G52CPP
C++ Programming
Lecture 24

Dr Jason Atkin

This lecture and beyond…

- This lecture
  - Lambda functions
  - When is a duck an instrument? (Multiple Inheritance)

- Next Tuesday
  - Revision and exam structure lecture
  - What may be on the exam
  - What will it look like
  - What types of questions
  - Going through examples
- Office hours for remaining 2 lectures this term
- Optional lectures and exam questions after Easter

A functor (function object)

```
class MyFunctor3
{
public:
   MyFunctor3( int& var )
      : var( var )
   {
   }

   int operator() ( int x )
   {
      return x + var;
   }

protected:
   int& var;
};
```

Constructor takes variable by reference and initialises data

Operator() uses data member

Note: the data member is a reference

- We could store information by reference (not a copy)

```
int iAdd = 5;
MyFunctor3 addVariable(iAdd);

cout << addVariable( 1 )
            << endl;

iAdd = 19;
cout << addVariable( 1 )
            << endl;

iAdd = -12;
cout << addVariable( 1 )
            << endl;
```

Lambda functions

- Lambda functions…
- Nameless functors that get created on the fly
- Specify what to capture when the lambda is created, like passing local/global variables into the constructors
  - Pass variables in by value or by reference
    - Determines what happens if the surrounding variable is altered or not, and whether changes are seen
  - Specify to capture specifics or all used
- Return type: usually implicit (from return)
  - Can specify the return type if necessary, using "-> type"
- Basic lambda function:

  [Capture] (parameters) -> return_type { function body }

```
[iAdd]( int i )->int { return i + iAdd; }
```

Using a basic lambda function

```
int aiMyArray[] = { 0,1,2,3,4,5,6 }; // Array
int iAdd = 5; // Local variable
iAdd = 7; // Change value of variable used
```

```
ApplyToAllElements( aiMyArray, 7,
   [iAdd]( int i )->int { return i + iAdd; }
   );
```

```
for ( const int& myelem : aiMyArray )
     cout << myelem << " ";
cout << endl;
```

Types of Captures []

- These work like passing the surrounding variables into the constructor of a functor
  - By reference or value
  - And storing them for use in the functor
  - Beware of lifespan of variables passed by reference!!! E.g. locals
  - **Lambda functions do not extend the lifespan of variables**
- Parameters are added within the []
  - [v] Capture v by value
  - [&v] Capture v by reference
  - [=] Capture everything which is used by value
  - [&] Capture everything which is used by reference
  - [] Capture nothing
  - [&,v] Example of multiple captures, capture all by reference except v which is captured by value
  - Multiple: separate with `,` and must not overlap (e.g. [&,&v]) them

How does the lambda function work?

- Code:

```
ApplyToAllElements( aiMyArray, 7,
      [iAdd]( int i )->int { return i + iAdd; }
      );
```

- Implementation:

```
template < typename T >
void ApplyToAllElements( int* aiArray, int iNum,
      T addfunc )
{
    for ( int i = 0; i < iNum; i++ )
    {
      aiArray[i] = addfunc( aiArray[i] );
    }
}
```

How does the lambda function work?

- Code:

```
ApplyToAllElements( aiMyArray, 7,
      [iAdd]( int i )->int { return i + iAdd; }
      );
```

- Effective Lambda code:

```
class annon
{
    annon( int iAdd ) : iAdd(iAdd) {}
    int operator() (int i) { return i + iAdd; }
    int iAdd;
}
```

```
  [iAdd]( int i )->int { return i + iAdd; }
```

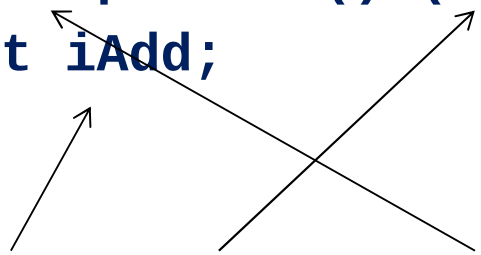
How does the lambda function work?

