# G52CPP
# C++ Programming

Some example questions

and

some tactical hints about how to do the exam

# Evaluate: SET/SEM

- Please visit Bluecastle https://bluecastle.nottingham.ac.uk/Account/Login?ReturnUrl=%2f
- And choose “My Surveys”
  - Please find both the SET and SEM evaluations for G52CPP
- Please take care to understand which way round 1 and 5 are, i.e. which is good
  - Many people have got them the wrong way around this year
  - Apparently 5 is bad not good

# Key changes this year

- Mostly in response to the feedback last year
- Less C at the beginning
- Discussed some key C++11 features
  - Covered Functors and Lambda functions
- Updated the coursework
  - Changed to SDL version 2
  - Added a requirement to use an STL container
- Moved the lecture content earlier in semester
  - To be able to start on coursework earlier
  - And the early Easter is a problem
- Covered less in lectures and took more time
  - Two optional lectures after Easter, and ‘question’ lectures

# Some exam comments

Same format as last year

**The University of Nottingham**

SCHOOL OF COMPUTER SCIENCE

A LEVEL 2 MODULE, SPRING SEMESTER 2013-2014

**G52CPP, C++ Programming**

Time allowed 1 hour and 30 minutes

*Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced*

**Answer Question 1 and ONE of Questions 2 or 3**

No calculators are permitted in this examination.

*Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject specific translation dictionaries are not permitted.*

*No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.*

***DO NOT turn your examination paper over until instructed to do so***

# Exam structure

- Do TWO questions, including Q1
  - I.e. Do Q1 + Q2  OR  Q1 + Q3
  - 30 marks per question
- Tactically choose your question from Q2 and Q3
  - Obviously, read the whole question
- Q1: *Mostly* the *more* basic C++ things
  - E.g. inheritance, pointers, references, const
  - Consider existing code
  - Write your own code to do things
- Q2, Q3: contain more advanced C++ things
  - May refer back to code from Question 1 (means you have to read less code, since Q1 is compulsory anyway)
  - Likely to contain topics which I know people find more difficult
  - E.g. STL, lambdas, casting, exceptions, templates, virtual inheritance and/or vtables, operator overloading, …

# Changes from previous years

- Expect more STL code
  - Standard Template Library
  - E.g. container classes and algorithms
  - Know at least the ones you used:
    - list<>, array<>, vector<>
    - for_each, count, count_if, sort, copy
  - push_back() and pop_back() are also useful to know (supported by vector and list)
    - + push_front() and pop_front(), supported by list and deque
  - Ensure that you understand the content of lab 3 (on STL)
- Expect something about functors and/or lambda functions somewhere (possibly using algorithms)

# Key STL Container Features

- array<type,size>
  - Wrapper for [] array, fixed size
- vector<type>
  - Dynamically resizing array, fast to find element by index
- deque<type>
  - Double ended queue, can use []
  - Add or remove from start or end, less efficiently anywhere else
- list<type>
  - Doubly-linked list
  - Fast insert and deletion, and movement to next/previous
  - No [] operator
- forward_list<type>
  - Singly-linked list
  - Move to next, insert after, etc

# Differences from pre-2013/2014…

- Last year changed from previous years
- Much less of “what is the output of this code…” questions than previous exams, due to external examiner request last year
  - Still at least one though
- Less focus on the C function library
- Compulsory question!
  - Some core things you cannot avoid
- Fewer questions (2 from 3, not 3 from 5)
  - Less of the module content is covered
  - Much of it is still somewhere on the exam though
- **Warning: remaining comments are general**

# Hints

- Read Q1 first and do what you can
- It should give you a more gentle intro
- Parts are referenced from Q2 and Q3
  - Useful to have worked through it before Q2 and Q3
- Read the introduction line for each, saying what Q2 and Q3 relate to
  - There will be other things too, but these list key things which will be in there – some cannot be mentioned
- Read ALL parts of the question
- Check the rest of the paper if you are stuck – sometimes it may jog your memory
  - e.g. does a later code sample answer something?

