ?

The C++ operators that cant be overloaded are

a. .
b. .*
c. ? :
d. sizeof
e. typeid
f. ::

Alternate linkage specifications

What happens if you're writing a program in C++ and you want to use a C library? If you make the C function declaration,

```
float f(int a, char b);
```

the C++ compiler will decorate this name to something like **_f_int_char** to support function overloading. However, the C compiler that compiled your C library has most definitely *not* decorated the name, so its internal name will be **_f**. Thus, the linker will not be able to resolve your C++ calls to **f( ).**

The escape mechanism provided in C++ is the *alternate linkage specification*, which was produced in the language by overloading the **extern** keyword.

```
extern "C" float f(int a, char b);
```

This tells the compiler to give C linkage to **f( )** so that the compiler doesn't decorate the name. The only two types of linkage specifications supported by the standard are **"C"** and **"C++,"**

```
        extern "C" void fun()
```

```
{
        cout<<""<<endl;
}

extern "C++" void fun1()
{
        cout<<""<<endl;
}
```

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

Or, for a header file,

```
extern "C" {
#include "Myheader.h"
}
```

The Standard Template Library

1. Container classes
    1. Sequences
        1. vector
        2. deque
        3. list
    2. Container adaptors
        1. stack
        2. queue
        3. priority_queue
    3. Associative Containers
        1. set
        2. map
        3. hash
    4. String package
        1. Character Traits
        2. char_traits
        3. basic_string

Introduction to STL.

The Standard Template Library, or *STL*, is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a *generic* library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL.

Containers

Like many class libraries, the STL includes *container* classes: classes whose purpose is to contain other objects. The STL includes the classes vector, list, deque, set, multiset, map, multimap, hash_set, hash_multiset, hash_map, and hash_multimap. Each of these classes is a template, and can be instantiated to contain any type of object.

Algorithms

The STL also includes a large collection of *algorithms* that manipulate the data stored in containers.ex. You can reverse the order of elements in a container.

Iterators

Iterators are a generalization of pointers: they are objects that point to other objects. As the name suggests, iterators are often used to iterate over a range of objects: if an iterator points to one element in a range, then it is possible to increment it so that it points to the next element.
Iterators are central to generic programming because they are an interface between containers and algorithms: algorithms typically take iterators as arguments, so a container need only provide a way to access its elements using iterators. This makes it possible to write a generic algorithm that operates on many different kinds of containers, even containers as different as a vector and a doubly linked list.

1 Sequences

1.a.vector<T, Alloc>

Description

A `vector` is a Sequence that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a `vector` may vary dynamically; memory management is automatic. `Vector` is the simplest of the STL container classes, and in many cases the most efficient.

Example

```
#include <iostream>
#include <vector>

using namespace std ;

typedef  Vector<int> Container;
typedef Container::iterator Iter;

const ARRAY_SIZE = 4;

void main()
{
    Container theVector;

    for (int i = 0; i < ARRAY_SIZE; i++)
        theVector.push_back((i + 1) * 100);
```

```
    cout << "First element: " << theVector.front() <<
endl;
    cout << "Last element:  " << theVector.back() <<
endl;
    cout << "Elements :     " << theVector.size() <<
endl;

    theVector.erase(theVector.end() - 1);
    theVector.erase(theVector.begin());

  for(Iter j = theVector.begin(); j != theVector.end();
j++)
          (*j) = 100;
}
```

Iterators in reversible containers

vector<string>::reverse_iterator r; for(r = lines.rbegin(); r != lines.rend(); r++) cout << *r << endl;

1.b. deque<T, Alloc>

Description

A `deque` is very much like a vector: like `vector`, it is a sequence that supports random access to elements, constant time insertion and removal of elements at the end of the sequence, and linear time insertion and removal of elements in the middle.
The main way in which `deque` differs from `vector` is that `deque` also supports constant time insertion and removal of elements at the beginning of the sequence. Additionally, `deque` does not have any member functions analogous to `vector`'s `capacity()` and `reserve()`, and does not provide any of the guarantees on iterator validity that are associated with those member functions.

Example

```
#include <iostream>
#include <deque>

using namespace std;

typedef deque<int >  INTDEQUE;
void printcontents (INTDEQUE  deque);

void main()
{
     INTDEQUE  dequetest;

     dequetest.push_back(1);
```

```
      dequetest.push_front(5);
      dequetest.pop_front();
      dequetest.pop_back();

      printcontents (dequetest);
}

void printcontents (INTDEQUE  deque)
{
     INTDEQUE::iterator it;
     for(it = deque.begin();it != deque.end();it++)
            cout << *it <<endl ;
}
```

1.c.list<T, Alloc>

Description

A `list` is a doubly linked list. That is, it is a Sequence that supports both forward and backward traversal, and (amortized) constant time insertion and removal of elements at the beginning or the end, or in the middle. `Lists` have the important property that insertion and splicing do not invalidate iterators to list elements, and that even removal invalidates only the iterators that point to the elements that are removed. The ordering of iterators may be changed (that is, `list<T>::iterator` might have a different predecessor or successor after a list operation than it did before), but the iterators themselves will not be invalidated or made to point to different elements unless that invalidation or mutation is explicit.

