2803ICT

Advanced C

Outline

- Advanced C
  - Function pointers
  - (complex) Declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable argument lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

Outline

- Advanced C
  - Function Pointers
  - (complex) Declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable argument lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

Function Pointer

```
int x;

int* pointer;

int function();

int (*pFunc)();
```

x is an integer

pointer is a pointer to integer

function is the name of a function that takes no parameters and returns an int value

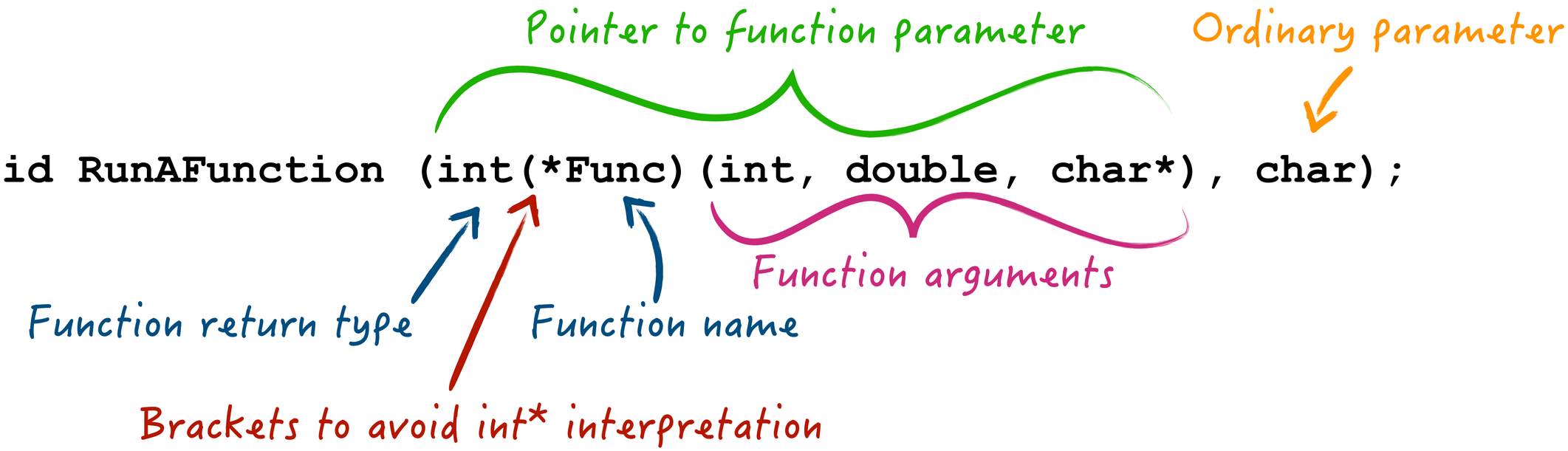
pFunc is pointer to function that takes no parameters and returns an int

```
int (*pFunc)();          //- ptr to function
pFunc = &function;       //- initialise its value
int res = (*pFunc)();    //- call  the function
int res = pFunc();       //- alternative call
```

Function as argument - example

We want a function to accept one of a series of functions of the type:

```
int DoSomething(int, double, char *);
int DoSomethingElse(int, double, char *);
int DoItAgainWithFeeling(int, double, char*);
```



Function as argument - example

```
void RunAFunction (int(*Func)(int, double, char*), char) {
     //…

     Func(6,2.5,”aaa”);

     //…

     }

     RunAFunction(DoSomething, ‘a’);
```

Function as argument - example

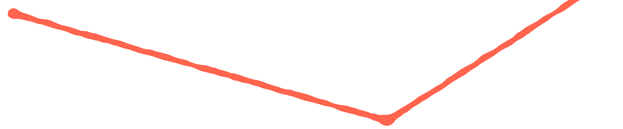
```
void RunAFunction (int(*Func)(int, double, char*), char);
```

Examples of calls to RunAFunction()

```
RunAFunction(DoSomething, 'a');
RunAFunction(DoSomethingElse, 'b');
RunAFunction(DoItAgainWithFeeling, 'c');
```

Example of use within RunAFunction():

```
Func(Num, Dec, Str);          // or
(*Func)(Num, Dec, Str);
```



Num, Dec, Str are local variables of RunAFunction()

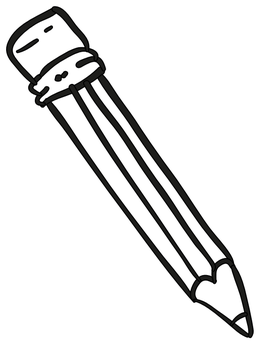
Function as argument - example 2

```
long step(int n)
{
  return n * n;
}
void work(int n, long(*func) (int) )
{
  int x, ans=0, y = 5;
  for(x=0; x<n; x++) ans += (*func)(x);
}
void main ()
{
  work(2, step);
}
```

Using `typedefs`

The main goal - make the code readable

```
typedef void func(int);      //- a function
typedef void (*pfunc)(int); //- a ptr to func
```



Declare a function named **`signal`**, that:

- takes an *integer* and a *pointer to a function* that takes int and return void
- returns a *pointer to a function* of the same type

Using `typedefs`

The main goal - make the code readable

```
typedef void func(int);      //- a function
typedef void (*pfunc)(int); //- a ptr to func
```

