Dynamic Allocation of Objects

- Dynamic allocation and de-allocation of memory
  - Use `new` and `delete` to dynamically allocate and de-allocate the required memory for objects
  - `new` allocates memory of the proper size, calls the *default constructor*, and then returns the pointer

```
elipso *p;
p = new elipso;
```

The corresponding constructor is called for initialization of the elipso object

```
elipso *h = new elipso(10, 10, 5, 3);
```

  - `delete` calls the destructor of the object, then frees the memory associated with the object

```
delete p;
```

Note that "delete" can be used only on objects created by "new" (no ordinary objects)

More on Constructor (1/2)

The compiler may use "return value optimization" (RVO) to avoid making a copy in function return

```
int main()
{
    initwindow(800, 600); cleardevice();
    elipso x(100, 100, 50, 30), y(50), z(x);
    std::cout<< " The area of z is " << z.area() << std::endl;
    getch(); closegraph();
}
```

Following the class definition on slide 31

Which constructor is called to initialize z here?

- Copy constructor
  - A special constructor used to create a new object from an existing object (copy of existing object)
  - The copy constructor is called when
    - An object is initialized (constructed) from another object
    - An object is *passed by value* to a function
    - An object is *returned by value* from a function (a usual case)

More on Constructor (2/2)

What happens if the argument of a copy constructor is not a reference?

```
ClassName::ClassName(const ClassName &);
ClassName::ClassName(const ClassName &, default_argument_list);
```

- ☞ The first argument is a **reference** to an object of the same type as the one being constructed

- Default copy constructor
  - The compiler automatically creates a copy constructor for each class if needed (known as a *default copy constructor*) that performs member-wise "shallow" copy

```
elipso x(5, 3), u = x;
func1(elipso x);
elipso y = func2(void);
```

Equal to `elipso u(x);`

Different from `elipso u; u=x;`

- ☞ Copy constructor vs. assignment operator (more on this later)

Ellipse – Copy Constructor (1/2)

```
#include <iostream>
#include <cmath>
#include <graphics.h>
class elipso
{
  private:
    int x, y, a, b, c;
  public:
    elipso(int, int, int, int);
    elipso(const elipso &x);
    void show(const char*s=NULL);
    double area() {return 3.14159*a*b;}
};
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);
}
```

Ellipse – Copy Constructor (2/2)

```
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");
}
elipso::elipso(const elipso &e)
{
    x=e.x; y=e.y;
    a=e.a; b=e.b; c=e.c;
    show("copy");
}

int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();
    elipso x(100, 100, 50, 30), y=x, &z=y;
    std::cout << "The area of y is " << y.area() << std::endl;
    getch(); closegraph();
}
```

Invoked through `elipso y=x;` or `elipso y(x);`

Member-wise copy

No need to create a user-defined copy constructor if **member-wise copy is sufficient** for the class (the compiler will create one such if needed)

No object construction for z

Ellipse – Object Order Take Two (1/2)

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

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

    ~elipso() {
      cout << "Object ID = " << id << " destructor called" << endl;}

    elipso(const elipso &x) {
      id = -1*x.id;
      cout << "Object ID = " << id << " copy constructor called "
      "(from Object ID = " << x.id << ")" << endl;}

  private:
    int id;
};
```

Ellipse – Object Order Take Two (2/2)

```
void func(elipso x)
{
    cout << "func() execution begins" << endl;
    elipso fifth(5);
    cout<< "func() execution ends" << endl;
}
int main()
{
    cout << "main() execution begins" << endl;
    elipso second(2);
    func(second);
    cout << "main() execution ends" << endl;
}
```

```
main() execution begins
Object ID = 2 constructor called
Object ID = -2 copy constructor called (from Object ID = 2)
func() execution begins
Object ID = 5 constructor called
func() execution ends
Object ID = 5 destructor called
Object ID = -2 destructor called
main() execution ends
Object ID = 2 destructor called
```

In call-by-value, a copy of the argument is created in the function stack frame for func() to access

Member Initializer

- Member initializer (member initialization list)