# Example question types…

- “This code should …., what is wrong with it?”
- “Will this code compile? If so, what will the output be? If not then why not and explain how to fix the problem”
- “Consider this code …., provide an implementation for the … function, which will …”
- “Provide a definition for a function which…”
- “Provide a definition for a class which ….”
- “Provide a definition for the member function … OUTSIDE of the class definition”
- “What does … mean in C++”
- “What is the difference between … and …”
- “What is wrong with this code…”

# Some things to consider…

- What is const? What should be const?
- Where are references used?
  Where should they be used?
- What pointers are being used?
- What is virtual? What should be virtual?
- Should functions be defined inside the class definition?
- What is public, private, protected etc?
  - What access should things be?

# Some example question types

For you to answer…

# What are the values?

```
class Base
{
protected:
     int i;
public:
     virtual int a() {return i+1;}
     int b() { return i+2; }
     void set( int n ) { i = n; }
};

class Sub : public Base
{
public:
     int a() { return i+3; }
     int b() { return i+4; }
     void set( int n ) { i = n+5; }
};
```

```
int main(int argc, char* argv[])
{
     Sub s;
     Base& r = s;

     s.set( 10 );
What is returned by:
 s.a(), s.b(), r.a(), r.b()?

     r.set( 20 );
What is returned by:
 s.a(), s.b(), r.a(), r.b()?

     return 0;
}
```

Modified 2010/2011 Q1E

# What are the values?

```
class Base
{
protected:
     int i;
public:
     virtua
     int b(
     void s
};

class Sub :
{
public:
     int a(
     int b() { return i+4; }
     void set( int n ) { i = n+5; }
};
```

```
int main(int argc, char* argv[])
{
                                        b()?
                                        b()?
     return 0;
}
```

**Show your working out!!!**

(in case you get it wrong, so we can give partial marks)

# Extensions and variants

- Provide the code for a C++ style cast from a Base* pointer to a Sub* pointer
- What other casts may have been appropriate for this? Explain the differences between them
- Add a member int j to the subclass as well and a question could be asked about the slicing problem
- Add constructors and the use of initialisation list, base class initialisation and/or default parameters could be tested

# Basic feature question possibilities

- Many examples could be easily generated:
  - References or pointers vs pass by value
  - static v non-static local variables
  - virtual v non-virtual functions
  - static v non-static member variables
  - Exception throwing and catching
  - Pointer arithmetic (espec. ++) or pointers to pointers
- Identify what is used in the code sample
  - Look for subtleties
  - These often test knowledge indirectly
  - Test your ability to apply your knowledge rather than to memorise
- What is needed/wrong in a sample? E.g.:
  - Was something static and needed to be non-static?
  - Did it need to be pass or return by reference and wasn't?
  - Did it need to be virtual and wasn't?

# Exam Content

- No details of common C-library functions are really needed
  - Be able to read/understand the code though if used
  - E.g. strcmp, strcpy, printf, …
- Ensure that you can create a **template function** and **operator overload**, but creating template class is **not** needed
  - Be able to create a macro (#define) and understand the difference
- Understand about conversion constructors and operators, copy constructors and assignment operators

# Other things to know (not exhaustive list!)

- `struct` vs `class` vs `union`
- `struct` and `class` features:
  - member functions
  - `virtual` functions
  - inline functions (and definitions within a class definition)
  - Scoping and ::
  - constructors (especially default and copy)
  - assignment operators,
  - Conversion constructors and operators
- `const` members, parameters, references, pointers
- `static` local variables, member data and functions
- Function pointers, v-tables
- Exceptions and exception handling
- Understand the C++-style casts
  - What do they do? What are the differences between them?

# How to revise

- Try the examples on the lecture slides
- Go through the slides in this presentation
  - There are loads more than I will go through in lectures
- Try the past papers – ensure that you can do them without having to look up the answers
  - If you get stuck, look up how to do it, don't look at the answer until you are sure
  - Copy code samples into a compiler and experiment with them
- Go through example code type questions
  - What is the question asking?
  - What do I need to know to answer it?
  - Are there any tricky bits?
  - What is the answer?