Declarations of the function **`signal`**:

```
void (*signal(int, void(*)(int)))(int);
func *signal( int, func* );
pfunc signal( int, pfunc );
```

Function pointer example - QSORT

To use qsort() you need to include **<stdlib.h>** and **<search.h>**

```
void qsort(void *base, size_t num, size_t size,
   int (*compare)(const void *e1, const void *e2));
```

Function pointer example - QSORT

To use qsort() you need to include **<stdlib.h>** and **<search.h>**

```
void qsort(void *base, size_t num, size_t size,
   int (*compare)(const void *e1, const void *e2));
```

Start of array to be sorted

#of elements

Element size in bytes

Function pointer example - QSORT

To use qsort() you need to include **<stdlib.h>** and **<search.h>**

```
void qsort(void *base, size_t num, size_t size,
   int (*compare)(const void *e1, const void *e2));
```

User supplied comparison function

Pointer to the key to search

Pointer to the element to be compared with the key

Function pointer example - qsort

To use qsort() you need to include **<stdlib.h>**

```
void qsort(void *base, size_t num, size_t size,
   int (*compare)(const void *e1, const void *e2));
```

```
compare( (void *) elem1, (void *) elem2 );
```

compare() compares two array elements, and returns a value specifying their relationship. The value must be:

0	elem1 = elem2
<0	elem1 < elem2
>0	elem1 > elem2

Function pointer example - QSORT

Griffith UNIVERSITY

Example: read the command-line params and sort them

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>


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

Function pointer example - QSORT

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int compare(const void *arg1, const void *arg2 )
{
      return strcmp(*(char**)arg1,*(char**)arg2);
}
void main( int argc, char **argv )
  {
        size_t big = sizeof(char*);
        argv++;          //- skip argv[0]
        argc--;
        qsort((void*)argv, (size_t)argc, big, compare);
        //--- print out sorted list --
        for(int i = 0; i < argc; ++i)
            printf("%s ", argv[i] );
        printf( "\n" );
  }
```

Output:

```
%qsort every good boy deserves favor
boy deserves every favor good
```

Function pointer example 2 - BSEARCH

```
void *bsearch(const void *key, const void *base,
      size_t num, size_t size,
      int (*compare)(const void *e1, const void *));
```

Pointer to object to search for

Example: find the word “cat” in the command-line params

Output:

```
%bsearch dog pig horse cat human rat cow goat
bsearch cat cow dog goat horse human pig rat
cat found
```

Function pointer example 2 - BSEARCH

```
int compare( char **arg1, char **arg2 );
void main( int argc, char **argv )
{
  char **result;
  char *key = "cat";
  size_t big = sizeof(char*);

  qsort( (void *)argv, (size_t)argc, big, compare); //sort
  for(int i = 0; i < argc; ++i )
     printf("%s ", argv[i] );

  result = (char**)bsearch((char*) &key,              //search
    (char*)argv, argc, big, compare);

  if (result) printf("\n%s found ", *result);
  else printf( "\nCat not found!\n" );
}
```

Outline

- Advanced C
  - Function Pointers
  - (complex) Declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable argument lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

Abstract declarators

- These are declarators **without** an **identifier**,
- Used in typedefs and formal parameter lists

```
int *
int *[5]
int (*)[5]
int *()
int (*)()
```

An array of five pointers to int

The type name for a ptr to type int

A ptr to a function returning an int

A function returning a pointer to int

A ptr to an array of five ints

Abstract declarators

- These are declarators **without** an **identifier**,
- Used in typedefs and formal parameter lists

```
int *           //– The type name for a ptr to type int
int *[5]        //– An array of five pointers to int
int (*)[5]      //– A ptr to an array of five ints
int *()         //– A function returning a pointer to int
int (*)()       //–A ptr to a function returning an int
```

Interpreting Declarations

- A simple way to **interpret complex declarators** is to read them "**from the inside out**," using the following four steps:

Start with the identifier and look directly to the right for brackets or parentheses (if any).

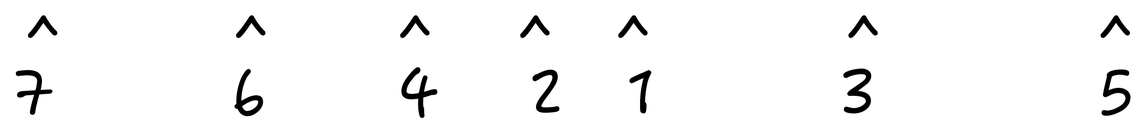
Interpret these brackets or parentheses, then look to the left for asterisks.

If you encounter a right parenthesis at any stage, go back and apply rules 1 and 2 to everything within the parentheses.

Apply the type specifier.

Example Declaration – 1