- Code:

```
ApplyToAllElements( aiMyArray, 7,
      [iAdd]( int i )->int { return i + iAdd; }
      );
```

- Effective Lambda code:

```
class annon
{
    annon( int iAdd ) : iAdd(iAdd) {}
    int operator() (int i) { return i + iAdd; }
    int iAdd;
}
```



```
[iAdd]( int i )->int { return i + iAdd; }
```

What do these do?

- Example lambda functions:

```
[iAdd]( int i ){ return i + iAdd; }

[=]( int i ) { return i + iAdd; }

[&iAdd]( int i ) {--iAdd; return i + iAdd; }

[&]( int i ) { ++iAdd; return i + iAdd;}
```

Why might lambdas be important?

- Many STL algorithms accept lambda functions, e.g.:

```
vector<int> vec = { 1,2,3,4,5,6,7 };

cout << "any_of < 2 : " <<
      any_of( vec.begin(),vec.end(),
            []( int i ) { return i < 2; } ) << endl;
cout << "count_if > 4 : " <<
      count_if( vec.begin(),vec.end(),
            []( int i ) { return i > 5; } ) << endl;
cout << "count_if even : " <<
      count_if( vec.begin(),vec.end(),
            []( int i ) { return !(i%2); } ) << endl;
cout << "find_if(divisible by 5) : " <<
      *find_if( vec.begin(),vec.end(),
            []( int i ) { return !(i % 5); } ) << endl;
```

When is a duck an instrument?

Multiple Inheritance

Multiple inheritance

- In Java you can `implement` multiple interfaces, but only `extend` one class
- **In C++ you can inherit from (extend) multiple classes**
- At times it makes sense to inherit from multiple base classes
  - Maybe something can be both a duck and an instrument?
  - You inherit all of the behaviour (i.e. function **implementations**), not just the interface
- **But be careful of multiple inheritance**
  - **There are dangers, and confusing elements**
  - **There may be easier ways (e.g. composition)**

What re-use options are there?

- There are other ways to support re-use:

1. Composition/aggregation
   - Models the '**has a**' or '**is a part of**' relationship
   - ***Composition*** is a stronger form
     - The 'part' only exists while the containing class exists
2. Inheritance
   - '**Is a**' or '**is a type of**'
   - **Implementation**: Make the 'type of' a sub-class
3. Uses / association
   - **Implementation**: Maintain a pointer or reference between them, to get to the other object
     - Create the other object separately, then set pointer to it
     - Other object is separate – needs to be destroyed separately

Musical Duck

Base classes

```
class Duck
{
public:
   // Constructor
   Duck( int weight = 1 )
      : weight(weight)
   {}

   // Get the weight
   int GetWeight() const
   { return weight; }

//protected:
   int weight;
};
```

Two classes.
Both have a weight,
one has a volume.

```
class Instrument
{
public:
   Instrument(  int weight = 1,
                int volume = 1 )
      : weight(weight)
      , volume(volume)
   {}

   int GetVolume() const
   { return volume; }

   int GetWeight() const
   { return weight; }

//protected:
   int volume;
   int weight;
};
```

Musical Duck 1 : Composition

```
class MusicalDuck1
{
public:
   // Constructor
   MusicalDuck1(
     int weight = 1,
     int volume = 2 )
   : d(weight)
   , i(weight,volume)
   {}

   // Contains a 'Duck'
   Duck d;

   // Contains 'Instrument'
   Instrument i;

   // Get instrument volume
   int GetVolume() const
   { return i.GetVolume(); }

   // Get weights
   int GetInstWeight() const
   { return i.GetWeight(); }

   int GetDuckWeight() const
   { return d.GetWeight(); }
};
```

Data from contained objects is available to the container object. Have to expose any methods manually.

