G52CPP
C++ Programming
Lecture 5

Dr Jason Atkin

http://www.cs.nott.ac.uk/~jaa/cpp/g52cpp.html

Lectures so far

- Functions:
  - Declarations and definitions
- Pointers
  - **&** Address-of
  - ***** De-referencing
  - Array names are pointers to first element
  - Pointers can be treated as arrays
  - Pointer arithmetic
  - Passing pointers as parameters

Pointer Arithmetic Summary

- Pointers store addresses
  - You can increment/decrement them (++,--)
    - Changing the address that is stored
  - **You can also add to or subtract from the value of a pointer**
  - They move in **multiples of the size** of the **type that they THINK they point at**
  - e.g.: If a **`short`** is 2 bytes, then incrementing a **`short*`** pointer will add 2 to the address
  - This is very useful for moving through arrays

This lecture

- The strcpy() example
- The stack
- Local, global and static variables
- Variable shadowing

Implementing strcpy

How we could implement strcpy

```
char src[] = {'C',' ',
     's','t','r',0};
char dest[7];
strcpy( dest, src );
```

```
char* mystrcpy(
   char* dest, char* src )
{
   char* p = dest;
   char* q = src;
   while ( *p++ = *q++ )
      ;
   return dest;
}
```

Address	Value	Name
1000	‘C’	src[0]
1001	‘ ’	src[1]
1002	‘s’	src[2]
1003	‘t ‘	src[3]
1004	‘r’	src[4]
1005	0	src[5]
6000	?	dest[0]
6001	?	dest[1]
6002	?	dest[2]
6003	?	dest[3]
6004	?	dest[4]
6005	?	dest[5]
6006	?	dest[6]

Note: *p++ is equivalent to *(p++) (post-increment has higher precedence)

Reminder: Operator Precedence

- Operators are evaluated in a specific order
  - Highest operator precedence applies first
- Examples (highest to lowest, not complete)

Increasing precedence

(), [], ++, --	Grouping, array access, post increment/decrement
++, --, *, &	Pre-increment, dereference, address of (right to left)
*, /, %	Multiplication, division, modulus
+ -	Addition, subtraction
<, <=, >, >=	Comparison
==, !=	Comparison: equal to, not equal to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Ternary conditional
=, +=, -= etc	Assignment and '… and assign' (right to left)

What actually happens when a function is called…

Process structure in memory

Stack

Data area that grows downwards towards the heap

LIFO data structure, for local variables and parameters

Heap

Data area that grows upwards towards stack

Specially allocated memory (malloc, free, …, probably new, delete)

Data and BSS (uninitialised data) segment

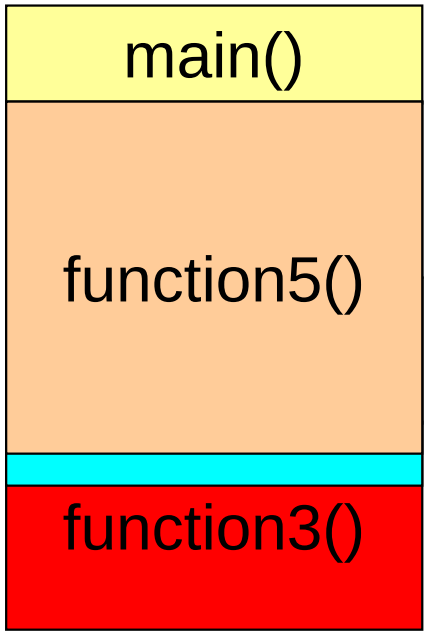
Read-only:	Constants	String literals
Read/write:	Global variables	Static local variables