```
char *( *(*var)() )[10];
```



The steps to interpret this are numbered in order as follows:

1. The identifier **`var`** is declared as
2. a pointer to
3. a function returning
4. a pointer to
5. an array of 10 elements, which are
6. pointers to
7. **`char`** values.

Example Declaration – 2

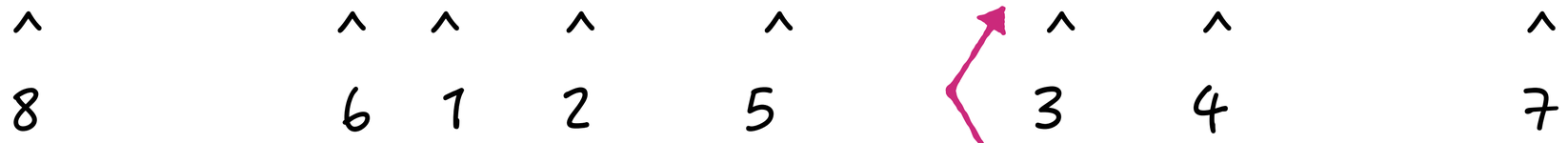
```
int *(*const *name[5][7])(void );
```

```
^       ^       ^   ^       ^       ^
6       4       3   1       2       5
```

1. **`name`** is declared as
2. an (2D) array of
3. pointers to a
4. **`const`** pointer. This **`const`** pointer points to
5. a function that takes no parameters and
6. returns a pointer to an **`integer`**.

Example Declaration – 3

```
double (*var( double(*)[3] ))[5];
```



1. **`var`** is declared as
2. a function that takes
3. a pointer to a
4. array of three
5. **`double`** values
6. and returns a pointer to
7. an array of five
8. **`double`** values.

The parentheses around the asterisk in the argument type are required; without them, the argument type would be an array of three pointers to **`double`** values.

Outline

- Advanced C
  - Function pointers
  - (complex) Declarations
  - volatile and extern
  - Command line arguments
  - Variable argument lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

`Volatile` Type Qualifier

- This is used to tell the compiler that the value of the **variable may be modified unexpectedly by the OS or hardware** outside of the running program.
- When a variable is declared as **`volatile`**, the **compiler reloads the value from memory each time** it is accessed by the program.

`Volatile` Type Qualifier

- Used for objects in memory that might be shared by:
  - multiple concurrent processes
  - interrupt service routines
  - memory-mapped I/O control hardware

`Volatile` Type Qualifier

- An item can be both **`const`** and **`volatile`**,

```
int const* volatile w = &object;
```

- Here the value of the memory pointed to by **`w`** cannot be modified by the program itself but it might be modified by another process

`extern` Type Qualifier

- This **permits one part** of your program to **access** a global variable or function **defined in some other part** of your program.
- Essentially tells the compiler to **look in another file** for the definition of a function or variable
- Example:

```
extern int array[2];
```

Outline

- Advanced C
  - Function pointers
  - (complex) Declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable argument lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

Command Line Arguments

- You can provide **input** to a C or C++ program by **passing arguments** on the **command line**
- The form of the main program arguments is:

```
void main (int argc,  char **argv,  char **envp)
```

number of arguments (includes the program name)

ragged array of arguments as strings

environment variables

- **`argc`**, **`argv`** and **`envp`** are parameter names by convention
- **`**argv`** and **`**envp`** are ragged arrays
- These are **all optional**

Command Line Arguments

```
>myprog hello 12 4.5
```

argc = 4

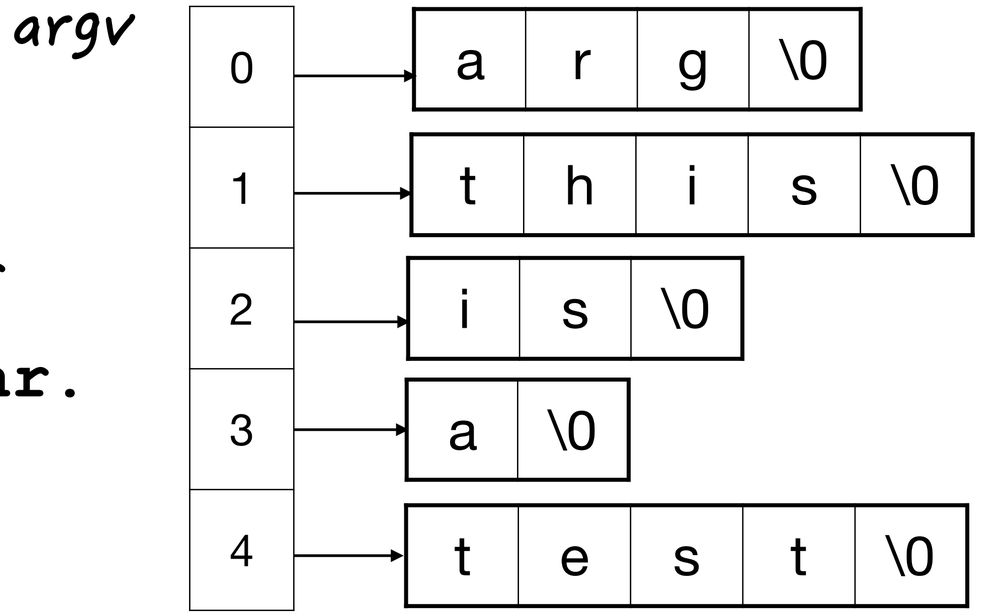
argv = array of 4 pointers to the strings: “myprog”, “hello”, “12”, “4.5”

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    for (i=0; i<argc; i++) {
       printf(“argv[%d] = %s\n", i, argv[i]);
    }
  return 0;
}
```

Command Line Arguments

- **`argv`** is an array of null terminated strings, eg.