Exposing the methods…

```
// Get instrument volume
   int GetVolume() const
   {
     return i.GetVolume();
   }

// Get weights
   int GetInstWeight() const
   {
     return i.GetWeight();
   }

   int GetDuckWeight() const
   {
     return d.GetWeight();
   }
};
```

- You need to expose the methods manually because they are on component objects
- Simple wrapper function code though

Musical Duck 2 : Inheritance

```
class MusicalDuck2
: public Duck
, public Instrument
{
public:
   // Constructor
   MusicalDuck2(
        int weight = 1,
        int volume = 2 )
   : Duck(weight)
   , Instrument(weight,volume)
   { }
};
```

```
// GetVolume() is inherited
// and available

// GetWeight() is inherited
// (twice) and available
```

GetVolume() is available automatically

GetWeight() is available from both base classes (i.e. twice)

Q: How do you think that we differentiate between them?

Musical Duck 1 : Composition

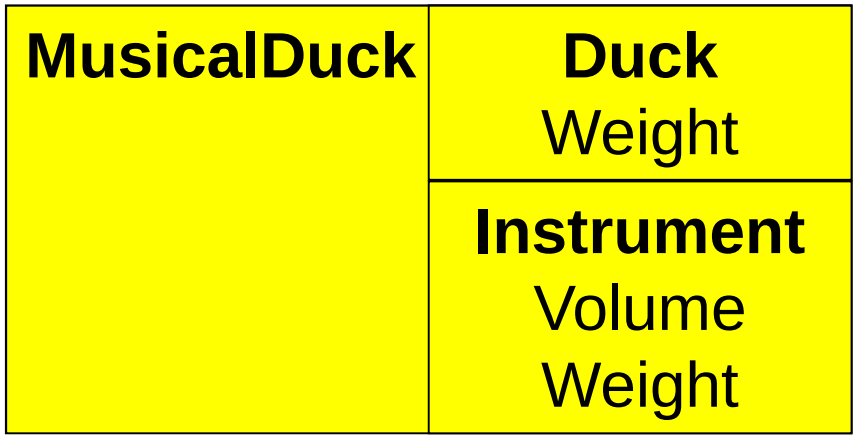
```
class MusicalDuck1
{
public:
   // Constructor
   MusicalDuck1(
      int weight = 1,
      int volume = 2 )
   : d(weight)
   , i(weight,volume)
   { }

// Contains a 'Duck'
Duck d;
// Contains 'Instrument'
Instrument i;
…
};
```

```
MusicalDuck1 mduck1;
printf( "Musical duck at %p\n",
        &mduck1 );
printf( "Duck at %p\n",
        &mduck1.d );
printf( "Duck.weight at %p\n",
        &mduck1.d.weight );
printf( "Instrument at %p\n",
        &mduck1.i );
printf("Instr.Volume at%p\n",
        &mduck1.i.volume );
printf( "Instr.Weight at %p\n",
        &mduck1.i.weight );
```

```
Musical duck at 0x22ccd0
Duck at 0x22ccd0
Duck.weight at 0x22ccd0
Instrument at 0x22ccd4
Instr.Volume at 0x22ccd4
Instr.Weight at 0x22ccd8
```

Musical Duck 1 : Composition

```
class MusicalDuck1
{
public:
   // Constructor
   MusicalDuck1(
      int weight = 1,
      int volume = 2 )
   : d(weight)
   , i(weight,volume)
   { }

// Contains a 'Duck'
Duck d;
// Contains 'Instrument'
Instrument i;
…
};
```



Musical duck at 0x22ccd0
Duck at 0x22ccd0
Duck.weight at 0x22ccd0
Instrument at 0x22ccd4
Instr.Volume at 0x22ccd4
Instr.Weight at 0x22ccd8

Musical Duck 2 : Inheritance

```
class MusicalDuck2
: public Duck
, public Instrument
{
public:
   // Constructor
   MusicalDuck2(
       int weight = 1,
       int volume = 2 )
   : Duck(weight)
   , Instrument(weight,volume)
   { }

…
};
```

