RMIT UNIVERSITY
Computer Science and Information Technology

Advanced Programming Techniques

(a.k.a. Programming in ANSI / ISO C)

Module 11 – Generics and Reusability

Updated by Paul Miller, Shaahin Madani, Ken Gardiner

Reusability

- Why re-invent the wheel?
- It is sometimes possible to reuse elements from a previous software lifecycle

- For example:
  - an existing software design might be transformable into one suitable for the new situation
  - existing code might be reusable for the new application

Code Reusability

- Common characteristics of re-useable software include:
- Modularity - divide required funcationality into small independent units that perform a logically separate functionality, with each module having a well defined interface.
- Loose-coupling – where components don't need to know the definition of other components. This allows components to be replaced by other components that provide the same service.

Code Reusability

- High cohesion - elements inside a module belong together, all serving the purpose of the module.
- Information hiding – where implementation details are hidden behind a stable public interface.

Reuse depends on commonalities of function across multiple applications.

Generics and Reusability

- Very often many functions, and even modules, could be applied to other applications (i.e. reused) if only the functionality was **more carefully parameterised** and/or if it was not hard-coded to work with **just one specific data type**.

- The basic idea behind *generics* is to write code that will **work with a variety of different types** – and in that sense is *generic*.
  - E.g. a generic list that you can give any type to

Generics and Reusability (Continued)

- Writing generic code, or indeed just reusable code, in the more general sense, **is not always straight forward**.
  - Also, being 'too generic' may make code and API too complex, in order to handle edge-cases.
- Generic functions, whilst often making subsequent reuse possible and therefore **reducing development time**, is not without drawback.
- Slight reduction is execution time is not uncommon, but by the same token, is not always the case.

Generics and Reusability (Continued)

- There is also an issue of **type safety**.
- The use of generic code which may apply to a variety of types means that **the compiler is no longer as much help in ensuring that types are dealt with correctly**
- This becomes the responsibility of the programmer.

Generic Programming in C

- One approach to generic programming in C is provided via the use of `void *`, the `sizeof` operator, function pointers and so on.

- There are some pitfalls with generic programming and it does enforce **greater discipline** on the programmer.

Datatypes and Generic Programming

- Let's think for a moment about a standard variable declaration in C. They follow syntax like:

```
TYPE var1, var2, var3;
```

- and for arrays:

```
TYPE arrname[NUMBER];
```

- What information does the compiler get from the type name? How is this used?

The void * type

- A **type-name** specifies to the compiler **how much space an item of that type will take up**.
- `void *` **does not** provide this information, so we need to provide this ourselves. Unlike other types, the `void *` type does **not** specify the **type of data** that is being pointed to.
- Therefore, the **size of data** pointed to is not specified.

The void * type

- What that means is that we can treat `void *` as a **generic type of pointer**. Much as in many object oriented languages the Object class (java.lang.Object) is considered to be a generic type.

- We can assign to or from any other type of pointer to a `void *`.

Generic Functions in the Standard Library

- There are a variety of functions in the C standard library that use a generic approach. You have already used some of them, but may not have thought too much about them.

- Any function that **either accepts void * as a parameter** or **returns a void *** was developed using a generic approach.

Generic Functions in the Standard Library

- Let's start by looking at the memory allocation functions:

```
void * malloc(size_t size);
void * realloc(void * ptr, size_t size);
void * calloc(size_t nmemb, size_t size);
void free(void * mem);
```

Generic Functions in the Standard Library (Continued)

- A generic function should be **transparent to whichever datatype is being used**.

- There are also a number of these in the I/O library. Let's look at a couple:

```
size_t fread(void * ptr, size_t size,
             size_t nmemb, FILE * stream);

size_t fwrite(const void * ptr,
              size_t size, size_t nmemb,
              FILE * stream);
```

Generic Functions in the Standard Library (Continued)

- The `fwrite` function allows us to write any kind of data to a binary file.
  - We need to pass in the size of each element and count of the number of elements to write.

- This is a common approach used in generic functions – we need to know **how large each element is** so that **we can iterate over them**. This approach (by having a size and a count) also allows for either a pointer to an individual data item or an array.

Generic Functions in the Standard Library (Continued)

- The problem with this approach is that the data file saved in this way may not be portable between different environments due to "endianness" - that is, the order of bytes of an item in memory.

Generic Functions in the Standard Library (Continued)

- Likewise, `fread` allows us to specify the number of elements and their size to be read in
  - again the byte order will be architecture dependent so moving between machines may cause data corruption.