- **`argv`** can be declared as:

`char *argv[]`: an array of pointers to **`char`**

`char **argv` : a pointer to pointers to **`char`**.

- The first string (**`argv[0]`**) is the **program name**, and each following string is an argument passed to the program. The last pointer (**`argv[argc]`**) is **`NULL`**.

Command Line Arguments

- Example if “**`$echo hello world`**” then:
  - argc = 3
  - argv[0] – the name of the program “echo”
  - argv[1] – the string “hello”
  - argv[2] – the string “world”

- Example: echo arguments

```
main(int argc, char *argv)
 {
   while ( --argc > 0 )
      printf((argc>1)?"%s": "%s\n", *++argv);
 }
```

Environment Variables

- The environment is a **symbolic variable** that represents an element of the user's operating system environment such as a path, a directory name or a configuration string.
- For example the PATH environment variable
- The environment block passed to main via **`envp`** is a "frozen" copy of the current environment.
- **`envp`** is a pointer to an array of environment strings, that can be declared as either :
- `char * envp[];` // an array of pointers to char
- `char **envp;` // a pointer to pointers to char
- The end of the array is indicated by a **`NULL`** pointer.

Environment Variables

Griffith UNIVERSITY

```
#include <stdio.h>

main(int argc, char* argv[],  char* envp[])
{
  int a;
  printf("The command name (argv[0]) is %s\n", argv[0]);
  printf("There are %d arguments:\n", argc-1);
  for (a=1; a<argc; a++)
     printf("\targument %2d:\t%s\n", a, argv[a]);

  printf("The environment is as follows:\n");
  a = 0;
  while (envp[a] != NULL)
     printf("\t%s\n", envp[a++]);
}
```

Outline

- Advanced C
  - Function pointers
  - (complex) Declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable Argument Lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

Variable Length Argument Lists

- In C and C++ functions can have variable length argument lists, that is, they can take a variable number arguments eg. printf and scanf are already of this form
- Functions that require variable lists are declared using the ellipsis **(...)** in the argument list.
- Requires the **<stdarg.h>** header file
- To access arguments passed to functions using this method, you must use the types **`va_list`**, **`va_start`**, **`va_arg`**, and **`va_end`** macros
- Variable argument lists are implemented whenever the last argument in a function is the ellipsis (…)
- A function that takes a variable number of arguments requires at least one “placeholder” argument, even if it is not used. If this place-holder argument is not supplied, you can not access the remaining arguments

Variable Length Argument Lists

```
va_start (ap, v)
```

- Is used to initialise the **a**rgument **p**ointer **ap** to point to a list of the arguments given after **v** the last argument actually declared in function header - **v** cannot be an array.

```
va_arg (ap, type)
```

- Is used to step through argument list - variables of any type can be in the list

```
va_end (ap)
```

- Used to clean up memory when done

Outline

- Advanced C
  - Function pointers
  - (complex) declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable argument lists
  - The Preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

The Preprocessor

Griffith UNIVERSITY

- The preprocessor is a **text processor** built into the compiler that manipulates the text of a C/C++ source file before the file is actually parsed.
- It defines directives that are typically used to make source programs **easy to change** and **easy to compile** in different execution environments.
- For example directives may be used to:
  - **Replace/substitute text** in the file,
  - **insert** the **contents** of other files into the source file,
- suppress compilation of part of the file by **removing sections** of text.
- Preprocessor statements do not support escape sequences.

Preprocessor Directives

- The preprocessor recognises the following directives

```
#define #undef      //- includes/ ignores macros
#ifdef  #ifndef     //- conditional compilation
#if     #endif      //- conditional compilation
#else   #elif       //- conditional compilation
#include             //- includes a file
#line               //- get or set the current line number
#error              //- produce compile-time error messages
#pragma            //- give special instructions to compiler
```

Preprocessor Directives

- Directives can appear anywhere in a source file.
- The hash sign (#) must be the first nonwhite-space character on the line before the directive
- Some directives include arguments or values.
- Directives are terminated by an EOL character
- Multiline preprocessor directives can created by placing a backslash (\) at the end of each line, this means the directive continues on the next line

```
#define STORY  Once upon a time in a far \
               away land there was a     \
               lonely cane toad …
```

Using `#define` / `#undef`

- **`#define`** is used to create **a macro** that the preprocessor will replace in the source file, eg

```
#define PI 3.1417
int area = r * PI * PI;
```

- **`#undef`** is used to tell the preprocessor to ignore a previously defined macro

```
#undef PI
```

Using `#define` / `#undef`

- Macros can be defined to accept one or more arguments

```
#define INCH2CM(a) ((a) * 2.54)
```

- Macros are **very fast** because they do not have any function call overheads – line **inline** in C++
- Macros don’t perform any type checking

Using #define / #undef

- It is very important to **enclose the macro arguments in brackets**, so that they are properly interpreted

```
#define SQ(x)     x * x
```

```
int a = SQ(k + 7) / 12
```

```
int a = k + 7 * k + 7 / 12
```

The result setting **k=1** becomes 8

```
#define SQ(x)     ((x) * (x))
```

```
int a = SQ(k + 7) / 12
```