Code (or text) segment

The program code

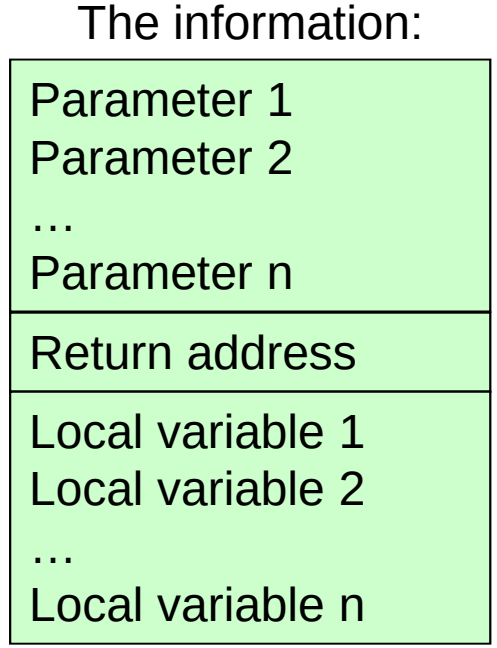
The stack

- A stack is a last-in-first-out (LIFO) structure
- Like a stack of books
  - You add to the top
  - You take from the top
- Function calls (stack frames) are stored on a stack in memory
- Aside: Note that **most** stacks go down in memory addresses
  - i.e. the stack frame for the new function is lower in memory



The stack frame

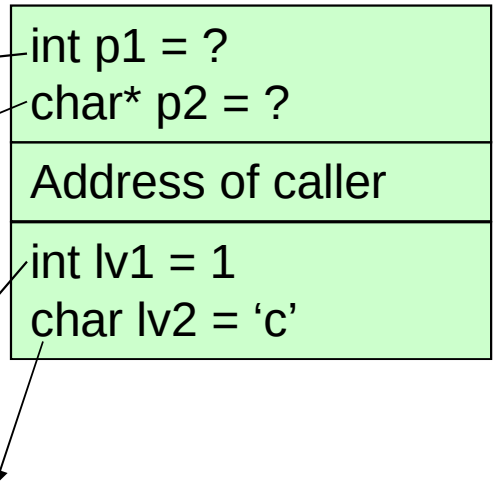
- When a function call is made, necessary information is collected together
  - Who called the function?
    - So the program knows where to return to when the function ends
  - What parameters were supplied?
  - Space to put the return value?
    - If not void, and not returned in register
  - Somewhere to store local variables while they are needed
- Values are stored together in a ‘stack frame’



Example stack frame

- For example:

```
int myfunc(
  int p1;
  char* p2)
{
  int lv1 = 1;
  char lv2 = 'c';
  return 4;
}
```



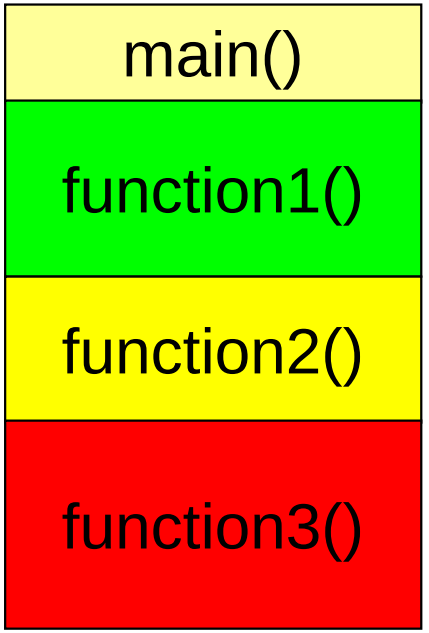
The information:

Parameter 1 Parameter 2 … Parameter n
Return address
Local variable 1 Local variable 2 … Local variable n

