

# Lectures 5-6: Introduction to C

---

- Motivation:
  - C is both a high and a low-level language
  - Very useful for systems programming
  - Faster than Java
- This intro assumes knowledge of Java
  - Focus is on differences
  - Most of the syntax is the same
  - Most statements, expressions are the same



# Outline

---

- Major differences with Java
- A simple program; how to compile and run
- Data-types and variables
- The preprocessor
- Composite data structures
- Arrays and strings
- Pointers



# Major differences with Java

---

- C is not object oriented
  - C programs are collections of **functions**, like Java methods, but not class-based.
  - No inheritance, subtyping, dynamic dispatch in C
- C is not interpreted
  - A C program is **compiled** into an executable machine code program, which runs directly on the processor
  - Java programs are compiled into a **byte code**, which is read and executed by the Java interpreter, another program



# C is less “safe”

---

- Run-time errors are not ‘caught’ in C
  - The Java interpreter catches these errors before they are executed by the processor
  - C run-time errors happen for real and the program crashes
- The C compiler trusts the programmer!
  - Many mistakes go un-noticed, causing run-time errors



# Memory management is different

---

- Memory areas
  - *Heap*: dynamically allocated storage
  - *Stack*: for function/method local variables
  - *Static*: for data living program lifetime
- In Java
  - All objects on heap
  - Unusable objects on heap recycled automatically by garbage collection
- In C
  - Data structures in all 3 areas
  - Programs must explicitly free-up heap storage that is no longer needed



# C has pointers ...

---

- Pointers are special variables that reference (or point to) another variable
  - Similar to Java references
- We have already seen pointers in assembly:  
`lw $t1, 0($s2)`
  - `$s2` is a pointer
  - C pointers are the same thing! (more later)



# The hello world program

---

```
#include<stdio.h>
/* This is a (multi-line)
   comment */
int main(void)
{ // This is a comment too
  printf("Hello world!\n");
  return 0;
}
```

Linux/DICE shell commands

Compile: `gcc hello.c`      Run: `./a.out`



# Compilation units

---

- Programs are divided into *compilation units*
  - Provide degree of modularity
  - Each commonly has main file for source code
  - *Header* files characterise public interfaces of units
- Each compiled separately to relocatable object code
  - Allows creation of object-code libraries
- A *linker* assembles these into an *executable*, resolving references between units
- A *loader* sets up the executable program in memory and initialises data areas, prior to program being run





# Compilation units example

---

**A.h:**  
*func decls*  
*var decls*  
*type defs*

...

**A.c:**  
  
`#include "A.h"`

*func defs*  
*var defs*

...

**B.h:**  
*func decls*  
*var decls*  
*type defs*

...

**B.c:**  
`#include <stdio.h>`  
`#include "A.h"`  
`#include "B.h"`

*func defs*  
*var defs*

...



# Built-in data types

---

- The usual basic data types are there:

char	8 bits
short	16
int	16, 32, 64 (same as machine word size)
long	32, 64
float	32
double	64
- Bit sizes are machine dependent
  - Unlike Java where an int is always 32 bits
- Normally signed, unsigned available too
- No boolean type exists
  - for any number (int, char,...): 0 false, other true



# Categories of variables

---

- Global (*external*) variables (statically allocated)
  - Defined outside of functions
  - Have *lifetime* of program and *scope* to file end
  - **extern** declarations extend scope before definition and to other files
  - Declare **static** to hide from other files
- Local (*automatic*) variables (allocated on stack)
  - Defined inside a function (before the statements)
  - Not available outside function
  - Distinct storage for each function invocation
  - Declare **static** for same storage for all invocations



# The C pre-processor: `cpp`

---

- Includes – imports header files  
`#include <stdio.h>`  
`#include "A.h"`
- Text substitution, e.g. define constants  
`#define NAME value`
- Macros (inline functions)  
`#define MAX(X,Y) (X>Y ? X : Y)`
- Conditional compilation  
`#ifdef DEBUG`  
`Printf("Debugging message");`  
`#endif`



# Composite data structures - struct

---

- Structures are like objects, but their types have no methods, unlike classes:

```
struct point {  
    int x, y;  
} p1;  
struct point p2;
```

- Components accessed using “.” operator  
`p1.x = 2;`
- Passing structures between functions
  - In Java references always used
  - In C either all data copied or pointers used



# Composite data structures - union

---

- Unions – declared and used similarly to structures:

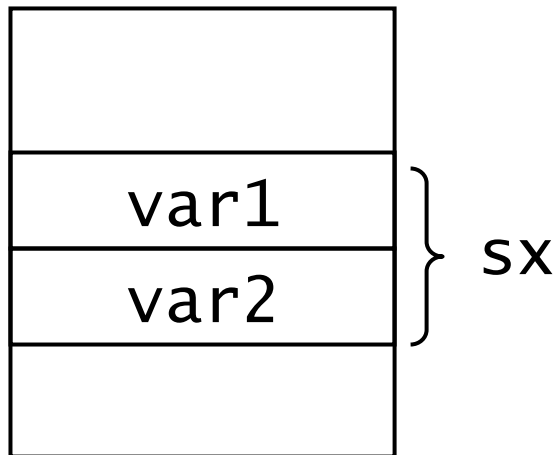
```
union geomObject {  
    struct circle;  
    struct rectangle;  
} g_obj;
```
- But all variables inside a union overlap in memory,
  - Space is reserved for the largest of them, not all
  - The same memory space can be interpreted in multiple ways



# In memory: structures v. unions

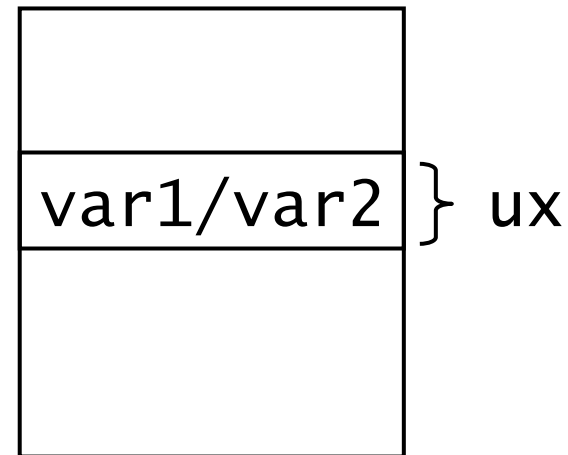
---

```
struct x {  
    int var1;  
    int var2;  
} sx;
```



sizeof(sx) → 8

```
union x {  
    int var1;  
    int var2;  
} ux;
```



sizeof(ux) → 4



# User-defined types

---

- Define names for new or built-in types

```
typedef <type> <name>;
```

- Example:

```
typedef unsigned char byte;
```

```
typedef struct {  
    struct point p;
```

```
    int rad;
```

```
} circle;
```

```
...
```

```
circle c1, c2;
```





# Arrays

---

- Syntax of C arrays similar to Java
- As in Java, C arrays have fixed size
- Example declarations of array:  

```
int n[] = {5, 8, 10}; // size fixed to 3  
circle c[4]; // array of structs
```
- C arrays have no knowledge of their length
  - No checking that indexes are within bounds
- In C is close relationship between arrays and pointers
  - Pointers commonly used to pass arrays between functions



# Strings

---

- C strings are simply arrays of type `char`
  - Encoded in 8bits using ASCII
- They end with `'\0'`, the **null** character
  - `char s[10]; // up to 9 characters long`
- String initialisation
  - `char s[10] = "string"; // '\0' implied`
  - `char s1[] = "another string";`
- Usual C rule for arrays apply:
  - Cannot store more chars than reserved at declaration
  - But bounds are not checked!



# Strings – common operations

---

- Assignment: `strcpy(s, "string");`
- Length: `strlen(s)`
- To get the 6<sup>th</sup> character: `s[5]`
  - First char at position 0, as in Java arrays
- Comparison, `strcmp(s1, s2)` returns:
  - 0 when equal
  - Negative number when lexicographically  $s1 < s2$
  - Positive when  $s1 > s2$
- Must `#include<string.h>` to call the functions
  - Type: `man string` to see what's available