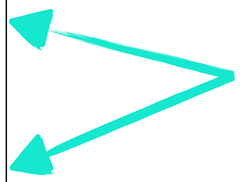
```
int a = ((k + 7) * (k + 7)) / 12
```

The result setting **k=1** becomes 6

Conditional Compilation

- **`#ifdef / #endif / #ifdef / #ifndef / #if / #else`** are used to control **what parts of a program get compiled**
- These are useful for writing programs that need to be compiled differently for different platforms

```
#define WIN32
#ifdef WIN32
   printf("win32 specific code");
#else
   printf("unix specific code");
#endif
```



The compiler will only compile one of the two code blocks depending on if `WIN32` has been defined

Example X-platform programming Loadable libraries:

```
#ifdef WIN32
 #include <direct.h>
 #include <windows.h>
#else
 #include <sys/types.h>
 #include <dlfcn.h>
#endif

#ifdef WIN32
 string nameOfLibToLoad("C:\opt\lib\libctest.dll");
 HINSTANCE lib_handle =
 LoadLibrary(TEXT(nameOfLibToLoad.c_str()));
#else
 string nameOfLibToLoad("/opt/lib/libctest.so");
 void * lib_handle = dlopen(nameOfLibToLoad.c_str(), RTLD_LAZY);
#endif
 if (!lib_handle) exit(-1);
```

Continued next slide…

Continued from previous slide

```
#ifdef WIN32
 void* fn_handle = (func_t*) GetProcAddress(lib_handle,
 "superfunctionx");
 if (!fn_handle) exit(-1);
#else
 dlerror(); // reset errors
  void* fn_handle= (func_t*) dlsym(lib_handle, "superfunctionx");
 const char* dlsym_error = dlerror();
     if (dlsym_error) exit(-1);
#endif
 /** ... **/

#ifdef WIN32
    FreeLibrary(lib_handle);
#else
    dlclose(lib_handle);
#endif
```

Example X-platform programming Loadable libraries:

Predefined Macros

- ANSI C supports a number of predefined macros
  - **__DATE__** The compilation date of the source file.
  - **__FILE__** The name of the current source file.
  - **__LINE__** The line number in the source file.
  - **__STDC__** Indicates conformance with ANSI C
  - **__TIME__** The compilation time of the source file.
  - **__TIMESTAMP__** date & time of last file modification
- Examples:

```
printf("This program compiled on %s", __DATE__);
printf("Executed line %d in file %s",  __LINE__, __FILE__);
```

Stringizing (#) Operator

- Causes its argument to be **converted into a string**

```
#define STR(x) #x
```

```
printf("%s", STR(1+2));
printf("%s", STR("1+2"));
```

-> printf ("%s", "1+2");

Output

```
1+2
1+2
```

- We can also use it to print out the definition of macros

```
#define ISZERO(x) (x ? FALSE : TRUE)
#define STR(x) #x
#define DUMP(x) STR(x)

printf("%s", DUMP(ISZERO(2)));
```

Output

```
(2 ? FALSE : TRUE)
```

Token-Pasting (##) Operator

- **Concatenates** its arguments together to make new tokens
- Allows the creation of C++ template like code in C

```
#define JOIN(a,b)        a ## b
```

```
JOIN(int_, List(););  ⇒ int_List();
JOIN(dbl_, List(););  ⇒ dbl_List();
```

```
#define PASTE(n)    printf("token" #n "= %d",  token##n )
int token9 = 9;
```

```
PASTE(9);      ⇒ printf( "token" "9" " = %d", token9 )
               ⇒ printf( "token9 = %d", token9 )
               ⇒ token9 = 9
```

Parsing Text - `strtok`

- Often you need to take some text input and **split it up** into different fields
- There are many ways to do this: one is

```
int n = 0;
char *tokens[100];
token[n] = strtok(inputString, “, ");
while (token[n++])
{
   tokens[n] = strtok( NULL, “," );
}
```

Parsing Text - `sscanf`

- Another way of parsing is similar to using **`printf`**
- **`sscanf`** parses input strings
- **`scanf`** reads and parses from standard input
- Say you want to parse "**Test 1 1.5 a –3 end**"
- You could do it as follows:

```
char str1[10], str2[5], str3[5];
int    num1, num3;
float num2;
int res = sscanf(string, "%s %d %.1f %c %d %s", str1, &num1,
                                &num2, str2[0], &num3, str3);
if (res < 6) printf("Error");
```

Outline

- Advanced C
  - Function pointers
  - (complex) Declarations
  - **`volatile`** and **`extern`**
  - Command line arguments
  - Variable argument lists
  - The preprocessor
- System Info and Time
  - Generic
  - Unix
  - Windows

System Time (Generic)

- There are different ways to get the time

  #include <time>

- The Kernel calendar time (secs since midnight 1 Jan 1970)

  time_t time(time_t * ptr); **//- secs**

- Convert time_t to a human readable value using

  char*ctime(const time_t * ptr);

  - returns: “Tue Feb 10 18:27:38 2011”

- Get running program time (in CLOCKS_PER_SEC * sec) using **clock_t clock();**

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

System Time (Generic)

- You can get the time broken down into days/years etc

struct tm* gmtime (time_t * ptr)//- greenwich mean time

struct tm* localtime (time_t * ptr) //- local time zone

```
struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    …
}
```

