Computer Programming

Class

Class

- Class
  - Defining a new data type and the associated operations (how the new data type is manipulated)
  - Grouping of data and functions
  - ☞ Different data types may have different ways of operations specific to themselves

```
/ produces different results for different types of
  operands (int vs. double)
```

```
The way to calculate the "area of a square" is
different from calculating the "area of a circle"
```

  - ☞ `struct` is in fact a *special case* of `class` in C++

Class Members

The defined function becomes "local member" of the class – not a **global function** as we have seen before

- Members
  - Data members (member data)
  - Function members (member functions)
- Class definition: a simplistic way

```
class class_name
{
  public:
    data_member_definitions;
    function_member_definitions;
};
```

public is an **access specifier**, meaning the following members are accessible to the public (i.e. to a statement outside the class)

☞ A variable of the type "`class_name`" is called an object, or an instance, of the class "`class_name`"

Rational in Class

```
Write two global functions instead?
void set(rational &, int, int);
void show(rational);
```

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

class rational {
  public:
    int n, d;
    void set(int x, int y) { n = x; d = y; }
    void show() { cout << n << "/" << d << "=" << 1.0*n/d << endl; }
};

int main( )
{
    rational a;
    a.set(4, 5);
    a.show();

    rational *p = &a;
    p->set(2, 3);
    p->show();
}
```

can directly access members **without relying on any handle**

```
void set(rational &r, int x, int y)
{ r.n = x; r.d = y; }
```

```
void show(rational r)
{ cout<< r.n << "/" << r.d << "="
      << 1.0*r.n/r.d << endl; }
```

```
set(a, 4, 5);
show(a);
```

```
(*p).set(2, 3);
(*p).show();
```

```
set(*p, 4, 5);
show(*p);
```

Object Handle

- Class is a *data type*
  - Class can be used in object, array, pointer, and reference

```
rational w[5];
rational &y = x;
rational *z = &x;
```



```
w[0].show();
   y.show();
  z->show();
```

```
   w->show();
(&y)->show();
 (*z).show();
```

  - Default member-wise assignment

```
rational v;
v = x;
```

```
Each data member in x is
assigned individually to
the same data member in v
```

  - Member selection operator
    - Dot (.) is preceded by an object or reference to an object
    - Arrow (→) is preceded by a pointer to an object
  - Object *handle*
    - Public class members are accessed through one of the handles on an object – an object name itself, a reference to an object, or a pointer to an object

Operator Precedence

Operators							Associativity	Type
()	[]	.	->				left to right	
++	--	static_cast<*type*>()					left to right	unary (postfix)
++	--	+	-	!	&	*	right to left	unary (prefix)
*	/	%					left to right	multiplicative
+	-						left to right	additive
<<	>>						left to right	insertion/extraction
<	<=	>	>=				left to right	relational
==	!=						left to right	equality
&&							left to right	logical AND
||							left to right	logical OR
?:							right to left	conditional
=	+=	-=	*=	/=	%=		right to left	assignment
,							left to right	comma

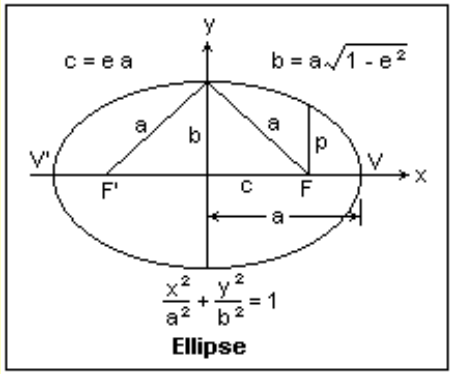
A New Class: Ellipse (1/2)

Resort to slides on installing SDL_BGI (library) if you have problems with this code

```
#include <iostream>
#include <cmath>
#include <graphics.h>
using namespace std;
class elipso
{
  public:
    int x, y, a, b, c;
    void initialize(int x0, int y0, int a0, int b0) {
        x=x0; y=y0; a=a0; b=b0;              // set center & semi-axes
        c=sqrt(fabs(a*a-b*b));
        show();
    }
    void show() {
        ellipse(x, y, 0, 360, a, b);         // draw a complete ellipse
        circle(x+c, y, 1);                   // mark the positive focus
        circle(x-c, y, 1);                   // mark the negative focus
    }
    double area() { return 3.14159*a*b; }
};
```

For SDL_BGI

Use #include "graphics.h" if it is in the local directory



Write `initialize()` and `area()` as two global functions outside class?

A New Class: Ellipse (2/2)