# Quick ‘knowledge’ questions

# Quick questions

- What is the difference between a class and a struct in C++?
- What is the :: operator used for in C++? Give two examples of its use.
- What is the difference between protected and private member data in C++?
- What is meant by the term ‘overloaded function’ in C++?
- Name four methods which may be created implicitly by a C++ compiler for a C++ class if they are needed.
- *Or “In addition to a default (no-parameter) constructor and a destructor, name two other methods which may be created implicitly by a C++ compiler for a C++ class if they are needed.”*

# More (complex?) ‘quick’ questions

- If you add a member function to a class, will the objects grow? Why/why not?
- What is the difference between a static and a non-static member function?
- What is the difference between a global function and a static member function?
- What does ‘friend’ do in C++?
- What is meant by “an overridden function” in C++? Give an example.
- What is a virtual destructor? When would it be useful?
- **Why should you not call a virtual function from a destructor?**
- **Why is Java’s ‘finally’ not needed in C++>?**

Another type of question to expect:

“Write code to …”

# Write code to…

1. Give an implementation for a default constructor for this class…

2. Implement a function for class B so that the following code would compile:

```
B a, b, c;
a = b * c;
```

- Work out what the function needs to do first
- i.e. operator overload for * operator in this case
- Then create the code to do it

# Quick code questions (08/09)

3d: Write a function called min() which takes two long parameters called val1 and val2 and returns the value of the lesser of the two values as a long, using the ternary operator ( ?: ). (i.e. it should return the minimum of the two values.)

3e: Provide the code for a macro called MIN using #define which will perform the same function as your min() function in question 3d.

# 2010/2011 Q1b

- Provide C/C++ code to define a macro (using #define) called PRODUCT, which takes two parameters and returns the product of the two parameters.
- e.g. PRODUCT(2,3) is 6, PRODUCT(1,5) is 5 and PRODUCT(-3,4) is -12

Answer:

```
#define PRODUCT(a,b) ((a)*(b))
```

# Best wishes for doing well in the exam

## Give me a 100% mark to celebrate this year ☺

The remaining slides are just practice questions, which you can work through in revision if you wish

We can look at past exam questions, or any of these in the two lectures I planned for this after Easter

You need to choose what you want me to go through, and let me know in advance (first come first served)

# References, pointers, parameters and return types

Revision and making sure this is clear
(People often struggle with this)

# Parameter revision

- Passing a parameter to a function by value makes a copy of it

```
void RefFoo1( int i ) { … }
```

- Passing a pointer to a function by value makes a copy of the pointer

```
void RefFoo1( int* pi ) { … }
```

- Passing a parameter to a function by reference gives a new name for the actual thing (no copy)
  - Acts like passing in a pointer

```
void RefFoo1( int& ri ) { … }
```

# Return type revision

- Returning something ‘by value’ makes a copy of the thing returned
  - Copy constructor is used for objects

```
int RefFoo1() { … return j; }
```

- Returning something ‘by reference’ returns the thing itself
  - No copy has to be made, but could be `const &` to **try** to avoid the original being altered

```
int& RefFoo1() { … return j; }
```

- Returning a pointer returns a copy of the pointer, which will point to the same thing

```
int* RefFoo1() { … return pj; }
```

- Note: You can return a reference to a pointer, as shown in the final lecture (not needed for exam!)

# Danger: References and pointers

```
// Return pointer to parameter – danger
int* PFoo1( int i ) { return &i; }
// Return reference to parameter - danger
int& RefFoo1( int i )  { return i; }

// Return pointer passed in - fine
int* PFoo2( int* pi ) { return pi; }
// Return reference passed in - fine
int& RefFoo2( int& i ) { return i; }

// Return pointer to local – danger
int* PFoo3( int i ) { int j = 2; return &j; }
// Return reference to local - danger
int& RefFoo3( int i ) { int j = 2; return j; }

// Return pointer to static local - fine
int* PFoo4( int i ) {static int k = 3; return &k;}
// Return reference to static local - fine
int& RefFoo4( int i ) { static int k = 3; return k;}
```

const

# const – a clarification and reminder

- **`const`** means that the *thing* is constant
- **`const`** variable
  - Cannot alter the value of the variable. A constant
  - Have to set value on initialisation – otherwise it is too late
- **`const`** on function parameters:
  - The function guarantees that it will not change the parameter that is passed in – if it does then compiler will give an error
- **`const`** on member function
  - The **`const`** member function guarantees not to change the object
  - i.e. the **`this`** pointer is constant
- **`const`** reference
  - Cannot alter the thing that is referenced
- **`const`** pointer (two types)
  - Constant pointer – pointer cannot be changed : **`char * const`**
  - Pointer to something that cannot be changed : **`const char *`**

# What is wrong with this code?

```
class C
{
public:
   float& get1()
   {
        return f;
   }
   float& get2() const
   {
        return f;
   }
   const float& get3() const
   {
        return f;
   }
private:
   float f;
};
```

- The code in the class C, to the left, will not compile.
- Which line(s) cause the compilation error and why?
- What clues can you get from the question?
- What are the likely possibilities?
- What could it be to do with?