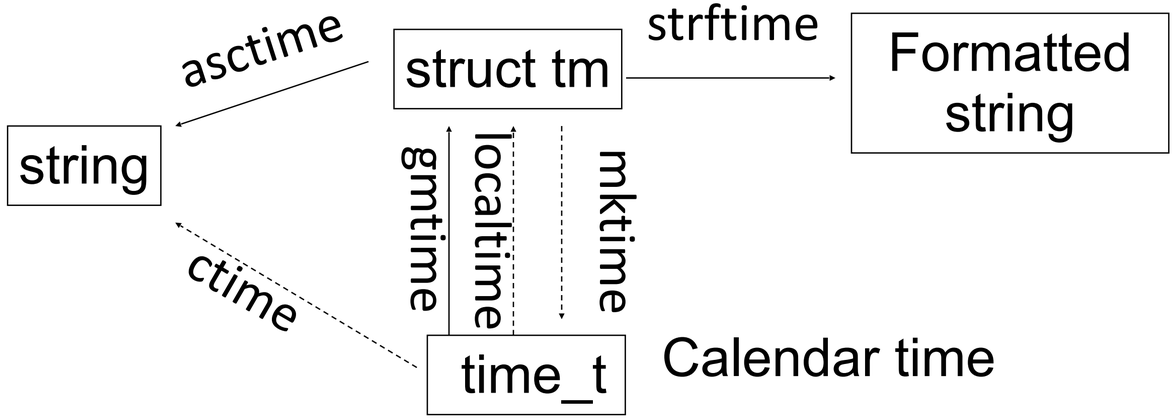
© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Time Conversions (Generic)

- **You can convert between these time formats human readable time using**

```
time_t mktime(struct tm*);                    //- convert tm to time_t
char* asctime(const struct tm*); //- like ctime
size_t strftime(char* buf, size_t maxsize, char* format, struct tm*);
```

  - **The format string uses special characters to define how to format the time that will be stored in buf**



© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Unix Microsecond Time Routine

- A Hires Kernel calendar time (in usecs)

```
int gettimeofday(struct timeval*, NULL)

struct timeval
{
 time_t tv_sec;     //- seconds
 long   tv_usec;    //- microseconds
};
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Unix Nanosecond Timer

```
int clock_getres(clockid_t clk_id, struct timespec *res);
int clock_gettime(clockid_t clk_id, struct timespec *tp);
int clock_settime(clockid_t clk_id, const struct timespec *tp);
struct timespec {
    time_t   tv_sec;        //- seconds
    long     tv_nsec;       //- nano seconds
};
```

- **Example**

```
#include <sys/time.h>
int main()
{
    timespec ts;
    // clock_gettime(CLOCK_MONOTONIC, &ts); // Works on FreeBSD
    clock_gettime(CLOCK_REALTIME, &ts); // Works on Linux
}
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Unix System Info

- System info

```
int uname(struct utsname* name)
stuct utsname
{
   char  sysname[];        //- Name of this operating system.
   char  nodename[];       //- Name of this node/host
   char  release[];        //- Current release level of this OS.
   char  version[];        //- Current version of this release.
   char  machine[];        //- Name of the hardware type
}
int gethostname(char* name, int length);
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Win32 System Info

void GetSystemInfo(LPSYSTEM_INFO lpSystemInfo);

- lpSystemInfo
  - **A pointer to a system_info structure that receives the information.**

```
typedef struct _SYSTEM_INFO {
    union {
            DWORD dwOemId;
            struct {
                    WORD wProcessorArchitecture;
                    WORD wReserved;
            };
    };
    DWORD     dwPageSize;
    LPVOID    lpMinimumApplicationAddress;
    LPVOID    lpMaximumApplicationAddress;
    DWORD_PTR dwActiveProcessorMask;
    DWORD     dwNumberOfProcessors;
    DWORD     dwProcessorType;
    DWORD     dwAllocationGranularity;
    WORD      wProcessorLevel;
    WORD      wProcessorRevision; } SYSTEM_INFO;
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Win32 Version Info

BOOL GetVersionEx(LPOSVERSIONINFO lpVersionInfo);

- lpVersionInfo
  - **An** OSVERSIONINFO **or** OSVERSIONINFOEX **structure that receives the operating system information.**
  - **Before calling the** GetVersionEx **function, set the** dwOSVersionInfoSize **member of this structure as appropriate**

```
typedef struct _OSVERSIONINFO
{
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
} OSVERSIONINFO, *LPOSVERSIONINFO;
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Win32 Example

```
int main()
{
    OSVERSIONINFO osvi;
    SYSTEM_INFO si;
    ZeroMemory(&si, sizeof(SYSTEM_INFO));
    ZeroMemory(&osvi, sizeof(OSVERSIONINFOEX));

    osvi.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
    if ( !(GetVersionEx ((OSVERSIONINFO *)  &osvi)) )
        return FALSE;

    GetSystemInfo(&si);
    return 0;
}
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