```
int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();

    elipso e;
    e.initialize(100, 100, 50, 30);
    cout << "The area is " << e.area() << endl;

    elipso *p = &e;
    cout << "The center is at (" << p->x << ", " << p->y << ")" << endl;
    getch(); closegraph();
}
```

Use SDL_BGI to create a window for plotting

A function can call initialize() only through a handle of the object

Use SDL_BGI to wait and close the window

- The ellipse is initialized by setting the major and minor axes, but the *focus* c is calculated internally
- All data members of the class are subject to direct access or modification *by code with a handle to the object*

Encapsulation

Encapsulation also has the advantage of allowing **interface** and **implementation** of a class to be decoupled (e.g. version upgrade)

- Manipulation of data members
  - *Developer* of the class vs. *user* of the class
  - A developer may want to avoid *careless or invalid operations* by users on members of the class
  - Encapsulation (hiding) of "internal data" to prevent inadvertent modification from "outside" the class
  - ☞ Any C++ statement that requires an *object handle* to access the member can be considered as "outside" the class
  - With encapsulation, getting an object handle does not necessarily mean having access to the object member

Ellipse – Encapsulation (1/3)

```
#include <iostream>
#include <cmath>
#include <graphics.h>
class elipso
{
  private:
    int x, y, a, b, c;
  public:
    void initialize(int, int, int, int);
    void show();
    double area() {return 3.14159*a*b;}
    double get_major() {return 2*(a>b?a:b);}
    double get_minor() {return 2*(a>b?b:a);}
    void   get_center(int *x0, int *y0) {
           *x0=x; *y0=y;}
    void   set_major(int);
    void   set_minor(int);
    void   set_center(int x0, int y0) {
           x=x0;y=y0;}
};
```

private is an access specifier, meaning the following members are accessible only to **member functions of this class**

Class members by default (without any preceding access specifier) are private

Allow read access of members (**get** interface)

Allow write access of members (**set** interface)

Ellipse – Encapsulation (2/3)

```
void elipso::show()
{
    ellipse(x, y, 0, 360, a, b);
    circle(x+c, y, 1);
    circle(x-c, y, 1);
}
void elipso::initialize(int x0, int y0, int a0, int b0)
{
    x=x0; y=y0; a=a0; b=b0;
    c=sqrt(abs(a*a-b*b));
    show();
}

void elipso::set_major(int x0)
{
    if (x0<get_minor()) return;
    if (a>b) a=x0/2;
    else     b=x0/2;
    c=sqrt(abs(a*a-b*b));
}
```

A member function can be defined outside the class definition, but its scope is still within the class – can be accessed only through an object handler

Note that **void show() {…}** without **elipso::** is to define a "global" function

Ellipse – Encapsulation (3/3)

```
void elipso::set_minor(int y0)
{
    if (y0>get_major()) return;
    if (a>b) b=y0/2;
    else     a=y0/2;
    c=sqrt(abs(a*a-b*b));
}

int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();

    elipso e;
    e.initialize(100, 100, 50, 30);
    e.set_major(80);
    e.set_center(200, 200);
    e.show();
    getch(); closegraph();
}
```

Users of the class (outside the class) cannot change the value of e.c now

More on Encapsulation

- The `friend` function and class
  - A friend function of a class is defined outside that class's scope, but has the right to access non-public (e.g. private) members of the class
  - A friend class of another class has the property that all its member functions can access non-public members of that class
  - ☞ Enhance usability while enforcing encapsulation
  - ☞ Declaration of friendship grants access
    - By declaring that class B is its friend, class A grants class B the right to access its private members (not the other way around)

```
class A {
  friend void function(A &);
  ...
```

Ellipse – `friend` (1/2)

The function `scale()` is now a **friend** of class elipso

```
#include <iostream>
#include <cmath>
#include <graphics.h>

class elipso
{
  friend void scale(elipso &, int);
  private:
    int x, y, a, b, c;
  public:
    void initialize(int, int, int, int);
    …
    (Reuse the class definition on p. 24 here)

};
```

Insert
`friend class C;`
to allow member functions in C to access members x, y, a, b, and c freely

```
class C
{
  double eccentric(
          elipso *p) {
    return p->c*1.0/p->a;
  }
  ...
};
```

- Friendship is not symmetric nor transitive
  - That class A is a friend of class B does not *infer* the reverse
  - A as a friend of B and B of C do not mean A is a friend of C

Ellipse – `friend` (2/2)

Do not forget to copy the class definition on pp. 25-26 to complete the class definition