```
elipso::elipso(int x0, int y0, int a0, int b0)
        : x(x0), y(y0)
{
    a = a0; b = b0;
}
```

Initialize **x** based on **x0** (copy constructor)

Assign **b** to a new value **b0** after it is constructed

  - All data members *can be* initialized using the member initializer
    - ☞ Multiple member initializers are separated by commas
    - ☞ Data members are constructed and initialized in the order in which they are declared in the class definition (not their presence and order in the member initialization list)
  - Member initializers execute before the body of the constructor executes
    - Members are initialized *before* initialization of the host object

Object Inside an Object

We can say that a new object is **composed** using existing objects

- Composition
  - An object can be declared within another class (member object vs. host object)
  - Member objects are constructed in the order in which they are declared in the class definition, and **before** the constructor body of the host object is executed
    - If a member object is not explicitly initialized, its default constructor will be called implicitly
    - If a member object is to be *explicitly initialized* (by a proper constructor), it **must** be done in the member initializer
  - The destructor of a member object is called **after** the destructor of the host object is executed, and in the reverse order from which they are constructed

Ellipse – Member Object (1/3)

```
#include <iostream>
#include <cmath>
using namespace std;
class point
{
    int x, y;
  public:
    point(int i=0, int j=0);
    point(const point& p);
    ~point();
};
point::point(int i, int j) : x(i), y(j)
{
    cout<< "constructor for point (" << x << ", " << y << ")" <<endl;
}
point::point(const point& p) : x(p.x), y(p.y)
{
    cout<< "CONSTRUCTOR for point (" << p.x << ", " << p.y << ")" <<endl;
}
point::~point()
{
    cout<< "destructor for point (" << x << ", " << y << ")" <<endl;
}
```

Ellipse – Member Object (2/3)

```
class elipso
{
  private:
    int a, b, c;
    point center;
  public:
    elipso(point o, int a0, int b0);
    ~elipso();
};
elipso::elipso(point o, int a0, int b0)
        : center(o), a(a0), b(b0), c(sqrt(abs(a*a-b*b)))
{
    cout << "constructor for elipso"<< endl;
}

elipso::~elipso()
{
    cout << "destructor for elipso" << endl;
}
```

Ellipse – Member Object (3/3)

```
int main()
{
    point p(100, 100);
    elipso z(p, 50, 30);
}
```

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

```
constructor for point (100, 100)
CONSTRUCTOR for point (100, 100)
CONSTRUCTOR for point (100, 100)
constructor for elipso
destructor for point (100, 100)
destructor for elipso
destructor for point (100, 100)
destructor for point (100, 100)
```

☞ Try call-by-reference (no new object "copy")

```
elipso::elipso(const point &o, double x, double y) {…}
```

Class and const

A constant data is in fact "not constant" across the class – it is **constant only for an instance**

- Constant member

To have a data member that is the same for all instances of the class, use **"static"**

  - Constant member data
    - ☞ const data members (and references) must be initialized using member initializers – instead of being assigned values in the constructor body

```
class elipso
{
    int x, y, a, b, c;
    const double PI;
  public:
    elipso() : PI (3.14159) {}
    double area() const;
};
```

C++11 allows a non-static data member to be initialized where it is declared (in its class)

It is wrong to use:
const double PI=3.14159;
in a class definition

A constant function is defined only if it is a **member function**

  - Constant member function
    - const member functions cannot modify data members of an object

Class and `const` (cont.)

- Constant object

```
const elipso e;
```

  - An object of which the data members cannot be modified
  - Member functions for `const` objects cannot be called unless these functions are declared `const`
    - No modification of the data members through the `const` member function
      - `const` member functions cannot call non-const member functions of the class on the same instance
  - An exception: constructors and destructor of a `const` object are still automatically executed to initialize the object without the above constraint

Ellipse – `const` Operator (1/2)

```
#include <iostream>
#include <cmath>
#include <graphics.h>
using namespace std;
class elipso
{
  int x, y, a, b, c;
  const double PI;
  public:
    elipso(int r) : PI(3.14) {x=y=r; a=b=r; c=0;}
    elipso(int, int, int, int);
    double area() const {return PI*a*b;}
    void show();
};
void elipso::show()
{
    ellipse(x, y, 0, 360, a, b);
    circle(x+c, y, 1);
    circle(x-c, y, 1);
}
```

A constant data member should be initialized in the constructor

Initialization of the const data member

A const member function (specified **both** in function declaration and definition)

Ellipse – `const` Operator (2/2)

```
elipso::elipso(int x0, int y0, int a0, int b0) : PI(3.14159)
{
    x=x0; y=y0; a=a0; b=b0;
    c=sqrt(abs(a*a-b*b));
}
int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();
    elipso x(200, 200, 50, 30);
    const elipso y(50);
    x.show();                          // non-const obj non-const func
    cout << x.area() << endl;          // non-const obj const func
    getch(); closegraph();
}
```

PI ***can be*** initialized different values in different constructors if needed

What happens if **x** is passed to a function like: **void show(const elipso&)**?