Lifetime of local variables

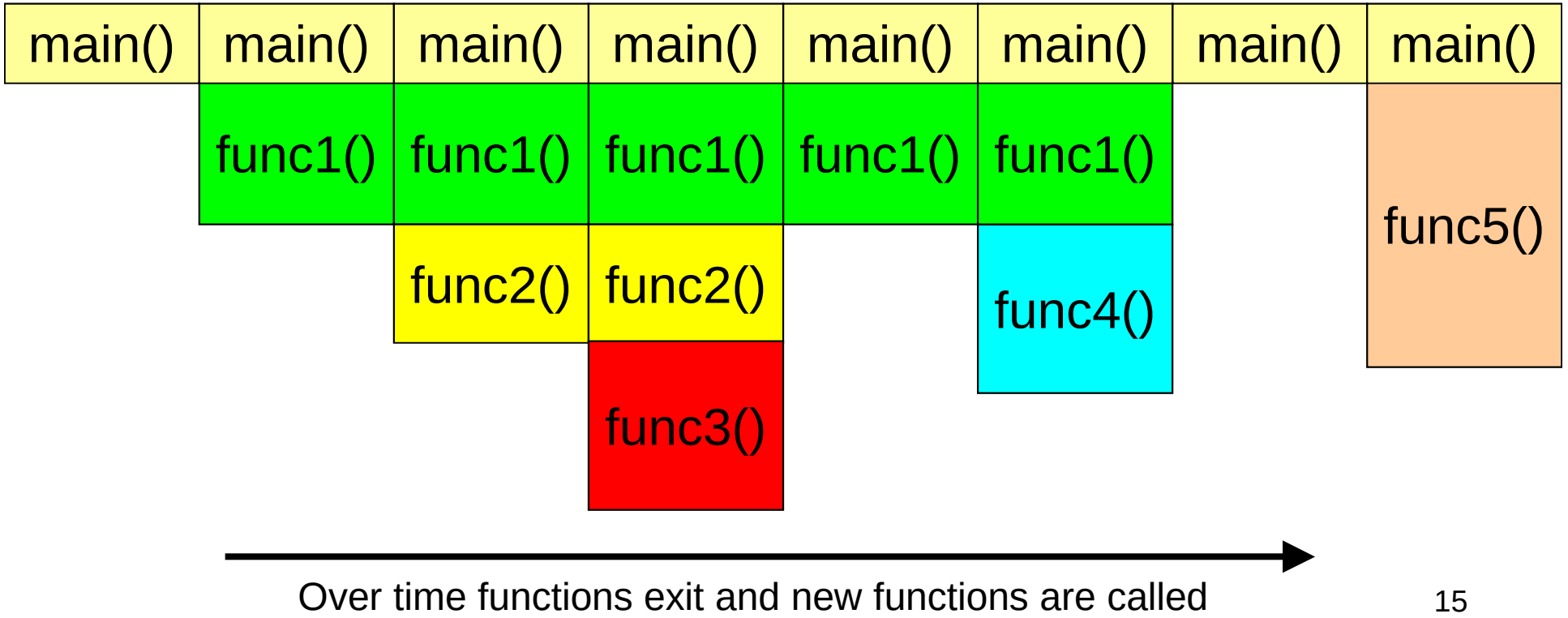
Disadvantage of local variables

- Local variables exist for the duration of the stack frame they are in
- i.e. while the program is inside the block they are declared in
  - Or any function called from that block