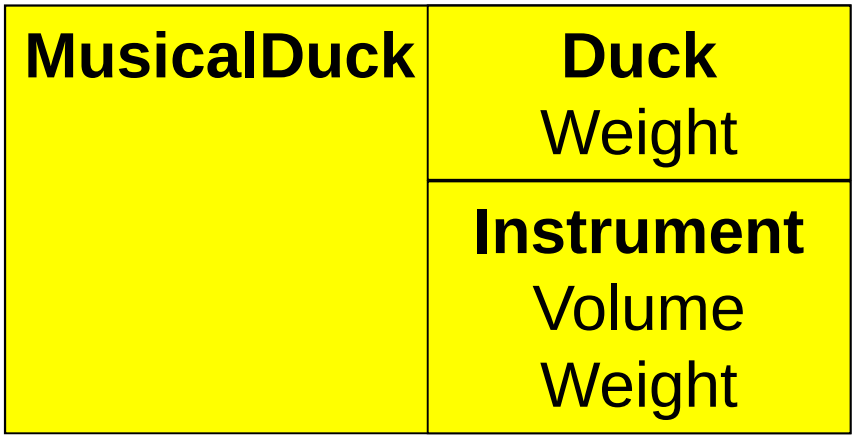
```
MusicalDuck2 mduck2;
printf( "Musical duck at %p\n",
   &mduck2 );
printf( "Duck at %p\n",
   (Duck*)(&mduck2) );
printf( "Duck.weight at %p\n",
   &mduck2.Duck::weight );
printf( "Instrument at %p\n",
   (Instrument*)(&mduck2) );
printf( "Instr.Volume at %p\n",
   &mduck2.volume );
printf( "Instr.Weight at %p\n",
   &mduck2.Instrument::weight );
```

Musical duck at 0x22ccd0
Duck at 0x22ccd0
Duck.weight at 0x22ccd0
Instrument at 0x22ccd4
Instr.Volume at 0x22ccd4
Instr.Weight at 0x22ccd8

Important notes:

Important notes:

- The base-class information is contained within the sub-class structure
- Casting a pointer can change the address:
  `(Instrument*)(&mduck2)`
- Composition may be easier in many cases
- Main difference is that you have to wrap/expose the functions yourself



If data or methods are available from multiple base classes you need to **disambiguate**

Use scoping to do this:

```
&mduck2.Duck::weight
&mduck2.Instrument::weight
```

Casting Pointers and References

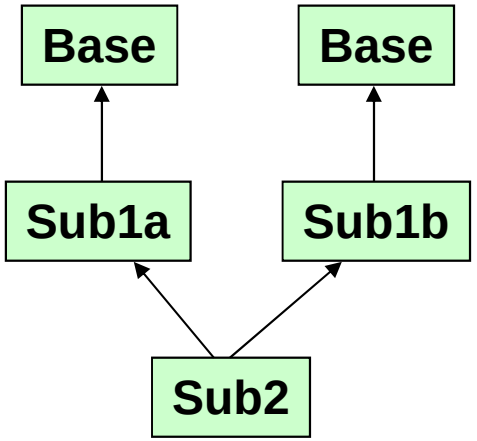
- **I used C-style casting to keep the code short**
  - **DO NOT DO THIS!!!**
- Use `static_cast` or `dynamic_cast` for base class to sub-class (no cast needed for sub to base class : IS-A)
  - Dynamic cast will check (at runtime) that the pointer really is to an object of that type
- **IMPORTANT:** If you cast pointers or references when multiple inheritance is being used, then addresses may change
  - Normally, casting a pointer just changes the type, but leaves the address unchanged
  - If you go to or from a second (or later) base class, the address (pointer value) will change!
  - If you go back again (to sub-class), the pointer value changes back again (use dynamic cast if necessary, to check the type)

Shared base classes

Common base classes

```
#include <cstdio>

struct Base
{
   int i;
};

struct Sub1a : public Base
{
   Sub1a()
   {
        i=1;
   }
};
```

```
struct Sub1b : public Base
{
   Sub1b()
   {
        i=2;
   }
};

struct Sub2
   :    public Sub1a,
        public Sub1b
{
};
```

