A Short Course on C++

Dr. Johnson

School of Mathematics

Semester 1 2008
1 POINTERS
   • Why Pointers
   • An Array is just a Pointer
   • Pointers and Functions

2 STANDARD LIBRARIES
   • Standard Containers

3 CLASSES
   • Constructing a Class
   • Using Classes
   • Inheritance
OUTLINE

1 POINTERS
   - Why Pointers
   - An Array is just a Pointer
   - Pointers and Functions

2 STANDARD LIBRARIES
   - Standard Containers

3 CLASSES
   - Constructing a Class
   - Using Classes
   - Inheritance
1. **POINTERS**
   - Why Pointers
   - An Array is just a Pointer
   - Pointers and Functions

2. **STANDARD LIBRARIES**
   - Standard Containers

3. **CLASSES**
   - Constructing a Class
   - Using Classes
   - Inheritance
1 POINTERS
   - Why Pointers
     - An Array is just a Pointer
     - Pointers and Functions

2 STANDARD LIBRARIES
   - Standard Containers

3 CLASSES
   - Constructing a Class
   - Using Classes
   - Inheritance
**Why do we need pointers?**

- Pointers are an important mechanism in any computer program.
- Fortran pointers are *hidden* from the user.
- Pointers store the **location** of a value, rather than the value.

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfff8</td>
<td>0xfff8</td>
</tr>
<tr>
<td>6</td>
<td>0xfff8</td>
</tr>
</tbody>
</table>

Memory

```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ptr_a</td>
</tr>
<tr>
<td>0xfff8</td>
<td>0xfff4</td>
</tr>
</tbody>
</table>

Compiler

```
 Declare pointers by putting * in front of the variable name. 

This means that the data it contains is a *memory address*.

There are two operators & and * associated with pointers. 

& in front of a variable means return the *memory address* of the variable.

* in front of a variable means return the value of data at that *memory address*.

We can use pointers like the variable itself by putting a * in front of it.
Using Pointers

• Declare pointers by putting * in front of the variable name.
• This means that the data it contains is a memory address.
• There are two operators & and * associated with pointers.
• & in front of a variable means return the memory address of the variable.
• * in front of a variable means return the value of data at that memory address.
• We can use pointers like the variable itself by putting a * in front of it.
 Declare pointers by putting * in front of the variable name.
This means that the data it contains is a memory address.
There are two operators & and * associated with pointers.
& in front of a variable means return the memory address of the variable.
* in front of a variable means return the value of data at that memory address.
We can use pointers like the variable itself by putting a * in front of it.
```cpp
main()
{
double a,*ptr_a;
a = 6;
ptr_a = &a;
cout << " a = " << a << " address of a = " << &a << endl;
cout << " ptr_a = " << ptr_a << " address of ptr_a = " << &ptr_a << endl;
}
```
**Examples**

```c++
main()
{
  double a,*ptr_a;
  a = 6;
  ptr_a = &a;
  cout << " a = " << a << endl;
  *ptr_a = 2;  // use pointer like it was the variable
  cout << " a = " << a << " and ptr_a = " << *ptr_a << endl;
}
```
1. **Pointers**
   - Why Pointers
   - An Array is just a Pointer
   - Pointers and Functions

2. **Standard Libraries**
   - Standard Containers

3. **Classes**
   - Constructing a Class
   - Using Classes
   - Inheritance
- An array name is just a pointer to a block of data.
- **We can use pointers instead of arrays for dynamic memory allocation.**

![Diagram showing memory and array storage]

- **Array Storage**
  - `0xffff0`: 6
  - `0xffff4`: 3
  - `0xffff8`: 1

- **Memory**
  - `0xfdd0`
  - `0xffff0`

- **Compiler**
  - `0xfdd0`

- **Note:** The hexadecimal values represent memory addresses.
OUTLINE

1 POINTERS
   • Why Pointers
   • An Array is just a Pointer
   • Pointers and Functions

2 STANDARD LIBRARIES
   • Standard Containers

3 CLASSES
   • Constructing a Class
   • Using Classes
   • Inheritance
- When we pass a variable into a function, the stored **value** is copied into the function, not the variable itself.
- To change the value of the variable itself, we must pass the memory location of the variable into the function.

```cpp
void swap(double *a,double *b)
{
    // stuff in here
}
main()
{
    double a=1,b=2;
    swap(&a,&b); // pass memory location with &
    cout << " a " << a << " b " << b;
}
```
OUTLINE

1. POINTERS
   - Why Pointers
   - An Array is just a Pointer
   - Pointers and Functions

2. STANDARD LIBRARIES
   - Standard Containers

3. CLASSES
   - Constructing a Class
   - Using Classes
   - Inheritance
A container is an object that holds other objects.

For example:
- vector – a variable-sized 1D array;
- list – a doubly-linked list;
- set – a set;
- map – an associative array;
- stack – a stack;
- queue – a queue;

We will concentrate on vectors.

We can have a vector of doubles, ints or any other data type.