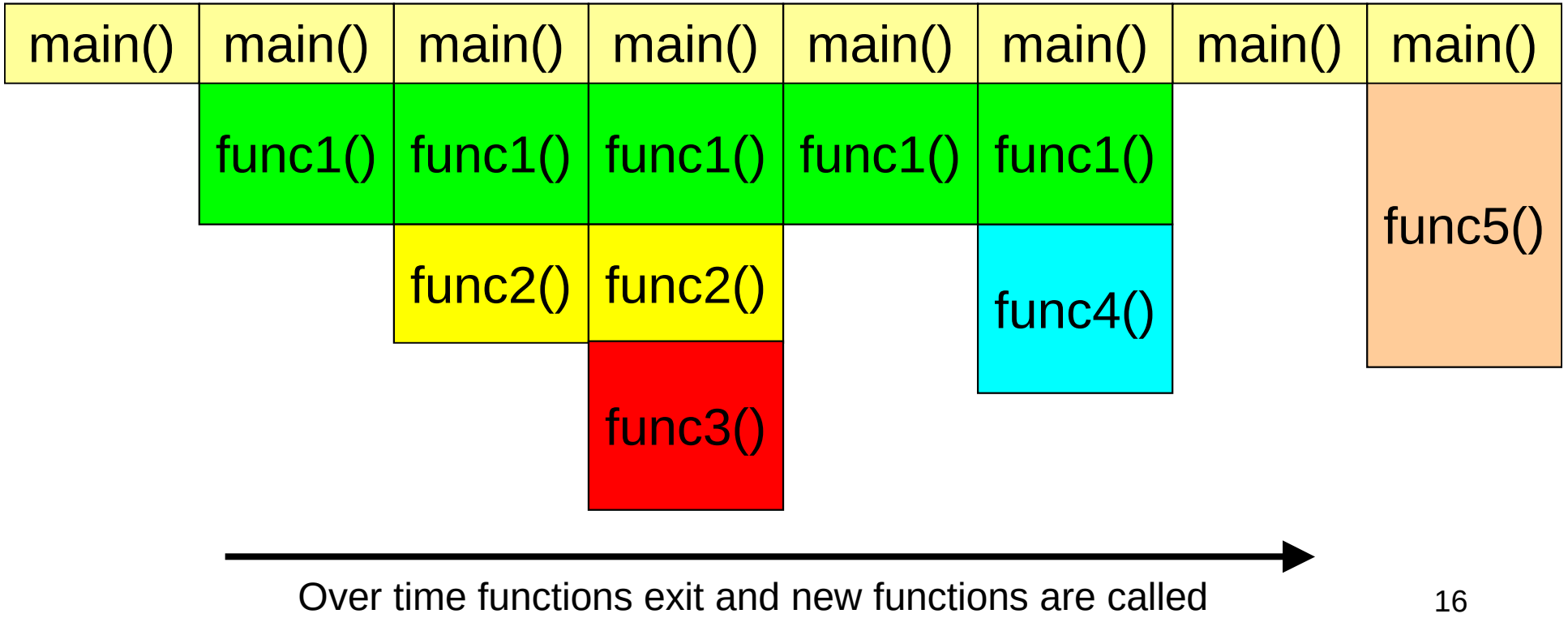
The stack

- If we declare a local variable in **`func1()`** how long will it last?
- When (from where) should we be able to use it?
- If we keep a pointer to it, where can we use the pointer?



Local variables' lifetimes

- The local variables for functions 2 and 3 are overwritten by those for function 4
- The local variables for functions 1 and 4 are overwritten by those for function 5
- So your local variables **may** be overwritten as soon as the block that they are defined in ends



Danger!

- Your local variables only exist for as long as the block in which they are defined
- DO NOT ACCESS THEM AFTER THAT
  - e.g. Through pointers
- DO NOT ASSUME THAT THEY KEEP THEIR VALUE AFTER THE FUNCTION ENDS

Global and static local variables

Or:

“Since my local variables get destroyed, where can I put things I need to keep?”

Global variables

```
int var = 1; /* Global variable */

void myfunc()
{
   printf( "Var = %d in myfunc\n", var );
}

int main( int argc, char* argv[] )
{
   myfunc();
   printf( "Var = %d in main\n", var );
   return 0;
}
```

Variable declared **outside** of **all** functions.

Available to all functions in the file.

Global variables

- Defined outside of all functions
- Global variables last for the duration of the program
  - Remember: local variables last for the duration of the block they are defined within
- All functions in the file can access globals
  - Values are maintained between function calls
- Not available in Java!
  - Static member variables have some similarities

Question:

- When might you want to use a global variable instead of local variable?

Hint: (example function)

```
int myfunc( int a, int b,
  int i1, int i2, char c1, char c2,
  char* str, long* pl,
  short* ps, …etc… )
{
  … do something with all parameters …
  return …;
}
```

When is it valid to use global variables?

- When doing so makes the structure EASIER to understand
  - WARNING: they often make it harder to understand and debug!
- E.g. If you need to access variables across multiple functions you have two options:

1. Pass the variables in as parameters to EVERY function that needs them
   - (Maybe not as separate variables though, see '**`struct`**' later)
   - Need to pass a pointer to them if they have to be altered
2. Make them globals
   - Then they will last for the duration of the program

- Pros and cons:
  - Globals can be altered from anywhere – harder to keep a track of what is altering them, and hence may be harder to debug problems
  - Locals can mean passing many different variables

Static local variables

- Local variables can be **`static`**
  - Means not moving/unchanging
  - NOT the same as static member variables!
  - NOT the same as **`const`** (see later)
- Static local variables remember their value between function calls
  - Like global variables
- But, you can only access them (by name) inside the one function they are defined in
  - Unless you keep a pointer to them

Example of static local variable

```
void foo()
{
   static int count = 0;
   count++;
   printf( "Value of static count is %d\n", count );
}
void bar()
{
   int count = 0;
   count++;
   printf( "Value of count is %d\n", count );
}
int main( int argc, char* argv[] )
{
   int i;
   for ( i=0 ; i < 5 ; i++ )
       foo();
   for ( i=0 ; i < 5 ; i++ )
       bar();
   return 0;
}
```

Static variable remembers its value
Initialisation only occurs in the first function call

Non-static creates a new variable for each call
Initialisation once for each new variable / function call

static_locals.cpp

Static variables are stored in the same place as global variables
i.e. NOT on the stack

