Classes

Section 1D

What is a C++ `class`?

Encapsulates a Sales Person properties and methods

```
class SalesPerson
{
private:                                  Members accessible ONLY within the class
    double sales[12];                 // 12 monthly sales figures.
    double totalAnnualSales();        // Utility function.

public:                                   Members accessible from everywhere, even outside the class
    SalesPerson();                    // Constructor.
    void setSales();                  // User supplies sales figures.
    void printAnnualSales();
};
```

Constructors

- A **constructor** is a member function whose name is the same as the class' name.
- Every time an object is created, its class constructor is invoked ***automatically***. Its job is to initialize the object appropriately.

```
class SalesPerson
{
private:

   double sales[12];
   double totalAnnualSales();

public:

   SalesPerson();
   void setSales();
   void printAnnualSales();
};
```

```
SalesPerson::SalesPerson()
{
   for( int I = 0; I < 12; I++ )
      sales[I] = 0;
}
```

Constructors

- We can have more than one constructor for a class (as long as they have different signature).
- How can we define a constructor that gets as input an array of double values?

```
class SalesPerson
{
private:
   double sales[12];
   double totalAnnualSales();

public:
   SalesPerson();
   SalesPerson( const double* s )
   void setSales();
   void printAnnualSales();
};
```

```
SalesPerson::SalesPerson()
{
   for( int I = 0; I < 12; I++ )
      sales[I] = 0;
}

SalesPerson::SalesPerson( const double* s )
{
   for( int I = 0; I < 12; I++ )
      sales[I] = s[I];
}
```

How do we create objects?

```
SalesPerson s1;
s1.setSales();
s1.printAnnualSales();
```

Creating an object by using the default (empty-argument) constructor.
No parentheses after the object's name!!!

```
double arr[] = {
   1, 2, 3,  4,  5,  6,
   7, 8, 9, 10, 11, 12 };
SalesPerson s2( arr );
s2.printAnnualSales();
```

Creating an object by using a constructor with arguments.
Pass arguments between parentheses after the object's name!!!

Recall to use the **dot operator** to access members in the class if the left-hand side is an *object*, or the **arrow operator** if the left-hand side is a *pointer to an object*.

Exercise 1

- Create a class called `Complex` to execute complex-number arithmetic. A complex number is represented as

  $$realPart + imaginaryPart * i$$

- Write a member function that adds another `Complex` number to the caller object and returns the result.
- Write a member function that subtracts another `Complex` number from the caller object and returns the result.
- Write a member function that prints out the `Complex` number in the format `(realPart, imaginaryPart)`

The `const` function qualifier

- If an object is `const`, it is only allowed to use member functions that promise not to modify the object contents. This functions must be declared and defined as `const`.

```
/**
 * @brief Printing the complex number.
 */
void print() const
{
   cout << "(" << realPart
   << ", " << imaginaryPart << ")" << endl;
}
```

```
int main()
{
   const Complex c2( 3, 1 );

   c2.print();
}
```

The `this` pointer

- Every C++ object has a pointer to itself, called `this`.
- The this pointer is used to implicitly reference all function and data members in an object. It can be used explicitly to differentiate function parameters that have the same name as the object data members.

```
/**
 * @brief Constructor with arguments.
 */
Complex( double realPart, double imaginaryPart )
{
   this->realPart = realPart;
   (*this).imaginaryPart = imaginaryPart;
}
```

