ENCAPSULATION

make user-defined data types
the same as builtin data types

What Is Encapsulation?

- Combine a data structure and its associated operations into a unit.
- The data structure is protected and cannot be directly accessed.
- The data structure can only be accessed through a group of publicized operations.

How to Implement Encapsulation?

- Use the `class` construct to combine a data structure and its associated operations.
- Use the `private` section to prohibit the access of the data structure.
- Use the `public` section to publicize the associated operations.

```
class Complex
{
public :
   void  initComplex(double r, double i)  { … }
   Complex  add(Complex c)  { … }
   Complex  sub(Complex c)  { … }
     …
private :
   double  r;
   double  i;
} ;
```

member functions

member variables

```
Complex  c1, c2, c3;
c1 = initComplex(2, 2);
c2 = initComplex(4, 4);
c3 = c1.add(c2);
```

c1 has an associated operation add

Interface and Implementation

- Encapsulation emphasizes the separation of interface and implementation.
- The interface contains only information needed for using a data type.
- The implementation contains all the details of implementing a data type.

```
// complex.h  -- The interface
class Complex
{
public :
   void  initComplex(double, double) ;
   Complex  add(Complex) ;
   Complex  sub(Complex) ;
      …
private :
   double  r;
   double  i;
} ;
```

Why are private data contained in the interface?

```
// complex.cpp  --  The implementation
#include “complex.h”
void  Complex::initComplex(double re, double im)  { … }
Complex  Complex::add(Complex c)  { … }
Complex  Complex::sub(Complex c)  { … }
     …
```

Advantages of Information Hiding

- The interface serves as a contract between the users and the implementer of the type.
- The users need to know as few information as possible in order to use the type.
- The changes of implementation of the type do not affect the use of the type.
- The reliability of the publicized operations guarantees the reliability of the type.

Initialization of Variables

- How are variables of builtin data types initializaed?
- int n = 7;
- int a[ ] = {1, 2, 3, 4, 5, 6, 7};
- Complex c = {7, 7};

Constructors

- A constructor is a member function that is automatically called when an object is created.
- A constructor is used to initialize the values of member variables and other sort of initialization.

```
// complex.h  -- The interface
class Complex
{
public :
   Complex(double, double) ;
   Complex  add(Complex) ;
   Complex  sub(Complex) ;
      …
private :
   double  r;
   double  i;
} ;
```

The name of a constructor is the same as the name of class!

A constructor does not have return type!

```
Complex  c1(2, 2), c2(4, 4), c3;
c3 = c1.add(c2);
```

```
// complex.cpp  --  The implementation
#include "complex.h"
Complex::Complex(double re, double im)  { r = re; i = im; }
Complex  Complex::add(Complex c)  { … }
Complex  Complex::sub(Complex c)  { … }
     …
```

The Member Initialization List

```
// complex.cpp  --  The implementation
#include "complex.h"
Complex::Complex(double re, double im)
{
   r = re;
   i = im;
}

// complex.cpp  --  The implementation
#include "complex.h"
Complex::Complex(double re, double im)
   : r(re), i(im)            This version is preferable
{
}
```

Multiple Constructors

- A class may have more than one constructor.
- The constructor that takes no parameters is called the default constructor.
- Each class should always include the default constructor.

```
// complex.h  -- The interface
class Complex
{
public :
   Complex( ) ;
   Complex(double) ;
   Complex(double, double) ;
      ...
private :
   double  r;
   double  i;
} ;


Complex  c1, c2(2), c3(4, 4);
```

```
// complex.cpp  --  The implementation
#include "complex.h"

Complex::Complex( )
   : r(0), i(0)
{ }

Complex::Complex(double re)
   : r(re), i(0)
{ }

Complex::Complex(double re, double im)
   : r(re), i(im)
{ }
```

The default constructor

No parentheses for the default constructor

Classes with Dynamic Data Structures

- A class may have a member variable that is a pointer to a dynamic data structure.
- The initialization of such member variables involves dynamic memory allocation.

```
// String.h  -- The interface
class String
{
public :
   String( ) ;
   String(char *) ;
      ...
private :
   char *  s;
} ;


String  s1, s2("Hello World!");
```

```
// String.cpp  --  The implementation
#include "String.h"

String::String( )
{
   s = new char[1];
   s[0] = '\0';
}

String::String(char * str)
{
   if (str) {
      s = new char[strlen(str) + 1];
      strcpy(s, str);
   } else {
      s = new char[1];
      s[0] = '\0';
   }
}
```

Destructors

- A destructor is a member function that is called automatically when an object is destroyed.
- A destructor is used to delete memory that is dynamically allocated when the object is created and other sort of clean-up tasks.
- A class should define its own destructor if it has pointer member variables.

```
// String.h  -- The interface
class String
{
public :
   String( ) ;
   String(char * str) ;
   ~String( );
      …
private :
   char *  s;
} ;

void foo( )
{
   String  s("Hello World!");
      …
}
```

```
// String.cpp  --  The implementation
#include "String.h"

String::~String( )
{
   delete [ ] s;
}
```

The destructor avoids memory leak!!

Copy Constructors

- A copy constructor is a constructor that has one parameter that is of the same type as the class.
- The parameter must be a call-by-reference parameter.
- The parameter is normally a constant parameter.

```
// String.h  -- The interface
class String
{
public :
   String( ) ;
   String(char *) ;
   String(const String &) ;
      ...
private :
   char *  s;
} ;
```

```
// String.cpp  --  The implementation
#include "String.h"

String::String(const String & str)
{
   s = new char[strlen(str.s) + 1];
   strcpy(s, str.s);
}
```

Create a new copy!

```
String  s1, s2("Hello World!");
String  s3(s2);
String  s4 = s2;
```

These two forms are the same!

The Copy Constructor Is Called Automatically When

- When a class object is being declared and is initialized by another object of the same type.
- Whenever an argument of the class type is passed as a call-by-value argument.
- When a function returns an object of the class type.

```
#include "String.h"

void showString(String str)
{
   cout << "The string is " << str << endl;
}

{
   String  s("Hello World!");
```

The default copy constructor performs memberwise copies!
Hence, at the beginning of the call, str and s point to the same string!

```
   showString(s);
```

The destructor of str deletes the string of str (and s) after the call!

```
}
```

The destructor of s tries to delete the string of s again!