# Answer

```
class C
{
public:
   float& get1()
   {
        return f;
   }
   float& get2() const
   {
        return f;
   }
   const float& get3() const
   {
        return f;
   }
private:
   float f;
};
```

- Answer: `get2()` will not compile
- It returns a reference to the float in the current object
- This means that the float could be changed, through the reference
- But the function is const, guaranteeing that it cannot be used to change the object
- `get1()` will work – it makes no guarantees
- `get3()` will work – it returns a const ref, i.e. the reference cannot be used to change the object

# Which lines will not compile?

```
class C
{
public:
   float& get1()
   { return f; }

   const float& get3() const
   { return f; }

private:
   float f;
};


int main()
{
   C ob;
   const C& ref = ob;

   float f1 = ob.get1();
   const float cf1 = ob.get1();
   float& rf1 = ob.get1();
   const float& crf1 = ob.get1();

   float f2 = ref.get1();
   const float cf2 = ref.get1();
   float& rf2 = ref.get1();
   const float& crf2 = ref.get1();

   float f3 = ob.get3();
   const float cf3 = ob.get3();
   float& rf3 = ob.get3();
   const float& rf3 = ob.get3();

   float f4 = ref.get3();
   const float f4 = ref.get3();
   float& rf4 = ref.get3();
   const float& rf4 = ref.get3();
}
```

# Consider the first four

```
class C
{
public:
   float& get1()
   { return f; }

   const float& get3() const
   { return f; }

private:
   float f;
};


int main()
{
   C ob;
   const C& ref = ob;

   float f1 = ob.get1();
   const float cf1 = ob.get1();
   float& rf1 = ob.get1();
   const float& crf1 = ob.get1();

   float f2 = ref.get1();
   const float cf2 = ref.get1();
   float& rf2 = ref.get1();
   const float& crf2 = ref.get1();

   float f3 = ob.get3();
   const float cf3 = ob.get3();
   float& rf3 = ob.get3();
   const float& rf3 = ob.get3();

   float f4 = ref.get3();
   const float f4 = ref.get3();
   float& rf4 = ref.get3();
   const float& rf4 = ref.get3();
}
```

# First four

- A non-`const float` reference is returned by `get1()`
- We can use this to initialise:
  - a float – i.e. a copy of the returned float
  - a constant float – i.e. a constand copy of the returned float
  - a float reference – it's fine by us if it changes it, the returned ref is not const
  - a constant float reference – guarantees not to change it, but we don't care even if it did

# Next four: float from a const ref

```
class C
{
public:
   float& get1()
   { return f; }

   const float& get3() const
   { return f; }

private:
   float f;
};


int main()
{
   C ob;
   const C& ref = ob;

   float f1 = ob.get1();
   const float cf1 = ob.get1();
   float& rf1 = ob.get1();
   const float& crf1 = ob.get1();

   float f2 = ref.get1();
   const float cf2 = ref.get1();
   float& rf2 = ref.get1();
   const float& crf2 = ref.get1();

   float f3 = ob.get3();
   const float cf3 = ob.get3();
   float& rf3 = ob.get3();
   const float& rf3 = ob.get3();

   float f4 = ref.get3();
   const float f4 = ref.get3();
   float& rf4 = ref.get3();
   const float& rf4 = ref.get3();
}
```

# Next four: float from a const ref

- All give compile error:

```
error: passing `const C' as `this'
argument of `float& C::get1()' discards
qualifiers
```

- What does this mean?
  - The **`this`** argument is the pointer to the current object and it is constant
  - But **`get1()`** is **not** a **`const`** function, so you cannot call it on a constant reference – it would allow the ref to be altered
- All of these four lines will fail to compile
  - the **`get1()`** fails – it's irrelevant what we try to do with it