```
void scale(elipso& e, int s)
{
    if (s<=0) return;
    e.a *= s;
    e.b *= s;
    e.c=sqrt(abs(e.a*e.a-e.b*e.b));
}

int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();

    elipso e;
    e.initialize(100, 100, 50, 30);
    scale(e, 2);
    e.set_center(200, 200);
    e.show();
    getch(); closegraph();
}
```

The **friend** function scale() can access private members x, y, a, b, and c freely

It is still necessary to get a **handle** of the class since scale() is not a class member function (cf. in initialize())

Note the handle needs to be passed as a **reference** instead of pass-by-value

Constructor

It is possible to call a constructor **explicitly** to create a temporary **unnamed object** (e.g. used for function argument or return by value) such as `elipso();` but it is wrong to call a constructor on an already-existing object such as `z.elipso(5);`

- Initialization
  - When objects of a class are created, it is desirable to *properly initialize* their initial states (e.g. set or reset meaningful values of its data members)
  - ☞ C++ provides a mechanism to do this automatically
- Constructors
  - Special functions used to initialize an object's data when the object is created
  - ☞ The constructor is called automatically when the object is created (no explicit call by the user is needed)

```
ClassName::ClassName(argument_list);
```

The constructor must be defined with the same name as the class name

```
The constructor cannot return
any value (not even "void" here)
```

Ellipse – Constructor (1/2)

```
elipso k(5), h = 5;
```

```
#include <iostream>
#include <cmath>
#include <graphics.h>

class elipso
{
  private:
    int x, y, a, b, c;

  public:
    elipso(int, int, int, int);
    elipso(int r) {x=y=r; a=b=r; c=0; show("1");}
    elipso() {x=y=a=b=10; c=0; show("0");}
    void show(const char *s=NULL);
    double area() {return 3.14159*a*b;}
};
```

Constructor with a single argument can also be used for type conversion (e.g. convert int to elipso)

Compiler can call conversion constructor **automatically** for implicit type conversion (e.g. argument coercion)

Constructor overloading

***can be** combined into:*

```
elipso(int r=10) {…}
```

- Which constructor is called depends on the way the object is created (e.g. the parameter list specified in the declaration of an object)

Ellipse – Constructor (2/2)

The compiler will not create a default constructor if there are **other constructors** already defined in the class

```
elipso::elipso(int x0, int y0, int a0, int b0)
{
    x=x0; y=y0; a=a0; b=b0;
    c=sqrt(abs(a*a-b*b));
    show("4");
}
void elipso::show(const char *s)
{
    ellipse(x, y, 0, 360, a, b);
    circle(x+c, y, 1);
    circle(x-c, y, 1);
    if (s) outtextxy(x-textwidth(s)/2, y-textheight(s), s);
}
int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();
    elipso x(100, 100, 50, 30), y(50), z;
    std::cout<<"The area of y is "<< y.area() <<std::endl;
    getch(); closegraph();
}
```

A default constructor is one with no arguments. It will be provided by the compiler if needed and there is no constructor in the class (but **no initialization** of the data).

Which constructor is called for "elipso z;"?

Destructor

The compiler provides an empty destructor if not specified by the programmer

- Destructor

```
ClassName::~ClassName();
```

Objects in the same scope & storage class are **destructed in the reverse order** that they are created

  - No arguments and no return type
  - Called implicitly when an object is destroyed
  - Only one destructor per class (*cf.* constructor)
- Object construction and destruction
  - An object is created/destroyed when program execution enters/leaves its scope (*function* prologue and epilogue)
    - Global scope: constructor called before any other function; destructor called after main() terminates
    - Static local storage: constructor called once where the object is defined; destructor called after main() terminates

Ellipse – Object Order (1/2)

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

class elipso
{
  public:
    elipso(int);
    ~elipso();
  private:
    int id;
};
elipso::elipso(int i)
{
    id = i; cout << "Object ID = " << id << " constructor called"
                 << endl;
}
elipso::~elipso()
{
    cout << "Object ID = " << id << " destructor called" << endl;
}
```

Ellipse – Object Order (2/2)

```
elipso first(1);
void func();

int main()
{
    cout << "main() execution begins" << endl;
    elipso second(2);
    static elipso third(3);
    func();
    cout << "main() execution resumes" << endl;
    elipso fourth(4);
    cout << "main() execution ends" << endl;
}
void func()
{
    cout << "func() execution begins" << endl;
    elipso fifth(5);
    static elipso sixth(6);
    cout<< "func() execution ends" << endl;
}
```

Do not use `system("pause");` for running the program here – Run the program in the command-line environment if needed

```
Object ID = 1 constructor called
main() execution begins
Object ID = 2 constructor called
Object ID = 3 constructor called
func() execution begins
Object ID = 5 constructor called
Object ID = 6 constructor called
func() execution ends
Object ID = 5 destructor called
main() execution resumes
Object ID = 4 constructor called
main() execution ends
Object ID = 4 destructor called
Object ID = 2 destructor called
Object ID = 6 destructor called
Object ID = 3 destructor called
Object ID = 1 destructor called
```