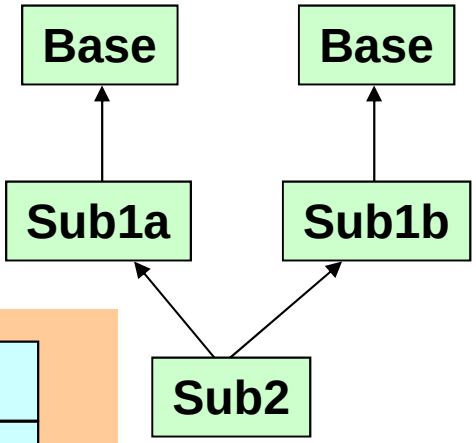


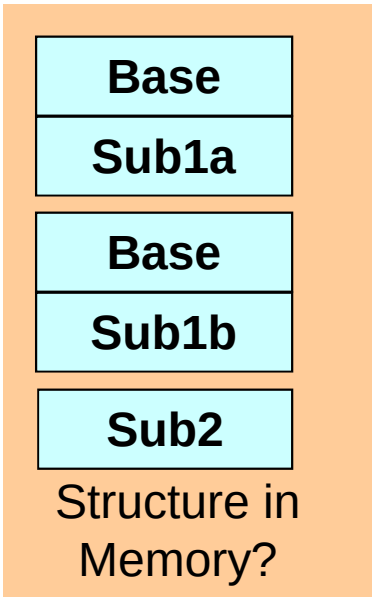
Common base classes

```
#include <cstdio>

struct Base { int i; };
struct Sub1a : public Base { Sub1a() {i=1;} };
struct Sub1b : public Base { Sub1b() {i=2;} };
struct Sub2 : public Sub1a, public Sub1b { };

int main()
{
   printf( "Sizes: %d %d %d %d\n",
       sizeof(Base), sizeof(Sub1a),
       sizeof(Sub1b), sizeof(Sub2) );

   Sub2 ob;
// printf( "%d\n", ob.i ); WRONG!!!
   printf( "%d\n", ob.Sub1a::i );
   printf( "%d\n", ob.Sub1b::i );
};
```

Sub1 and **Sub2** each have a copy of **i**, which they inherit. **Sub2** has 2 copies

```
Output:
4 4 4 8
1
2
```



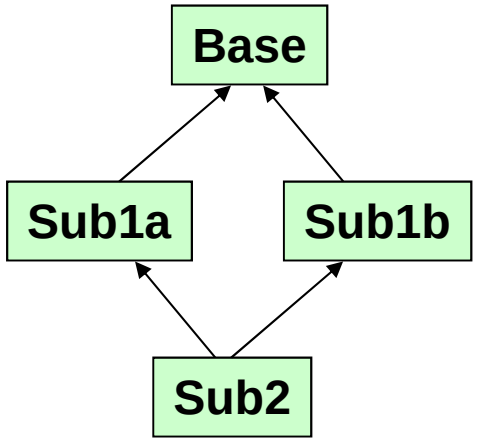


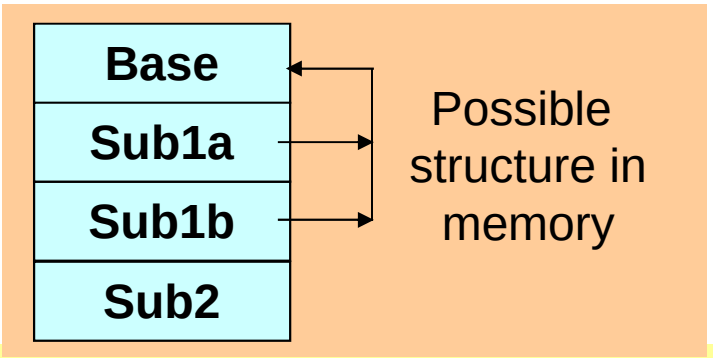
Structure in Memory?