System Information Functions

- **GetComputerName** Returns the NetBIOS name of the local computer
- **GetComputerNameEx** Returns the NetBIOS or DNS name of the computer.
- **GetComputerObjectName** Returns the computer name in a specified format
- **GetCurrentHwProfile** Retrieves the current hardware profile.
- **GetKeyboardType** Retrieves information about the current keyboard.
- **GetNativeSystemInfo** systeminfo for applications running under WOW64.
- **GetProductInfo** Retrieves the product type for the operating system.
- **GetSysColor** Retrieves the current color of a display element.
- **GetSystemDirectory** Retrieves the path of the system directory.
- **GetSystemInfo** Retrieves information about the current system.
- **GetSystemMetrics** Retrieves the specified system metric.
- **GetSystemWindowsDirectory** Retrieves the path of the Windows directory.
- **GetUserName** Returns the user name of the current thread.
- **GetVersion** Returns the version number of the operating system.
- **GetWindowsDirectory** Retrieves the path of the Windows directory.
- **IsProcessorFeaturePresent** checks if a processor feature is supported.
- ….MANY OTHERS

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Win32 Timer Functions

- DWORD GetTickCount(); // milisecs secs since reboot

- Calendar Time can be retrieved as well

```
void GetSystemTime(SYSTEMTIME* lpSystemTime);
BOOL SetSystemTime(SYSTEMTIME* lpSystemTime);
void GetLocalTime(SYSTEMTIME* lpSystemTime);
typedef struct _SYSTEMTIME {
        WORD wYear;
        WORD wMonth;
        WORD wDayOfWeek;
        WORD wDay;
        WORD wHour;
        WORD wMinute;
        WORD wSecond;
        WORD wMilliseconds;
} SYSTEMTIME,  *PSYSTEMTIME;
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Windows Hires Timer

```
class CHiResTimer
{
protected:
        __int64 m_nStart;
        __int64 m_nStop;
        __int64 m_nFreq; //- ticks per second
        void        Stop();
        bool        m_bSupported;
        __int64 Diff()       {return m_nStop - m_nStart;}
public:
        CHiResTimer(bool bAutoStart = true);
        virtual ~CHiResTimer() {};