# Third four

```
class C
{
public:
   float& get1()
   { return f; }

   const float& get3() const
   { return f; }

private:
   float f;
};


int main()
{
   C ob;
   const C& ref = ob;

   float f1 = ob.get1();
   const float cf1 = ob.get1();
   float& rf1 = ob.get1();
   const float& crf1 = ob.get1();

   float f2 = ref.get1();
   const float cf2 = ref.get1();
   float& rf2 = ref.get1();
   const float& crf2 = ref.get1();

   float f3 = ob.get3();
   const float cf3 = ob.get3();
   float& rf3 = ob.get3();
   const float& rf3 = ob.get3();

   float f4 = ref.get3();
   const float f4 = ref.get3();
   float& rf4 = ref.get3();
   const float& rf4 = ref.get3();
}
```

# Third four

- **`get3()`** returns a constant reference
- You can use it to initialise a float (a copy)
- You can store it in another constant reference
- You CANNOT just make a non-constant reference refer to it – would allow it to be altered
- This is the only line which fails:

```
float& rf3 = ob.get3();
```

- However, you could use **`const_cast`** to remove the **`const`**-ness

```
float& rf3 = const_cast<float&>(ob.get3());
```

# Last four

```
class C
{
public:
   float& get1()
   { return f; }

   const float& get3() const
   { return f; }

private:
   float f;
};


int main()
{
   C ob;
   const C& ref = ob;

   float f1 = ob.get1();
   const float cf1 = ob.get1();
   float& rf1 = ob.get1();
   const float& crf1 = ob.get1();

   float f2 = ref.get1();
   const float cf2 = ref.get1();
   float& rf2 = ref.get1();
   const float& crf2 = ref.get1();

   float f3 = ob.get3();
   const float cf3 = ob.get3();
   float& rf3 = ob.get3();
   const float& rf3 = ob.get3();

   float f4 = ref.get3();
   const float f4 = ref.get3();
   float& rf4 = ref.get3();
   const float& rf4 = ref.get3();
}
```

# Last four

- **`get3()`** returns a constant reference
  - This is safe, even when called on a constant object
    - We are guaranteeing not to change it anyway
  - Previously it failed in trying to return a **`float&`** from a constant object (using **`get1()`**)
  - Now it succeeds in getting a **`const float&`**, using **`get3()`**, since we are guaranteeing not to change anything – the reference that we get is **`const`**
- So, this is the only line which fails:

```
float& rf4 = ref.get3();
```

  - We are trying to assign a **`const`** reference (from **`get3()`**) to a non-**`const`** reference – which would lose the **`const`** property
  - Could use a **`const_cast`** to remove the **`const`**-ness

# Example past paper question

# 2008/2009 Q1b

It is required to call the function **`printf()`** if, and only if, the integer variables **`x`** and **`y`** are both nonzero.

Which (one or more) of the following would have the desired result?

```
I.   if ( x & y )          { printf( "i" ); }
II.  if ( x && y )         { printf( "ii" ); }
III. if ( x | y )          { printf( "iii" ); }
IV.  if ( x || y )         { printf( "iv" ); }
V.   if ( !(!x || !y) ) { printf( "v" ); }
```

# What is it testing?

- Do you know what the & operator does?
- What about the && operator?
- What is the difference between & and && ?
- How does ! work with |, ||, & and && ?

# struct and union sizes

# `struct` and `union` sizes

```
#pragma pack(1)

struct S1
{
   short s;
   char c;
};

struct S2
{
   long l1;
   unsigned long l2;
};

struct S3
{
   S1 a1;
   S2 a2;
};
```

```
union U1
{
   char c;
   short s;
   long l;
};

union U2
{
   long l;
   U1 u;
};

int main()
{
   printf( "S1 size %d\n", sizeof(S1) );
   printf( "S2 size %d\n", sizeof(S2) );
   printf( "S3 size %d\n", sizeof(S3) );
   printf( "U1 size %d\n", sizeof(U1) );
   printf( "U2 size %d\n", sizeof(U2) );
}
```