Summary: global vs local variables

- Global variables (defined outside of functions)
  - All functions in the file can access them
    - Values are maintained between function calls
  - Can (optionally) be hidden from other files
    - (See static global variables)
- Local variables (defined within a function)
  - Declared within blocks within a function
    - The same as local variables in Java
  - Non-static local variables ‘die’ when the block ends
- Static local variables
  - Maintain value between function calls
  - Have lifespans like global variables

Variable shadowing

Putting other variables with the same name into the shadows

(hiding them)

Variables and shadowing

```
int var = 1; /* Global variable */

int myfunc( int var )
{
    printf( "Var = %d at start of myfunc\n", var );
    {
        int var = 3;
        printf( "Var = %d in sub-block 1\n", var );
    }
    printf( "Var = %d in myfunc\n", var );
    return var;
}

int main( int argc, char* argv[] )
{
    myfunc( var + 1 );
    printf( "Var = %d in main\n", var );
    return 0;
}
```

varshadow.cpp

Variables and shadowing

```
int var = 1; /* Global variable */

int myfunc( int var )
{
    printf( "Var = %d at start of myfunc\n", var );
    {
        int var = 3;
        printf( "Var = %d in sub-block 1\n", var );
    }
    printf( "Var = %d in myfunc\n", var );
    return var;
}

int main( int argc, char* argv[] )
{
    myfunc( var + 1 );
    printf( "Var = %d in main\n", var );
    return 0;
}
```

varshadow.cpp

Variables can be global or local to a function
This var exists for the life of the program.

A function parameter.
This 'var' exists for the life of the function.
It shadows the global var.

A block within a function.
This 'var' exists for the block and shadows the parameter

Output:

```
Var = 2 at start of myfunc
Var = 3 in sub-block 1
Var = 2 in myfunc
Var = 1 in main
```

Pointers to variables

Using variables via pointers

- Even if something is not visible, you can use a pointer to it, as long as it exists
- e.g. return a pointer to some static local variable and use it elsewhere
  - Do not do this with non-static local variables – they will not exist after the function ends/stack frame vanishes
- Globals exist for the lifetime of the program
  - So you can use pointers to them at any time
- A static local variable exists for lifetime of program
  - From at least the first usage to the end of the program
  - So you can use pointers to them at any time

Example

```
int iGlobal = 1;

int* funcstatic()
{
   static int iStatic = 10;
   iStatic++;
   return &iStatic;
}
int* funclocal()
{
   int iLocal = iGlobal;
   iLocal++;
   return &iLocal;
}

int overwrite()
{
   int iOverwrite1 = 20;
   int iOverwrite2 = 30;
   iOverwrite1 = iOverwrite2;
   return iOverwrite1;
}
```

```
int main(int argc, char* argv[])
{
   int* piStatic = funcstatic();
   int* piLocal = funclocal();
   funcstatic();
   funclocal();

   printf( "%d %d %d\n", iGlobal,
        *piStatic, *piLocal );

   overwrite();

   printf( "%d %d %d\n", iGlobal,
        *piStatic, *piLocal );

   return 0;
}
```

visibility.cpp

Please have a go at the previous example

The sample code can be found in the cpp samples on the labs page

The dangers of pointers to local variables

- Do not refer to data **on the stack** outside the function
  - This means **local variables** or **actual parameters**
- Things to avoid:
  - Returning a pointer to a local variable or parameter
  - Storing the address of a local variable or parameter
- You **CAN** refer to them, but **SHOULD NOT**
- This is a very common (and major) early C/C++ mistake
  - The variable no longer exists, and the memory may be reused
  - Using your pointer will corrupt whatever is using the memory now
  - Until the memory gets reused, you won't see the problem
- Not a problem in Java
  - You cannot get a reference/pointer to something that is on the stack

Visibility is different from lifetime

- Just because a variable exists, doesn't mean that you can access it
  - Globals : access from anywhere
    - May be shadowed by parameters or local variables
    - In C++ (not C) you can use Scope Resolution to access globals when they are shadowed
    - Can be 'hidden' within a file
  - Static local variables
    - Only access from inside the function
    - Exist all of the time, like globals
- Do not use a pointer to something, if the thing it points to no longer exists

What now…

- Labs now
  - Go to the labs web page
- Demo lecture this afternoon
  - Building a Zombies program
    - String manipulation
    - Some simple I/O
    - malloc() example
    - Loops, etc

Next lecture

- Structures and unions
- sizeof()