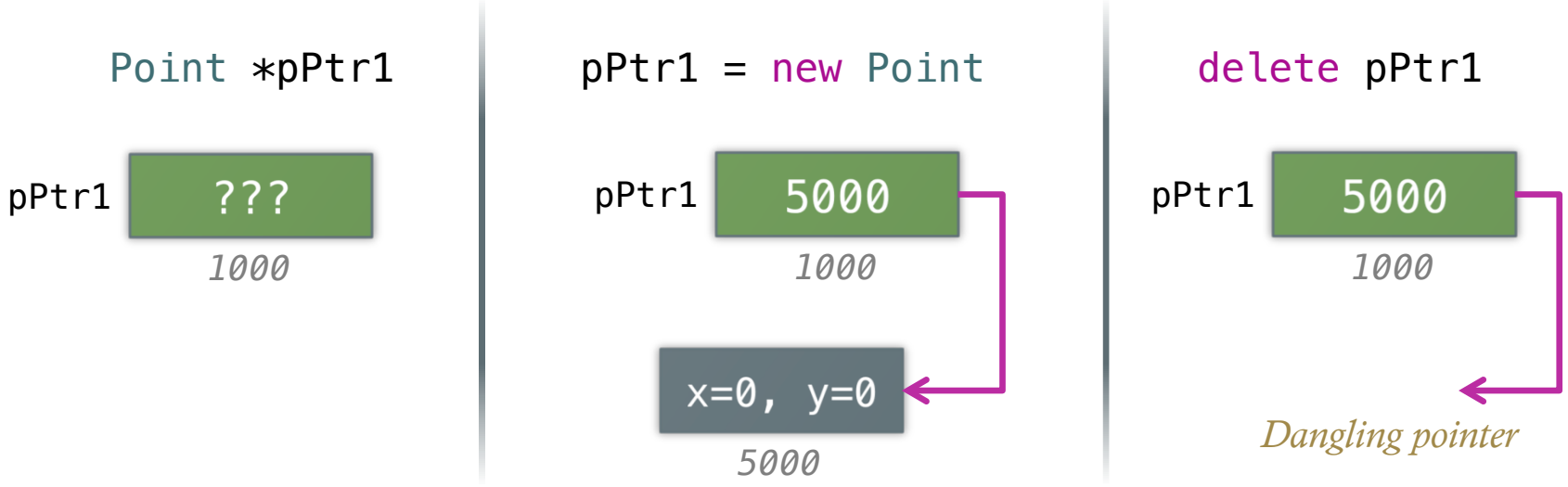
Exercise 2

- Write a class that represents a `Point` in 2D, with **public** coordinates `x` and `y`. It should have
  - A default constructor that sets `x` and `y` to zero.
  - A constructor that takes 2 arguments, named `x` and `y`, for both of the object coordinates, respectively.
  - A function that receives as input another 2D `Point` and computes the distance with the caller `Point` object.

$$distance = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

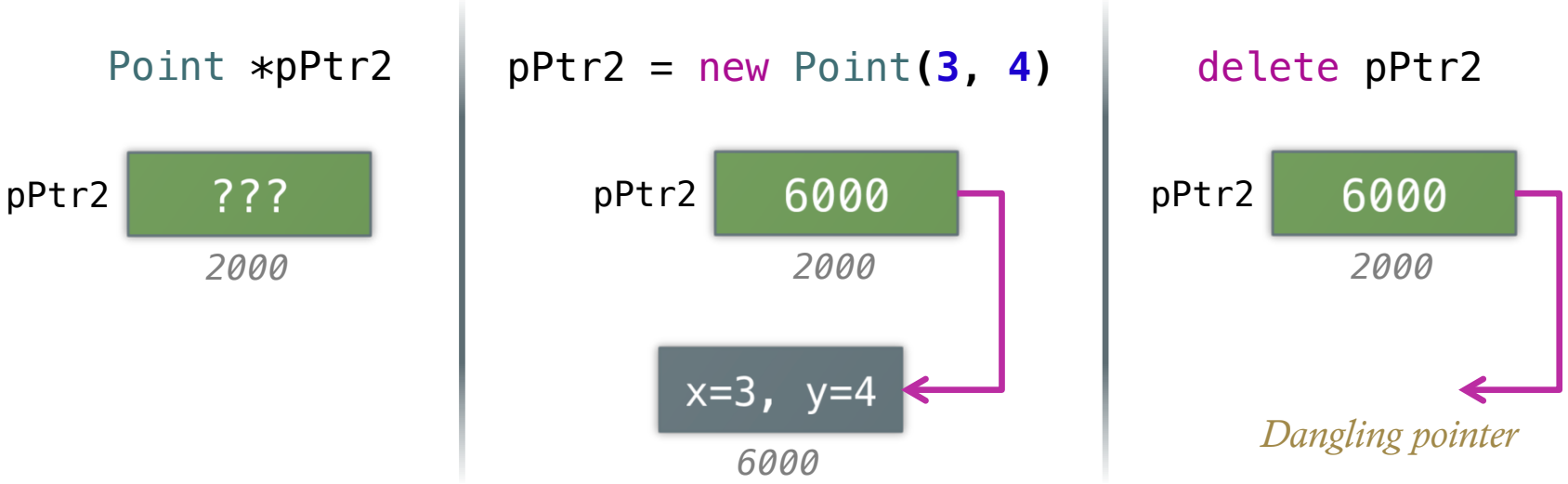
Dynamic Memory Allocation

- We can create objects "on the fly," in runtime, by using the operator **`new`**.
- Dynamically created objects are allocated in the heap, and stay there *until* we remove them by using the operator **`delete`**.



Dynamic Memory Allocation

- How do we dynamically create an object when its constructor requires arguments?



- Before `delete`-ing `pPtr1` and `pPtr2`, we can use them as follows:

```
cout << pPtr1->distance( *pPtr2 ) << endl;
```

Exercise 3

- Write a class `Path` which may have up to 100 `Point` objects:
  - It should have an **array** of up to 100 dynamically created points, which starts off empty.
  - A `pointCount` member keeps track of how many points are in the path.
  - It provides the user with an `add( double x, double y )` function that allows to 'insert' a new point in the path.
  - The user can retrieve the i^{th} point by using a `getPoint(int I)` member function.
  - It is possible to obtain the total length of the path by calling a `getTotalDistance()` method.

Destructors

- A destructor is a special function that is unique in a class, and it's called right before the object is destroyed.
- Use it to free any memory your object created dynamically.

Notice "~Class_Name"

```
/**
 * @brief Destructor.
 */
~Path()
{
   // Free all points memory.
   for( int I = 0; I < pointCount; I++ )
      delete points[I];

   cout << "Destructor has been called!";
}
```

Questions?

- You may find this material and answers to the proposed exercises at:

  http://cs.ucla.edu/~langel/cs31/session9