**Q: What is the output assuming that:**
**`sizeof(short)` is 2**
**`sizeof(long)` is 4**

**Reminder: by definition `sizeof(char)` is 1**

# `struct` and `union` sizes

```
#pragma pack(1)

struct S1
{
   short s;
   char c;
};

struct S2
{
   long l1;
   unsigned long l2;
};

struct S3
{
   S1 a1;
   S2 a2;
};
```

2+1 = 3

4+4 = 8

3+8 = 11

```
union U1
{
   char c;
   short s;
   long l;
};

union U2
{
   long l;
   U1 u;
};

int main()
{
   printf( "S1 size %d\n", sizeof(S1) );
   printf( "S2 size %d\n", sizeof(S2) );
   printf( "S3 size %d\n", sizeof(S3) );
   printf( "U1 size %d\n", sizeof(U1) );
   printf( "U2 size %d\n", sizeof(U2) );
}
```

max(1,2,4) = 4

max(4,4) = 4

**What is the output assuming that:**
**sizeof(short) is 2**
**sizeof(long) is 4**

# template functions

# Question

- A function is required which will take two integers as parameters and which will use the ternary operator to determine the minimum of the two parameters passed in
- It should return this minimum value, as an `int`
- Provide an implementation of a function called `mymin()` which will perform as described above

# Answer

- Return type is **`int`**
- Parameter types are both **`int`**
- Ternary operator is the **`? :`** operator

```
int mymin( int i, int j )
{
    return i < j ? i : j;
}
```

- Hint: Look elsewhere in the exam questions if you cannot remember the details for a function definition – there are bound to be a lot of them

# Question

- Convert your answer to the previous part into a template function which will accept two parameters of the same type and return a value of the same type

# Reminder: template functions

**How to make a template function:**

1. First generate function for specific type(s)
2. Next replace all copies of the types with template types
3. Finally, add the keyword **`template`** at the beginning and put the type(s) in the **`<>`** with keyword **`typename`** (or **`class`**)

# Answer

1. Add the initial `template <typename T>`
2. Convert all the types to `T`

- Initial version:

```
int mymin( int i, int j )
{
   return i < j ? i : j;
}
```

- Template version:

```
template <typename T>
T mymin( T i, T j )
{
   return i < j ? i : j;
}
```

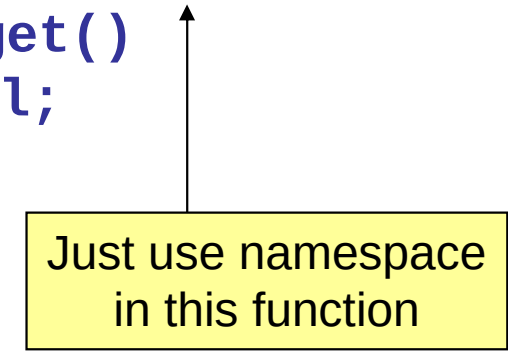
# Question

You have been provided with a simple class, C, and some functions which use it. However, the code will not compile. Correct the class definition.

```
class C
{
public:
   // Constructor
   C(int i = 0) : i(i)
   {}
   // Get value
   int get()
   { return i; }
   // Set value
   void set(int i)
   { this->i = i; }
private:
   int i;
};
```

```
void output( const C& c )
{
   using namespace std;
   cout << c.get()
        << endl;
}

int main()
{
   C c1(2), c2;
   output( c1 );
   output( c2 );
}
```



# Answer

The parameter **c** of **output()** is passed by *constant* reference. It is used to call **get()**. **get()** was not **const**, so didn't guarantee to not change **c**

```
class C
{
public:
   // Constructor
   C(int i = 0) : i(i)
   {}
   // Get value
   int get() const
   { return i; }
   // Set value
   void set(int i)
   { this->i = i; }
private:
   int i;
};
```

```
void output( const C& c )
{
   using namespace std;
   cout << c.get()
        << endl;
}

int main()
{
   C c1(2), c2;
   output( c1 );
   output( c2 );
}
```

# Function pointers

# Question

- What is the output of the following code?

```
#include <iostream>

int mult2( int i )
{
   return i*2;
}

int square( int i )
{
   return i*i;
}

int main()
{
   using namespace std;

   int (*f)(int) = &mult2;
   int (*g)(int) = &square;

   for (int i = 0; i < 5 ; i++)
       cout << (*f)(i)
            << endl;

   for (int i = 0; i < 5 ; i++)
       cout << (*g)(i)
            << endl;

   for (int i = 0; i < 5 ; i++)
       cout << (*g)((*f)(i))
            << endl;

   f = g;

   for (int i = 0; i < 5 ; i++)
       cout << (*f)(i) << endl;
}
```