        bool        IsSupported() const  { return m_bSupported; };
        void        Reset();
        double    Elapsed()      { Stop(); return Diff() / double(m_nFreq); }
        __int64   sElapsed()     { Stop(); return Diff() / m_nFreq; }
        __int64   msElapsed()  { Stop(); return Diff() / (m_nFreq / 1000); }
        __int64   usElapsed();
};
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Windows Hires Timer

```
CHiResTimer::CHiResTimer(bool bAutoStart)
{
    m_bSupported = QueryPerformanceFrequency(
                                (LARGE_INTEGER *)&m_nFreq) != 0;
     m_bSupported &= (m_nFreq != 0);
     if (bAutoStart) Reset();
}
__int64   CHiResTimer::usElapsed()
{
    Stop();
    const unsigned long div = 0x10c6f7a0b5edUL;        //= 2^64 / 1e6
    __int64 d = (m_nStop - m_nStart);
    if (d < div)       return (d*1000000UL)/m_nFreq;     // pick best one
    else               return (d/m_nFreq)*1000000UL;
}
void        CHiResTimer:: Stop()
{   QueryPerformanceCounter((LARGE_INTEGER *)&m_nStop);  }
void        CHiResTimer:: Reset()
{   QueryPerformanceCounter((LARGE_INTEGER *)&m_nStart); }
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Error Handling

UNIX

- System calls return –1 on error
- When there is an error the OS sets a global error value based on the errors listed in /usr/include/errno.h
- The program can print a description of an error to the console using

  perror(str);

Windows

- Win32 API returns zero (FALSE) if a function fails.
- To get extended error information, use the **FormatMessage()** function.

© Ruben Gonzalez 1999-2010, © 2018 René Hexel


Software Development Gudie

Considerations

- Dealing with uncertainty and complexity
  - Uncertainty - Requirements engineering
  - Complexity - Application of good design methods
- Software maintenance - characteristics of maintainable software
  - Properly structured – little coupling, high cohesion
  - Appropriately documented (file, function, line level)
  - Coding standards (style guide, variable naming etc)
- Systems level considerations,
  - interaction of software with its environment.
  - Performance, hardware, users, other software etc

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Software Development Phases

- There are 4 basic processes involved in software development, analysis, and design, coding and testing.
- 1. Analysis
  - Primary aim – to generate requirements specification /
  - Information Structures (Objects)
  - Functional block diagram
- 2. Design
  - software architecture: Class Diagram / Structure Chart / etc
  - Data Structures (eg linked lists, queues, trees, arrays etc)
  - Algorithm selection
  - File formats / screen layout / user interface
- 3. Coding
  - Choice of language
  - Coding Style / Methodology
  - Software documentation
- 4. Testing
  - Boundary value testing
  - Functional testing
  - Structural testing
  - Acceptance Testing

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Requirements Engineering

- Categories
  - Functional
  - Nonfunctional
  - Quantifiable
  - System / software level
- Analysis
  - Classification
  - Clarification
  - Negotiation
  - Modeling
- Gathering
  - Sources
  - Methods
- Specification
  - Evaluation and use of System requirements
  - Evaluation and use of Software requirements
- Req. Validation
  - Reviews
  - Prototyping
  - Acceptance testing

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Software Design Issues

- **Two phase Design Process**
  - **Architectural design – high level / block diag**
  - **Detailed design – algorithms, data structs etc**

- **Design Issues:**
  - **Event Handling**
  - **Fault tolerance**
  - **Security,**
  - **Performance (speed / memory use)**
  - **Data persistence,**
  - **Hardware Constraints**
  - **usability**

- **Relationship between requirements, design and validation**

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Software Design Methods

- System Modeling
  - Information (data flow)
  - Behavioral (event driven)
  - Structure
  - Abstraction
  - Coupling / cohesion
  - Encapsulation / Information hiding
- Design Patterns
  - creational
  - structural
  - behavioural
- Design Paradigms
  - Function oriented (top-down structure decomposition)
  - Object oriented design
  - Data centric design
- Software Architecture
  - Distributed
    - Client / Server, N-layered
  - Concurrent
    - Multiprocess,
    - multithreaded
    - parallel,
  - Pipes and filters

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Module/Function Cohesion

- This is a measure of how well a module hangs together, the interrelation of the code and the data references within the module.
- Cohesion is measured on a scale starting at

1. Coincidental – coincidence, nonsense
2. Logical – similar operations eg all I/O together
3. Temporal cohesion – together because of timing
4. Procedural cohesion – together because of control, ordered flow of tasks/steps.
5. Communication – shares data within module
6. Sequential cohesion – both procedural+communication
7. Functional – operations only carry out a single task
8. Information based – a collection of functionally cohesive modules sharing a hidden data structure.

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Module/Function Coupling

- This is a measure of the independence of two (or mode) modules with respect to each other.
- The higher the coupling, the less independence and the harder it is to think of them as modules.
- The lower the coupling, the more independent modules are and the easier it is to separately reason about, design, modify & test the modules
- Coupling is measured on a scale starting at

1. Content – modules dive into each other – nonsense
2. Common coupling – access the same global data
3. Control coupling – pass explicit (ie not implied by the data) control information between modules
4. Stamp coupling – modules refer to the same data structures although not to the same field
5. Data coupling – communicate by parameters only

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Software Validation

- Objectives
  - Acceptance
  - Installation
  - Alpha / beta
  - Regression
  - Performance
  - Security
  - Stress
  - Configuration
  - Usability / HCI
- Levels
  - Unit
  - Integration
  - System
- Methods
  - Input based
    - Boundary value
    - Random
    - Equivalence
  - Code based
    - Control flow
    - Data flow
    - Fault based
  - Others
    - Usage based
    - Model based
    - Black box
    - White box

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Documentation

- Requirements Specification
  - System level – non functional eg performance
  - Software Level – functional, user
  - Acceptance Tests (to be performed)
- Design Specifications
  - Architectural Design
    - Logical Block diagram
    - State machines
    - Control flow
    - Data flow
  - Detailed Design
    - Structure charts
    - UML class diagram
    - Pseudo code
- Code Documentation (high level: doxygen)
- Test results (acceptance tests, other tests)

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Bad C Program

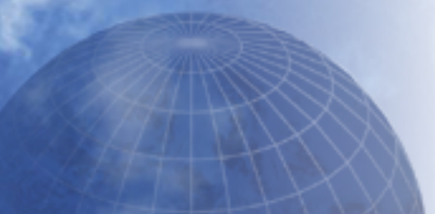
```
struct NODE{char *data; NODE *next;};
typedef struct NODE Node;
main()
{
   char string[32];
   scanf("%s", string );          // get & add 1st string to list
   Node *Head = malloc(sizeof(Node));
   Head->next = NULL;             // ←NOTE !
   Head->data = malloc( strlen(str) );
   strcpy(Head->data, string);

   scanf("%s", string );          // get & add 2nd string to list
   Head->next = malloc(sizeof(Node) );
   Head->next->data = malloc( strlen(str) );
   strcpy(Head->next->data,string);
   Head->next->next = NULL;       // ←NOTE !
}
```

© Ruben Gonzalez 1999-2010, © 2018 René Hexel

Better C Program

```
struct NODE{char *data; NODE* next;};
typedef struct NODE Node;
void AddToList(Node* ptr, char* str)
{
   Node* tmp = malloc(sizeof(Node) );
   tmp->next = ptr;
   tmp->data = malloc( strlen(str) );
   strcpy(tmp->data,str);
}
main()
{
   Node *Head;
   char string[32];
   scanf("%s", string );
   AddToList(Head, string);
   scanf("%s", string );
   AddToList(Head, string);
}
```



© Ruben Gonzalez 1999-2010, © 2018 René Hexel

C+ Program (not quite C++)

```
struct NODE {char *data, NODE *next;};
typedef struct NODE Node;
struct LIST {Node *head, void (*Add)(LIST*, char*);};
typedef struct LIST List;
Void Add(List* this, char* str)
{
   Node* ptr = this->head;
   head = malloc( sizeof(Node) );
   head->next = ptr;
   head->data = malloc( strlen(str) );
   strcpy(head->data, str);
}
main()
{
   List myList;
   char string[32];
   scanf("%s", string );
   myList.Add(&myList, string);
   scanf("%s", string );
   myList.Add (&myList, string);
}
```

The initialisation of LISTOBJ is left as an exercise to the reader ☺

© Ruben Gonzalez 1999-2010; © 2010 René Hexel

Questions?

© Ruben Gonzalez 1999-2010, © 2018 René Hexel