- Again, it is your responsibility to ensure you know the type of data being read / written and that you don't read in using sizes other than how the data was written.

qsort()

- The standard library includes an implementation of the quicksort algorithm.
- The quicksort algorithm takes a simple divide and conquer approach – we select a pivot element and move everything with a lower value than the pivot to an earlier position in the array and everything with a larger value to a higher position in the array.
- When this algorithm is applied recursively to an array, we end up with a completely sorted array.

qsort() (Continued)

- The qsort() function implements this algorithm with the use of generics to allow any type of data to be sorted. Its prototype is:

```
void qsort(void *base, size_t nmemb,
           size_t size,
           int (*compar)(const void *,
           const void *));
```

- The base is the starting address of the array, nmemb is the number of elements, size is the size of each element, and there is a function pointer to a comparison function.

qsort() (Continued)

- Let's assume we have an array of integers. We could implement int_cmp for the comparison of integers as follows:

```
int int_cmp(const void * first, const void * second)
{
    int i_first = *((int*)first);
    int i_second = *((int*)second);
    int cmp = i_first - i_second;
    return cmp;
}
```

qsort() (Continued)

- We can then call qsort() to do our sorting for us like this:

```
qsort(array, LEN, sizeof(int),
      int_cmp);
```

- And our array will be sorted for us. You can see from this how powerful generic programming in C is. Using this one predefined function and a custom function for comparison we are now able **to sort an array of any kind of data**.

bsearch()

- This is a binary search function available in standard library.
- Again, this is a divide and conquer algorithm – at each step we divide the search space in two:
  - We start in the middle of the array. If the item we are looking for is less than the current one we go left to the middle element of the left half. If the item we are looking for is greater than the current element, we go right to the middle element of the right hand of the array.
  - This continues until we find the element we are looking for or we have run out of elements to search.

bsearch() (Continued)

- What this means is that if the elements are **not sorted**, then we can't find the data and the efficiency built into bsearch becomes a liability.

- bsearch is given a key to look for. Returns a pointer to an element that matches the key, or null if not found

bsearch() (Continued)

- The bsearch() function implements this algorithm with the use of generics to allow any type of data to be sorted. Its prototype is:

```
void* bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *,
              const void *));
```

Generic Memory Functions

- Along with the generic functions for writing binary data, there are a series of functions to allow us to do **memory manipulations on data without needing to know about the type**.

- These functions come mostly from *string.h* so, in a sense, they are "string manipulation functions". However, they make **no assumption about null termination** (`\0` at the end).

Generic Memory Functions (Continued)

```
void *memcpy(void *dest, const void *src,
             size_t n);
```

- Copies n bytes from src to dest and returns a pointer to dest. This function is actually **faster** than strcpy as it doesn't need to check for the '\0' character after each copy.

```
void *memchr(const void *s, int c, size_t n);
```

- Scans `s` until it comes across the value `c`. It then returns the address of this first occurrence.

Generic Memory Functions (Continued)

```
int memcmp(const void *s1, const void *s2,
           size_t n);
```

- Compares s1 to s2 for n bytes. Returns a number on a similar basis to `strcmp()`. This function is faster than `strcmp()` as **it does not need to compare the value with the nul character**.

Generic Memory Functions (Continued)

```
void *memccpy(void *dest, const void *src,
              int c, size_t n);
```

- Copies up to `n` bytes from `src` to `dest`, stopping after the first `c` has been copied. It returns a pointer to `dest`. This is similar in functionality to `strncpy()`.

Generic Memory Functions (Continued)

```
void *memmove(void *dest, const void *src,
              size_t n);
```

- Moves the memory chunk pointed to by `src` to `dest`, moving `n` bytes. Here, unlike with `memcpy` and `memccpy`, `src` and `dest` can overlap.

```
void *memset(void *s, int c, size_t n);
```

- Sets the first `n` bytes in `s` to the value `c`.

A Generic List (Continued)

- In previous weeks when we have looked at lists, we have used the personlist data structure with a basic person type. If we were going to manage a dynamic list with such a data structure, we might have a declaration such as:

```
struct list
{
    struct node * head;
    unsigned size;
};
```

A Generic List (Continued)

And the definition for a node would be:

```
struct node
{
    struct node * next;
    person * data;
}
```

A Generic List (Continued)

- Let's make some changes for our list to be generic.
- Firstly, we'll change our data to `void *` to allow for any data type:

```
void * data;
```

- As we have discussed previously there are some things that we don't know about with `void *` and so we need to use some helper functions.
- Let's add a comparison function to our list struct as well as a free function. Our structs will now look like the next slide.

A Generic List (Continued)

```
struct node {
    void * data;
    struct node * next;
};

struct list {
    struct node * head;
    unsigned size;
    int (*cmp)(const void*, const void*) cmpfn;
    void (*data_free)(void*) freefn;
};
```

A Generic List (Continued)

- Let's initialise the function pointers in our struct as part of list initialisation. So what might the function prototype for list_new look like?

```
struct list * list_new(
                int (*cmp)(void*, void*),
                void (*free_data)(void*));
```

A Generic List (Continued)

```
struct list * list_new(
                int (*cmp)(void*, void*),
                void (*free_data)(void*));
```

- The first parameter is a pointer to a function that can compare to data elements (needed since the list is ordered)
- The second parameter is a pointer to a function for freeing a data element

A Generic List (Continued)

- Given the list is a sorted list, what should be changed in the *add* function?
- Comparison of one data item with another data item needs to be decoupled from *add* to make the code able to compare various types
- Function *add* needs use the user-provided function (*cmp*) for performing such a comparison.