Virtual base classes

```
#include <cstdio>

struct Base { int i; };
struct Sub1a : virtual public Base { Sub1a() {i=1;} };
struct Sub1b : virtual public Base { Sub1b() {i=2;} };
struct Sub2 : public Sub1a, public Sub1b {};

int main()
{
   printf( "Sizes: %d %d %d %d\n",
       sizeof(Base), sizeof(Sub1a),
       sizeof(Sub1b), sizeof(Sub2) );

   Sub2 ob;
   printf( "%d\n", ob.i );
   printf( "%d\n", ob.Sub1a::i );
   printf( "%d\n", ob.Sub1b::i );
};
```

```
Output:
4 8 8 12
2
2
2
```





Can now use `ob.i` (only one copy)

Note: Size increased by 4 bytes, for the pointer to virtual base class

Safe multiple inheritance and alternatives

Multiple inheritance dangers

- Be careful if you use multiple inheritance
- Beware of:
  - **Inheriting the same names from multiple base classes**
  - **Inheriting the same base class twice, through two different intermediate classes**
- To resolve the problem:
  - Use scoping operator `::` to dis-ambiguate
  - Or use virtual base classes, to keep one copy
  - Or ensure that only one base class has any data, or any non-abstract methods …

Abstract/pure-virtual base class

No implementation is given for any of these functions They must be implemented in **concrete** sub-classes

```
class PureVirtual
{
  virtual void func1() = 0;
  virtual int func2() = 0;
  virtual double func3(int,double) = 0;
};
```

- No member data is specified
- All functions are pure virtual (i.e. abstract, **`= 0`**)
  - MUST be implemented in any concrete sub-class
- **This class acts like a Java interface and can be used in the same way**

Should I Use Inheritance?

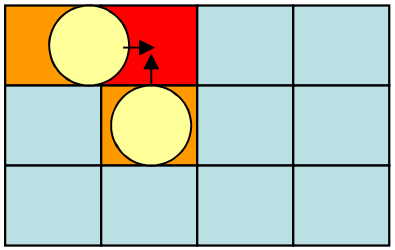
- Inheritance says this object IS an object of the other type, not just that they have SOME commonality
- Do not assume that inheritance is always the answer
  - Be sure that you really want ‘is-a’ and not ‘has-a’
  - Aggregation or composition are often better options if you just want to reuse some code
  - Although you then have to re-implement function wrappers
- Do not assume that multiple inheritance is needed
  - It is **never** necessary (but is sometimes useful)
- Do you need to treat different sub-class types as the base class? (i.e. need to model ‘is-a’?)
- To be safe, adopt the Java way of having only one base class any data or function implementations
  - i.e. all but one base class is an ‘interface’

Some coursework comments (that we didn't cover last lecture)

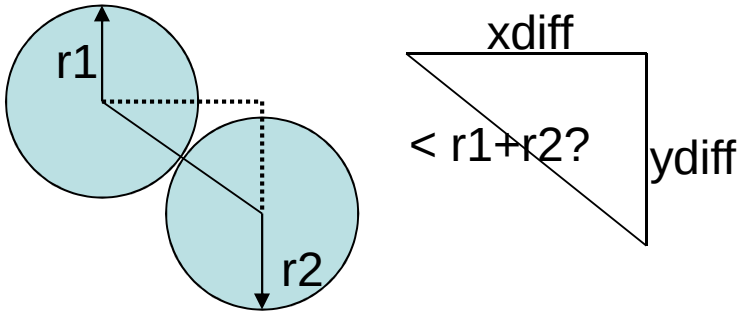
Collision detection etc

Collisions

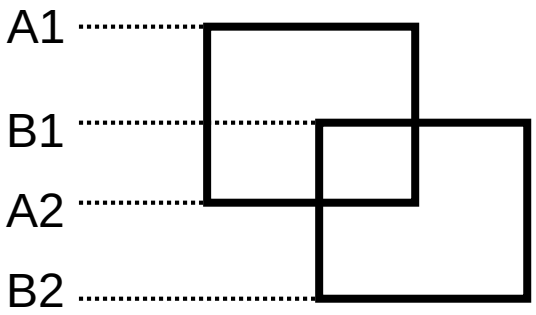
- Same tile
  - Easy but unreliable



- Circles intersect
  - Centres less than total radius apart
  - Know radii
  - Know x dist between centres (xdiff)
  - Know y dist between centres (xdiff)
  - Pythagoras' Theorem: Collision if $Xdiff^2 + ydiff^2 < (r1+r2)^2$