# Answer: function pointers

- What is the output of the following code?

```
#include <iostream>

int mult2( int i )
{
   return i*2;
}

int square( int i )
{
   return i*i;
}

int main()
{
   using namespace std;
```

```
int (*f)(int) = &mult2;
int (*g)(int) = &square;

for (int i = 0; i < 4 ; i++)
     cout << (*f)(i)
          << endl;

for (int i = 0; i < 4 ; i++)
     cout << (*g)(i)
          << endl;

for (int i = 0; i < 4 ; i++)
     cout << (*g)((*f)(i))
          << endl;

f = g;

for (int i = 0; i < 4 ; i++)
     cout << (*f)(i) << endl;
```

Output:

0
2
4
6

0
1
4
9

0
4
16
36

0
1
4
9

**f=mult2**

**g=square**

**g=square**
**f=mult2**
**square(mult2(i))**

**f=square**

static

# static – a reminder

- Static applies to three things:

1. Local variables
   - Maintain their value beyond function calls
   - Not stored on the stack
2. Global functions and variables
   - Hides them within a file, file access only
   - Does not show them to the linker
3. Member functions/data (in a class)
   - Associated with the class rather than instances of the class (next week)
   - i.e. functions have no `this` pointer
     - So cannot access non-static member data – they would not know which object to affect

# Question

- What is the output of the following code?

```
int f1( int i )
{
  int j = ++i;
  return j;
}

int f2( int i )
{
  static int j = ++i;
  return j;
}
```

```
int main()
{
  using namespace std;
  int i;

  for (i=0 ; i<5 ; i++)
     cout << f1(i)
          << endl;

  for (i=0 ; i<5 ; i++)
     cout << f2(i)
          << endl;
}
```

# Answer: static local variables

- What is the output of the following code?

```
int f1( int i )
{
   int j = ++i;
   return j;
}

int f2( int i )
{
   static int j = ++i;
   return j;
}
```

Output:
1
2
3
4
5
1
1
1
1
1

```
int main()
{
   using namespace std;

   for ( int i = 0
               ; i < 5
               ; i++ )
      cout << f1(i)
           << endl;

   for ( int i = 0
               ; i < 5
               ; i++ )
      cout << f2(i)
           << endl;
}
```

# Question

- What is the output of the following code?

```
int j;
static int k;

int f1( int i )
{
   j = ++i;
   return j;
}

int f2( int i )
{
   k = ++i;
   return k;
}
```

```
int main()
{
   using namespace std;

   for ( int i = 0
               ; i < 5
               ; i++ )
      cout << f1(i)
           << endl;

   for ( int i = 0
               ; i < 5
               ; i++ )
      cout << f2(i)
           << endl;
}
```

# Answer: static global variables

- Static globals are just not accessible outside of the file. Within the file they make no difference!

```
int j;
static int k;

int f1( int i )
{
   j = ++i;
   return j;
}

int f2( int i )
{
   k = ++i;
   return k;
}
```

Output:
1
2
3
4
5
1
2
3
4
5

```
int main()
{
   using namespace std;

   for ( int i = 0
               ; i < 5
               ; i++ )
      cout << f1(i)
           << endl;

   for ( int i = 0
               ; i < 5
               ; i++ )
      cout << f2(i)
           << endl;
}
```

What is wrong with this code?

# Problems with this code?

```
#include <cstdio>
#include <cstring>
int main()
{
     char* s1 = new char[30];
     char s2[] = "is not";
     const char* s3 = "likes";
     s3 = "allows";
     strcpy( s2, s3 );
     sprintf( s1, "%s %s %s using functions.",
             "C++", s2, "fast code" );
     printf( "String was : %s\n", s1 );
     delete s1;
}
```

# Answer **2008/2009 Q4c**

- delete should be delete[] (i.e. array delete)
- Array length is not big enough for the second usage. (Needs to be 40+)
- Note: The const is fine