Example

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

list<int> L;
L.push_back(0);
L.push_front(1);
L.insert(++L.begin(), 2);
copy(L.begin(), L.end(), ostream_iterator<int>(cout, " "));
// The values that are printed are 1 2 0
```

Container adapters

2.a.stack<T, Sequence>

Description

A `stack` is an adaptor that provides a restricted subset of Container functionality: it provides insertion, removal, and inspection of the element at the top of the stack. `Stack` is a "last in first out" (LIFO) data structure: the element at the top of a `stack` is the one that was most recently added. [1] `Stack` does not allow iteration through its elements. [2]

`Stack` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is deque, but a different type may be selected explicitly.

Example

```
int main() {
  stack<int> S;
  S.push(8);
  S.push(7);
  S.push(4);
  assert(S.size() == 3);

  assert(S.top() == 4);
  S.pop();

  assert(S.top() == 7);
  S.pop();

  assert(S.top() == 8);
  S.pop();

  assert(S.empty());
}
```

2.b.queue<T, Sequence>

Description

A `queue` is an adaptor that provides a restricted subset of Container functionality A `queue` is a "first in first out" (FIFO) data structure. [1] That is, elements are added to the back of the `queue` and may be removed from the front; `Q.front()` is the element that was added to the `queue` least recently. `Queue` does not allow iteration through its elements. [2]
`Queue` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is deque, but a different type may be selected explicitly.

Example

```
int main() {
  queue<int> Q;
  Q.push(8);
  Q.push(7);
  Q.push(6);
  Q.push(2);

  assert(Q.size() == 4);
  assert(Q.back() == 2);

  assert(Q.front() == 8);
  Q.pop();

  assert(Q.front() == 7);
  Q.pop();

  assert(Q.front() == 6);
  Q.pop();

  assert(Q.front() == 2);
```

```
  Q.pop();

  assert(Q.empty());
}
```

2.c.priority_queue<T, Sequence>

Description

A `priority_queue` is an adaptor that provides a restricted subset of Container functionality: it provides insertion of elements, and inspection and removal of the top element. It is guaranteed that the top element is the largest element in the `priority_queue`, where the function object `Compare` is used for comparisons. [1] `Priority_queue` does not allow iteration through its elements. [2] `Priority_queue` is a container adaptor, meaning that it is implemented on top of some underlying container type. By default that underlying type is vector, but a different type may be selected explicitly.

Example

```
int main() {
  priority_queue<int> Q;
  Q.push(1);
  Q.push(4);
  Q.push(2);
  Q.push(8);
  Q.push(5);
  Q.push(7);

  assert(Q.size() == 6);

  assert(Q.top() == 8);
  Q.pop();

  assert(Q.top() == 7);
  Q.pop();

  assert(Q.top() == 5);
  Q.pop();

  assert(Q.top() == 4);
  Q.pop();

  assert(Q.top() == 2);
  Q.pop();

  assert(Q.top() == 1);
  Q.pop();

  assert(Q.empty());
}
```

3. Associative Containers

3.a.set<Key, Compare, Alloc>

Description

`Set` is a Sorted Associative Container that stores objects of type `Key`. `Set` is a Simple Associative Container, meaning that its value type, as well as its key type, is `Key`. It is also a Unique Associative Container, meaning that no two elements are the same.
`Set` and multiset are particularly well suited to the set algorithms includes, set_union, set_intersection, set_difference, and set_symmetric_difference. The reason for this is twofold. First, the set algorithms require their arguments to be sorted ranges, and, since set and multiset are Sorted Associative Containers, their elements are always sorted in ascending order. Second, the output range of these algorithms is always sorted, and inserting a sorted range into a `set` or `multiset` is a fast operation: the Unique Sorted Associative Container and Multiple Sorted Associative Container requirements guarantee that inserting a range takes only linear time if the range is already sorted.
`Set` has the important property that inserting a new element into a `set` does not invalidate iterators that point to existing elements. Erasing an element from a set also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

Example

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  const int N = 6;
  const char* a[N] = {"isomer", "ephemeral", "prosaic",
                      "nugatory", "artichoke", "serif"};
  const char* b[N] = {"flat", "this", "artichoke",
                      "frigate", "prosaic", "isomer"};

  set<const char*, ltstr> A(a, a + N);
  set<const char*, ltstr> B(b, b + N);
  set<const char*, ltstr> C;

  cout << "Set A: ";
  copy(A.begin(), A.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
  cout << "Set B: ";
  copy(B.begin(), B.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;

  cout << "Union: ";
  set_union(A.begin(), A.end(), B.begin(), B.end(),
            ostream_iterator<const char*>(cout, " "),
            ltstr());
  cout << endl;

  cout << "Intersection: ";
  set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                   ostream_iterator<const char*>(cout, " "),
                   ltstr());
```

```
  cout << endl;

  set_difference(A.begin(), A.end(), B.begin(), B.end(),
                 inserter(C, C.begin()),
                 ltstr());
  cout << "Set C (difference of A and B): ";
  copy(C.begin(), C.end(), ostream_iterator<const char*>(cout, " "));
  cout << endl;
}
```