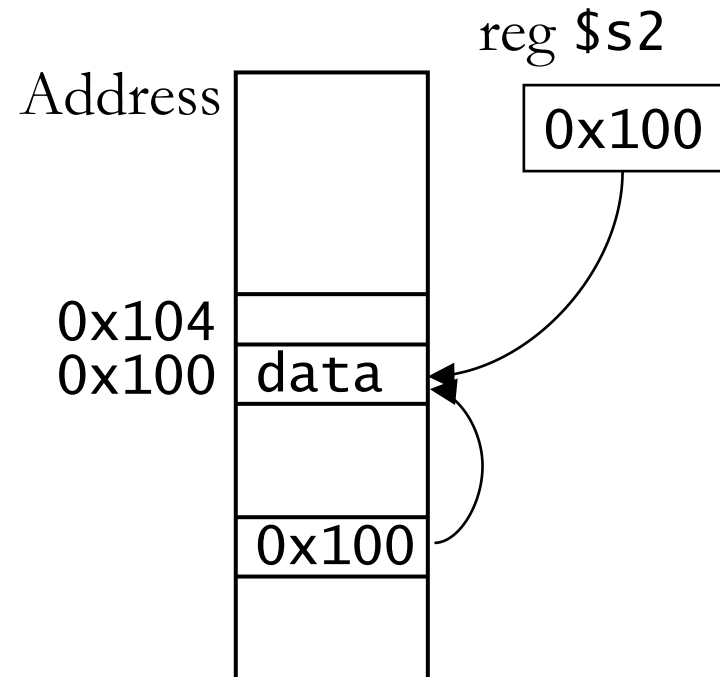
# Pointers

- We have seen pointers in assembly:

```
lw $t1,0($s2)
```

- \$s2 points to the location in memory where the “real” data is kept
- \$s2 is a register, but there’s nothing stopping us to have pointers stored in memory like “normal”

variables



# C pointers

---

- A C pointer is a variable that holds the address of a piece of data
- Declaration:  
`int *p; // p is a pointer to an int`
  - The compiler must know what data type the pointer points to
- Basic pointer usage:  
`p = &i; // p points to i now`  
`*p = 5; // *p is another name for i`
- `&` - address of, `*` dereference operator



# Pointers as function arguments

---

- In Java
  - an argument with primitive type is passed by value (function gets copy of value)
  - an argument with class type is passed by reference (function gets reference to value)
- In C
  - All arguments passed by value
  - To get effect of ‘pass by reference’, use an argument with a pointer type



# Example – the swap function

---

```
void swap_wrong(int a, int b) {  
    int t=a;  
    a=b; b=t;  
}
```

swap\_wrong swaps the local variables a, b which are unknown outside of the function

```
void swap(int *a, int *b) {  
    int t=*a;  
    *a=*b; *b=t;  
}
```

Function call: swap(&x, &y);



# Pointer arithmetic and arrays

---

C allows arithmetic on pointers:

```
int a[10];
```

```
int *p;
```

```
p = &a[0]; // p points to a[0]
```

$p+1$  points to  $a[1]$

– Note that  ~~$\&a[1] = \&a[0]+4$~~

– The compiler multiplies  $+1$  with the data type size

In general:  $p+i$  points to  $a[i]$ ,  $*(p+i)$  is  $a[i]$

Even  $*(a+i)$   $p[i]$  are allowed

– but cannot change what  $a$  points to. It's not a variable





# More pointer arithmetic

---

Common expressions:

`*p++` use value pointed by `p`, make `p` point to next element

`*++p` as above, but increment `p` first

`(*p)++` increment value pointed by `p`, `p` is unchanged

- Special value `NULL` used to show that a pointer is not pointing to anything
  - `NULL` is typically `0`, so statements like `if (!p)` are common
- Dereferencing a `NULL` pointer is a very common cause of C program crashes



# Example – pointer arithmetic

---

Return the length of a string:

```
int strlen(char *s)
{
    char *p=s;
    while (*s++ !='\0')
        ;
    return s-p;
}
```

- Argument/variable `s` is local, so we can change it
- Pointer increment, dereference and comparison all in one! No statement in the loop body
- Note pointer subtraction at return statement



# Dynamic memory allocation

---

- Pointers are not much use with statically allocated data
- Library function `malloc` allocates a chunk of memory at run time and returns the address

```
int *p;  
if ((p = malloc(n*sizeof(int))) == NULL) {  
    // Error  
}  
  
...  
free(p); // release the allocated memory
```



# Pointers to pointers

---

- Consider an array of strings:  
`char *strTable[10];`
- The strings are dynamically allocated  $\Rightarrow$  any size
- But the table size is fixed to 10 strings
- How can we have both dynamically changing in size at runtime?  
`char **strTable;`
- Since a pointer is a variable, we could have another pointer pointing to it: pointer to pointer!



# Pointers to pointers - details

---

- Space must be allocated both for the table and the strings themselves

```
char **strTable;
strTable = malloc(n*sizeof(char *));
for (i=0; i < n; i++) {
    ...
    // s gets a string of length l
    *(strTable+i) = malloc(l*sizeof(char));
    strcpy(strTable[i], s);
}
// strTable[i][j] == (*(strTable+i)+j)
```



# That's all folks

---

- Not all C features have been covered, but this introduction should be enough to get you started
- Useful things to learn on your own:
  - Standard input/output: `printf`, `scanf`, `getc`, ...
  - File handling: `fopen`, `fscanf`, `fprintf`, ...
- Look over past exam papers for simple C programming exercises