You can even have vectors of your own data classes.
Containers

- A container is an object that holds other objects.
- For example:
  - vector – a variable-sized 1D array;
  - list – a doubly-linked list;
  - set – a set;
  - map – an associative array;
  - stack – a stack;
  - queue – a queue;
- We will concentrate on vectors.
- We can have a vector of doubles, ints or any other data type.
- You can even have vectors of your own data classes.
**Vectors**

- We can substitute vectors for arrays in our codes.
- Vector have built in functions to handle dynamic memory allocation.
  - We declare a vector as follows:
    ```cpp
    vector<double> vec_1;  // empty double vector
    vector<int> vec_2(10);  // ten element integer vector
    ```
  - and can resize them using resize and clear
    ```cpp
    vec.clear();  // empties vec
    vec.resize(10);  // vec now has 10 entries
    ```
We can substitute vectors for arrays in our codes.

Vector have built in functions to handle dynamic memory allocation.

We declare a vector as follows:

```cpp
vector<double> vec_1; // empty double vector
vector<int> vec_2(10); // ten element integer
```

and can resize them using resize and clear

```cpp
vec.clear(); // empties vec
vec.resize(10); // vec now has 10 entries
```
OUTLINE

1. POINTERS
   - Why Pointers
   - An Array is just a Pointer
   - Pointers and Functions

2. STANDARD LIBRARIES
   - Standard Containers

3. CLASSES
   - Constructing a Class
   - Using Classes
   - Inheritance
**Declaring a Class**

- The C++ **class** is used to define an object.
- An object is a collection of data, along with functions that act on that data.
- The data and functions may have restricted access.
- We declare classes in the following way:

```cpp
class class_name {
private data and functions
access specifier:
data and functions
::
access specifier:
data and functions
} object list;
```
We can declare a new `Point` in the main code with the command:

```cpp
Point a, b, c;
```

and access data or functions with a dot:

```cpp
a.x = 1; a.y = 1;
cout << " Distance to origin ";
cout << a.distance_to_origin() << endl;
```
OUTLINE

1. POINTERS
   • Why Pointers
   • An Array is just a Pointer
   • Pointers and Functions

2. STANDARD LIBRARIES
   • Standard Containers

3. CLASSES
   • Constructing a Class
   • Using Classes
   • Inheritance
In the previous example there is no restrictions on access to data, everything is declared as public.

There are three types of access-specifiers:

- private - no access outside the class;
- protected - access inside derived classes;
- public - full access outside the class;

We can think of this as read and write access to data.
In the previous example there is no restrictions on access to data, everything is declared as **public**.

There are three types of access-specifiers:

- private - no access outside the class;
- protected - access inside derived classes;
- public - full access outside the class;

We can think of this as read and write access to data
There are many situations where you will not want your data to be changed accidentally. Protecting data can help stop bugs in your code.

class Point{
    // class name
    int x,y;  // private access data

    public:
    void set_x(int x_){x=x_;};
    void set_y(int y_){y=y_;};  // write to x and y
    int get_x(void){return x;};
    int get_y(void){return y;};  // read x and y
    double distance_to_origin(void)
    {return sqrt(x*x + y*y);};
};  // finish a class with a semi-colon
When a new piece of data is created, C++ allows for defaults commands to be carried out.

This allows us to initialise the data default values...

or pass in parameters as arguments...

or run functions to act on the data.

The function run on initialisation is a **constructor**.

The function run on deletion is a **destructor**.
**CONSTRUCTING YOUR DATA**

- When a new piece of data is created, C++ allows for defaults commands to be carried out.
- This allows us to initialise the data default values...
- or pass in parameters as arguments...
- or run functions to act on the data.
- The function run on initialisation is a **constructor**.
- The function run on deletion is a **deconstructor**.
class Point{  // class name
    int x,y;  // private access data

public:
    void set_x(int x_){x=x_;};
    void set_y(int y_){y=y_;};
    int get_x(void){return x;};
    int get_y(void){return y;};
    Point(){x=0;y=0;};    // default constructor
    Point(int x_,int y_){set_x(x_);set_y(y_);};
    ~Point(){};
    double distance_to_origin(void)
    {return sqrt(x*x + y*y);};
};  // finish a class with a semi-colon
Deconstructing your data

- Deconstructors are used to prevent memory leaks.
- This shouldn’t be a problem unless you use pointers for dynamic storage.
- Using vectors and the std libraries should mean that you don’t need to use deconstructors.
**Pointing to an Object**

- We can create a pointer to an object in the same way as standard data types.

```cpp
Point a(2,5),*ptr_a;  // we have a Point and a pointer to Point
```

- There is special syntax `->` to access data via the pointer.

```cpp
ptr_a=&a;  // ptr_a is the location of a
ptr_a->get_x();  // access data from a
```

- You may want to pass an object to a function with a pointer.
We use overloading to enable a function with the same *name* to act differently in different situations.