```
    y.show();                          // const obj non-const func (wrong)!!
    cout << y.area() <<endl;           // const obj const func
```

☞ What happens if show() is called inside the constructor?

`this` Pointer

```
Individual objects contain only data
        (not including the functions)
sizeof(rational)=8, sizeof(elipso)=20
```

- Member function
  - Member functions are shared by all objects of the class
  - ☞ How do member functions know which object's data to manipulate?
- `this` pointer
  - Compiler passes *an implicit argument* called `this` to the object's (non-static) member functions
  - `this` pointer is of type `class_name * const` for non-constant object
    - A constant pointer to the current object of type `class_name`
  - `this` points to the address of the object handle so the function can use it to access the correct object data

Ellipse – Using `this`

Recall the case when `initialize()` and `area()` are written as two global functions (an object handle is needed)

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

class elipso
{
  public:
    int x, y, a, b, c;
    double area()
    {
      const double PI=3.14159;
      return PI*a*b;
    }
};
```

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

class elipso
{
  public:
    int x, y, a, b, c;
    double area()
    {
      const double PI=3.14159;
      return PI*this->a*this->b;
    }
};
```

If **e** is an elipso object, in the call **e.area()**, **this** is set to **&e** for **area()** to access the data of **e**

In each member function, **this** is used **implicitly** (by compiler) to access the correct data member

Similarly, in the call **f.area()** to object **f**, **this** is set to **&f**

The programmer does not need to explicitly use **this** in this case

Explicit use of `this` Pointer

- Cascade function call
  - Multiple function calls can be invoked in one statement

```
object.func1().func2().func3();
```

- Implementation using `this`

Recall *return by reference*

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

Replace the function set_major() on p. 25

Rewrite the function set_minor() similarly

Return the original object (*this), instead of a **copy of the object**

```
elipso e1, e2;
e1.set_major(50).set_minor(30);
e2.set_minor(30).set_major(50);
```

In e1.set_major(50), **this** is **&e1**

In e2.set_minor(30), **this** is **&e2**

Class and `static`

```
C++11 allows a non-static data
member to be initialized where it
is declared (in its class)
```

- Static data members
  - Each object of a class has its own copy of the data members, except for static data members
    - A static data member is *shared by all objects* of a class
    - A static data member can be considered as "class-wide" information (aka. class variable)
    - ☞ Example: the radius of a circle depends on the number of circles over a specified region
  - A static data member must be defined and initialized at file scope (*outside* the class definition) (C++98)
    - An exception is `const static` data member of **integral** type that can be initialized in the class definition
  - ☞ A class's static members exist even when no objects of that class exist (it is not associated with any object)

static Member (1/2)

```
class elipso {
  const static char tag ='C';
};
//const char elipso::tag;
```

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <graphics.h>
using namespace std;
class elipso
{
  int r;
  const double PI;
  static int count;
  public:
    elipso(int x);
    static int get_count() {return count;}
};
int elipso::count = 0;
elipso::elipso(int x) : PI(3.14159)
{
    count++; r=x/count ;
    circle(rand()%getmaxx(), rand()%getmaxy(), r);
}
```

A member function that does not access non-static data members can be declared as static

`static` Member (2/2)

Visibility of the static data member is still restricted by the access specifier in the class

```
int main(int argc, char*argv[])
{
    initwindow(800, 600); cleardevice();
    cout << "# of circles before is " << elipso::get_count() << endl;
    elipso *c1 = new elipso(100);
    elipso *c2 = new elipso(100);
    elipso *c3 = new elipso(100);
    cout << "# of circles is " << c1->get_count() << endl;
    delete c1;
    delete c2;
    delete c3;

    cout << "# of circles after is " << elipso::get_count() << endl;
    getch(); closegraph();
}
```

Cannot write `c1->count` or `elipso::count` since **count** is private

- A static member function can be invoked even without any object handle
- A static member function does not have `this` pointer

Review

- `class`
  - Grouping of data and function members
  - Use explicit `private` and `public` to achieve the desired encapsulation
  - Friend functions and classes
  - Class constructor and destructor
  - Member initialization list
  - Object as class members
  - Class and `const`
  - Static class members