- Rectangles intersect
  - Check the corners
  - Check x and separately
  - If B1 or B2 between A1 & A2
    or A1 or A2 between B1 & B2



BouncingBall1 : MovementObject

```
class BouncingBall1 : public BouncingBall
{
public:
   BouncingBall1(BouncingBallMain* pEngine, int iID, int
       iDrawType, int iSize, int iColour, char* szLabel,
       int iXLabelOffset, int iYLabelOffset, TileManager*
       pTileManager );

   void SetMovement(
       int iStartTime, int iEndTime, int iCurrentTime,
       int iStartX, int iStartY, int iEndX, int iEndY );

   void DoUpdate( int iCurrentTime );

protected:
   /* Movement position calculator */
   MovementPosition m_oMovement;

   // Pointer to the tile manager
   TileManager* m_pTileManager;
};
```

Using the Movement object

- Allows a caller to specify where the object will move from and to and when.
- **Setup()** sets up a new movement:
  - Start position (x and y), End position (x and y), Start time, End time
- **Calculate()** sets up an internal x and y member according to the time
- **GetX()** and **GetY()** retrieve the calculated time
- **HasMovementFinished(iCurrentTime)** returns true if move completed
- **Reverse()** reverses the x and y coordinates, and updates times to reverse the move

```
void BouncingBall1::SetMovement( int iStartTime, int
   iEndTime, int iCurrentTime,
    int iStartX, int iStartY, int iEndX, int iEndY )
{
   m_oMovement.Setup( iStartX, iStartY, iEndX, iEndY,
        iStartTime, iEndTime );
   m_oMovement.Calculate( iCurrentTime );
   m_iCurrentScreenX = m_oMovement.GetX();
   m_iCurrentScreenY = m_oMovement.GetY();
}
```

Tiles

Tile based games assume a rectangular map consisting of a grid of tiles

- Each tile has a type
- Type determines how it is drawn and whether it blocks movement
- e.g.
  ‘X’ = wall,
  ‘ ’ = passage
  ‘-’ = pellet to eat

x coordinate

y coordinate

x=0 y=0	x=1 y=0	x=2 y=0	x=3 y=0	x=4 y=0
x=0 y=1	x=1 y=1	x=2 y=1	x=3 y=1	x=4 y=1
x=0 y=2	x=1 y=2	x=2 y=2	x=3 y=2	x=4 y=2
x=0 y=3	x=1 y=3	x=2 y=3	x=3 y=3	x=4 y=3

BouncingBallMain.h

```
class BouncingBallMain :
public BaseEngine
{
protected:
  …

// A member object. Object is created when
  the BouncingBallMain is created
TileManager m;
```

BouncingBallMain.cpp

- Specify how many tiles wide and high

```
m.SetSize( 20, 20 );
```

- Specify the screen x,y of top left corner

```
m.SetBaseTilesPositionOnScreen( 250, 100 );
```

- Tell it to draw tiles
  from x1,y1 (i.e. 2,0) to x2,y2 (i.e. 17.19) in tile array,
  to the background of this screen

```
m.DrawAllTiles( this /*Engine*/,
  this->GetBackground() /*Or foreground*/,
  2, 0, 17, 19 );
```

BouncingBall – update tiles

- **Find the X value of the tile**

```
int iTileX = m_pTileManager->
    GetTileXForPositionOnScreen(m_iCurrentScreenX);
```

- **Find the Y value of the tile**

```
int iTileY = m_pTileManager->
    GetTileYForPositionOnScreen(m_iCurrentScreenY);
```

- **Get the value of that tile**

```
int iCurrentTile = m_pTileManager->
    GetValue( iTileX, iTileY );
```

- **Change the value of that tile and redraw it**

```
m_pTileManager->UpdateTile( GetEngine(), iTileX,
    iTileY, iCurrentTile+1 );
```

Next lecture

Exam structure and revision hints