3.b.map<Key, Data, Compare, Alloc>

Description

`Map` is a Sorted Associative Container that associates objects of type `Key` with objects of type `Data`. `Map` is a Pair Associative Container, meaning that its value type is pair<`const Key, Data`>. It is also a Unique Associative Container, meaning that no two elements have the same key.
`Map` has the important property that inserting a new element into a `map` does not invalidate iterators that point to existing elements. Erasing an element from a `map` also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

Example

```
#include <map>
#include <iostream.h>
using namespace std;

typedef map<char* const,int> AgeMap;
typedef AgeMap::iterator Iter;

void main()
{
      AgeMap age;

      age["Jack"] = 30;
      age["bill"] = 40;

      int a = age["Jack"];
      int n = age.size();

      age.erase("Jack");

for(Iter i = age.begin(); i != age.end(); i++)
cout<<" age of " << i->first << " is" << i->second<<endl;

      Iter j = age.find("Bill");

if( j != age.end())
cout<<" age of " << j->first << " is" << j->second<<endl;


}
```

Converting between sequences

```
deque<int> d;
vector<int> v1(d.begin(), d.end());
```

3.c.hash<T>

Description

The function object `hash<T>` is a Hash Function; it is used as the default hash function by all of the Hashed Associative Containers that are included in the STL.
The `hash<T>` template is only defined for template arguments of type `char*`, `const char*`, crope, wrope, and the built-in integral types. [1] If you need a Hash Function with a different argument type, you must either provide your own template specialization or else use a different Hash Function.

Example

```
int main()
{
  hash<const char*> H;
  cout << "foo -> " << H("foo") << endl;
  cout << "bar -> " << H("bar") << endl;
}
```

Creating and initializing C++ strings

- Create an empty **string** and defer initializing it with character data
- Initialize a **string** by passing a literal, quoted character array as an argument to the constructor
- Initialize a **string** using '='
- Use one **string** to initialize another

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

int main() {
  string imBlank;
  string heyMom("Where are my socks?");
  string standardReply = "Beamed into deep "
    "space on wide angle dispersion?";
  string useThisOneAgain(standardReply);
}
```

- Use a portion of either a C **char** array or a C++ **string**

- Combine different sources of initialization data using **operator+**
- Use the **string** object's **substr( )** member function to create a substring

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

int main() {
  string s1
    ("What is the sound of one clam napping?");
  string s2
    ("Anything worth doing is worth overdoing.");
  string s3("I saw Elvis in a UFO.");

  // Copy the first 8 chars
  string s4(s1, 0, 8);

  // Copy 6 chars from the middle of the source
  string s5(s2, 15, 6);

  // Copy from middle to end
  string s6(s3, 6, 15);
  // Copy all sorts of stuff
  string quoteMe = s4 + "that" +
  // substr() copies 10 chars at element 20
  s1.substr(20, 10) + s5 +
  // substr() copies up to either 100 char
  // or eos starting at element 5
  "with" + s3.substr(5, 100) +
  // OK to copy a single char this way
  s1.substr(37, 1);
  cout << quoteMe << endl;
}
```

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

int main() {
```

```
 string source("xxx");
 string s(source.begin(), source.end());
 cout << s << endl;
}
```

Appending, inserting and concatenating strings

```
#include <string>
#include <iostream>

using namespace std;
int main()
{
      string bigNews("I saw Elvis in a UFO. ");

      cout << "Size = " << bigNews.size() << endl;
      cout << "Capacity = " << bigNews.capacity() << endl;
      bigNews.insert(1, " thought I ");

      cout << "Size = " << bigNews.size() << endl;
      cout << "Capacity = " << bigNews.capacity() << endl;

      bigNews.reserve(500);

      bigNews.append("I've been working too hard.");

      }
```

Replacing string characters

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

int main() {
  string s("A piece of text");
  string tag("$tag$");
  s.insert(8, tag + ' ');
  cout << s << endl;
  int start = s.find(tag);
  cout << "start = " << start << endl;
  cout << "size = " << tag.size() << endl;
  s.replace(start, tag.size(), "hello there");
  cout << s << endl;
}
```

for more details on Container classes http://www.sgi.com/Technology/STL/.