One example is a simple square function:

1. Take in an `int` and return an `int`

   ```c
   int square(int arg) return (arg * arg);
   ```

2. Take in a `double` and return a `double`

   ```c
   double square(double arg) return (arg * arg);
   ```
1. **POINTERS**
   - Why Pointers
   - An Array is just a Pointer
   - Pointers and Functions

2. **STANDARD LIBRARIES**
   - Standard Containers

3. **CLASSES**
   - Constructing a Class
   - Using Classes
   - Inheritance
C++ allows the creation of new classes from existing classes.

We say the new class is derived from a base class.

The general syntax for creating derived classes is

```cpp
class class_name : access-specifier base_class_name;
```

Here Circle inherits all data and functions from Point.

```cpp
class Circle: public Point{
    double radius; // inherits x and y

public:
    void set_r(int r){radius=r;};
    int get_r(void){return radius;};
    Circle(int x_, int y_, int r):Point(x_, y_)set_r(r);;
}; // finish a class with a semi-colon
```
C++ allows the creation of new classes from existing classes.

We say the new class is derived from a base class.

The general syntax for creating derived classes is

```cpp
class class_name : access-specifier
base_class_name ;
```

Here Circle inherits all data and functions from Point.

```cpp
class Circle : public Point{
  double radius;  // inherits x and y

public:
  void set_r(int r){radius=r;};
  int get_r(void){return radius;};
  Circle(int x_,int y_,int r):Point(x_,y_)set_r(r);;
};  // finish a class with a semi-colon
```
Does the function `distance_to_origin` still apply?

- We can declare a new function in `Circle`, with the same name, to override the function in `Point`.
- We add the function in `Circle`:

```c++
double distance_to_origin(void)
{
    return max(0.,sqrt(x*x + y*y)-radius);
}
```

- We need to change the access to `x` and `y` in `Point` to `protected`, to allow read/write access for `Circle`. 
Does the function `distance_to_origin` still apply?

We can declare a new function in `Circle`, with the same name, to override the function in `Point`.

We add the function in `Circle`:

```cpp
double distance_to_origin(void)
{
    return max(0.,sqrt(x*x + y*y)-radius);
}
```

We need to change the access to `x` and `y` in `Point` to `protected`, to allow read/write access for `Circle`. 
Does the function `distance_to_origin` still apply?

We can declare a new function in `Circle`, with the same name, to override the function in `Point`.

We add the function in `Circle`:

```cpp
double distance_to_origin(void)
{
    return max(0.,sqrt(x*x + y*y)-radius);
}
```

We need to change the access to `x` and `y` in `Point` to protected, to allow read/write access for `Circle`. 
Overloading Functions

- Does the function `distance_to_origin` still apply?
- We can declare a new function in `Circle`, with the same name, to override the function in `Point`.
- We add the function in `Circle`:

```cpp
double distance_to_origin(void)
{
    return max(0., sqrt(x*x + y*y) - radius);
};
```

- We need to change the access to `x` and `y` in `Point` to protected, to allow read/write access for `Circle`. 
Now because Circle is a Point, a pointer to Point will also be a pointer to Circle

Then we may write

```cpp
Point *ptr_a; // a pointer to Point
Circle b(2,5,2); // a circle
ptr_a = &b; // set the pointer to the circle b
cout << "distance to origin is ";
ptr_a->distance_to_origin();
```

The output is
distance to origin is 5.38516
Now because Circle is a Point, a pointer to Point will also be a pointer to Circle.

Then we may write

```cpp
Point *ptr_a; // a pointer to Point
Circle b(2,5,2); // a circle
ptr_a = &b; // set the pointer to the circle b
cout << " distance to origin is ";
ptr_a->distance_to_origin();
```

The output is

distance to origin is 5.38516
Now because Circle is a Point, a pointer to Point will also be a pointer to Circle

Then we may write

```cpp
Point *ptr_a;   // a pointer to Point
Circle b(2,5,2); // a circle
ptr_a = &b;    // set the pointer to the circle b
```

```cpp
cout << " distance to origin is ";
ptr_a->distance_to_origin();
```

The output is

distance to origin is 5.38516
Because we use a pointer, the original distance function is called.

Get round this with a **virtual** function.

Simply change the appropriate line in the class `Point` to

```cpp
virtual double distance_to_origin(void)
{
    return sqrt(x*x + y*y);
}
```

and the output will now be

distance to origin is 3.38516
Because we use a pointer, the original distance function is called.

Get round this with a virtual function.

Simply change the appropriate line in the class Point to

```cpp
virtual double distance_to_origin(void) {
    return sqrt(x*x + y*y);
};
```

and the output will now be
distance to origin is 3.38516
A lot to take in in a short space of time!
Read and review Andrews notes.
Use online sources for further examples on object-orientated programming.
The best way to learn is to practice!
SUMMARY

- A lot to take in in a short space of time!
- Read and review Andrews notes.
- Use online sources for further examples on object-orientated programming.
- The best way to learn is to practice!