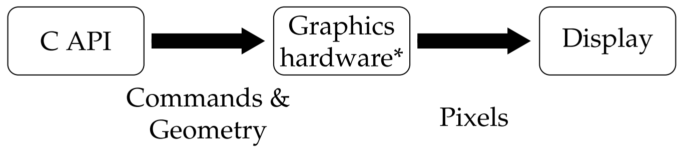
External Libraries

1 External libraries

- One of the reasons why C is so popular is the huge collection of tried and tested libraries available across many different computing platforms. E.g. OpenGL



- Commands from your program are sent by the API to the graphics hardware which generates pixels for display

*in OpenGL the hardware behaves as a state machine

2 OpenGL programming

- On its own OpenGL is:
  1. Low level
  2. O/S independent
- Hence it is usually used with:
  - GLU a utility library with high level shape support
  - GLUT utility library for window creation and I/O

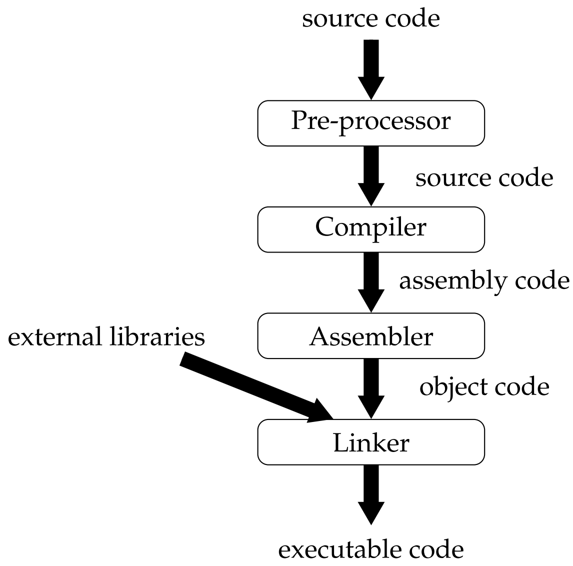
3 Usage of libraries

- If a library is **statically linked** then a copy of the library is included in the executable
  - No need to worry about what version of the library you have
- C/C++/assembly can be combined
- Often bound to other languages e.g. php, XML, curl
- Many of these libraries will be *dynamically linked*

4 Dynamic vs static linking

- *Dynamic* linking takes place at run-time not build-time
  + Reduces filespace demands (bloat) by keeping only one copy of the library
  + Only one copy of the library is loaded into memory
  + Can help with updates e.g. for security
  - Using dynamic linking can be slightly slower than static linking
- LGPL (Lesser Gnu Public License) often used
- Dynamic libraries are called differently by OSs
- Linux: shared objects (`.so`)
- Windows: Dynamic Link Libraries (`.dll`)
- OSX: `.dylib`
- Can lead to "DLL Hell": many versions of the same dynamic library
- Best to include version number with library

5 Compilation Model



6 Creating a static library

- A static library is effectively just an archive containing object (`.o`) files and is created with the archiver `ar`.
- On UNIX, static libraries use the `.a` extension.

```
gcc -c linkedlist.c -o bin/static/linkedlist.o
gcc -c anotherfile.c -o bin/static/anotherfile.o
ar rcs bin/static/libLL.a bin/static/linkedlist.o \
    bin/static/anotherfile.o
```

- To link statically:
  - Use the `-L` flag to list a (non-standard) directory where the library can be found
  - Use the `-l` flag to give the name of the library. Note that it assumes the library starts with `lib` and has the extension `.a` (static) or `.so` (dynamic)

```
gcc  main.o  -Lbin/static -lLL -o bin/main-static
```

- Can now run

```
bin/main-static
```

7 Loading/unloading a library

- We can add code to see when a library is loaded into memory and when it is unloaded.
  - N.B. This is a `gcc` extension and might not work on other compilers.

```
void __attribute__ ((constructor)) initLibrary(void){
 printf("Library is being loaded\n");
}

void __attribute__ ((destructor)) cleanUpLibrary(void){
 printf("Library is being exited\n");
}
```

8 Creating a shared library

- Objects files for a shared library need to be compiled with the `-fPIC` option
- On UNIX, static libraries use the `.so` extension.
  - PIC="Position Independent Code", since we don't know where in memory the library will be loaded at run-time

```
gcc -fPIC -c linkedlist.c -o bin/dynamic/linkedlist.o
gcc -fPIC -c anotherfile.c -o \
    bin/dynamic/anotherfile.o
gcc -shared bin/dynamic/linkedlist.o \
    bin/dynamic/anotherfile.o -o bin/dynamic/libLL.so
```

- To link dynamically:

```
gcc   main.o  -Lbin/static -lLL -o bin/main-static
```

- If we try to run it, we get an error:

```
bin/main-shared: error while loading shared libraries:
    libLL.so: cannot open shared object file: No such
    file or directory
```

- Need to tell the operating system where to find the library:

```
# In bash:
LD_LIBRARY_PATH=`pwd`/bin/dynamic/:$LD_LIBRARY_PATH
# In tcsh (the default shell on mira):
setenv LD_LIBRARY_PATH \
    `pwd`/bin/dynamic:$LD_LIBRARY_PATH
```

- (Note that ` above is a backtick)

9 Function Pointers

- We've seen pointers to variables. We can also have pointers to functions!

```
#include<stdio.h>
void hello_function(int times);

int main(){
  void (*func_ptr)(int);
  func_ptr=hello_function;
  func_ptr(3);
  return 0;
}

void hello_function(int times){
  for(int i=0;i<times;i++) {
    printf("Hello, World!\n");
  }
}
```

We want function pointers to pass a function to a function

10 Using qsort()

- `stdlib.h` contains an implementation of the quicksort algorithm. The function declaration is:

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

- `void *base` is a pointer to the array
- `size_t nmemb` is the number of elements in the array
- `size_t size` is the size of each element
- `int (*compar)(const void *, const void *)` is a function pointer composed of two arguments and returns '0 when the arguments have the same value, <0 when `arg1` comes before `arg2`, and >0 when `arg1` comes after `arg2`.

```
#include <stdio.h>
#include <stdlib.h>
int compare (const void *, const void *);
int main() {
  int arr[] = {52, 14, 50, 48, 13};
  int num, width, i;
  num = sizeof(arr)/sizeof(arr[0]);
  width = sizeof(arr[0]);
  qsort(arr, num, width, compare);
  for (i = 0; i < 5; i++)
    printf("%d ", arr[i]);
  printf("\n");
  return 0;
}

int compare (const void *arg1, const void *arg2) {
  return *(int *)arg1 - *(int *)arg2;
}
```

11 Implement calloc() and realloc().

- Write functions `calloc2()` and `realloc2()`, that use `malloc()` and `free()` to implement the functionality of `calloc()` and `realloc()`, respectively.
- Remember that `calloc()` sets the allocated memory to zero (for this exercise, you may ignore testing for integer overflows when multiplying the arguments of `calloc()` together).
- When implementing the copying part of `realloc()`, recall that `char` is 1 byte; the C standard states that you may use `char *` pointers to access individual bytes of memory.