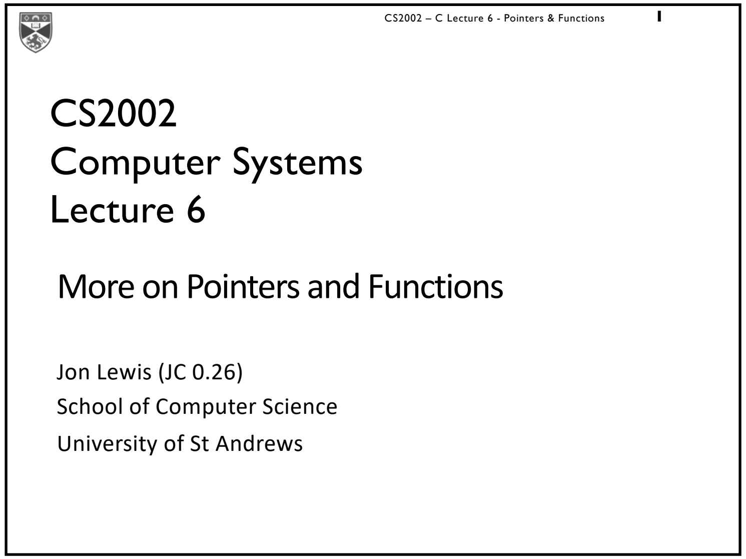
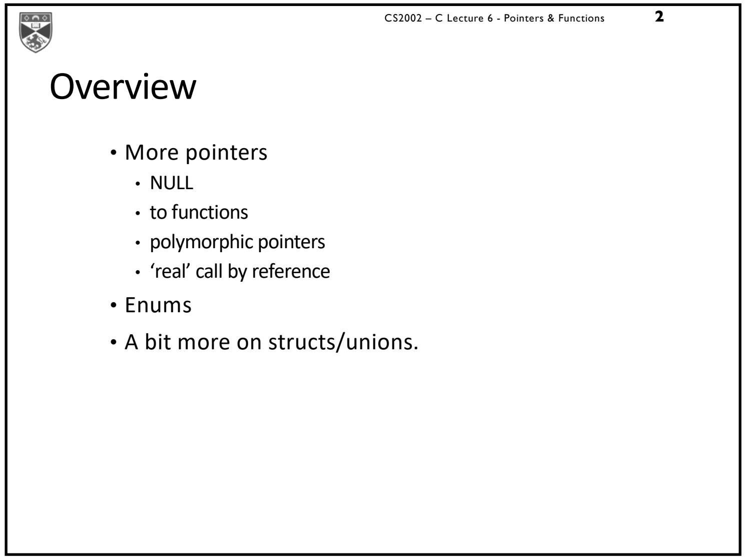


CS2002
Computer Systems
Lecture 6

More on Pointers and Functions

Jon Lewis (JC 0.26)
School of Computer Science
University of St Andrews

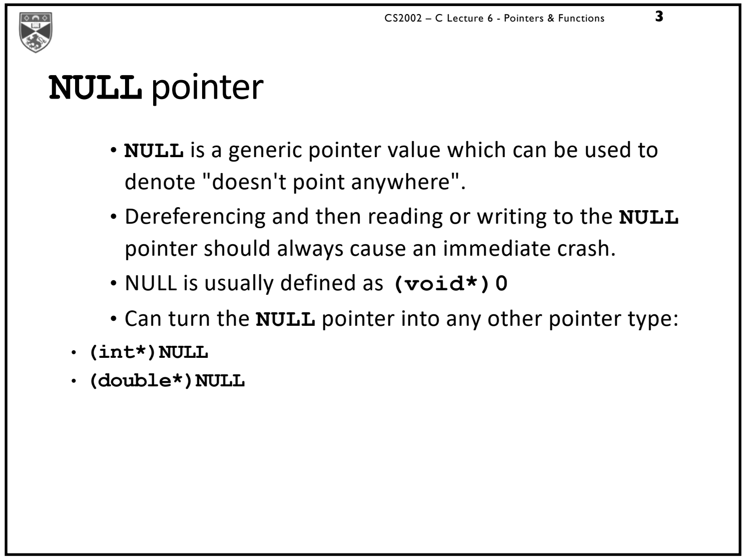
1



Overview

- More pointers
  - NULL
  - to functions
  - polymorphic pointers
  - ‘real’ call by reference
- Enums
- A bit more on structs/unions.

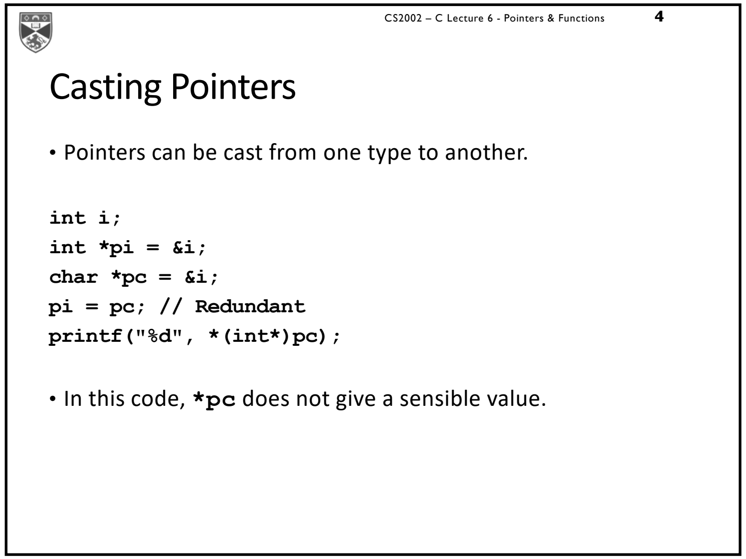
2



`NULL` pointer

- **`NULL`** is a generic pointer value which can be used to denote "doesn't point anywhere".
- Dereferencing and then reading or writing to the **`NULL`** pointer should always cause an immediate crash.
- NULL is usually defined as **`(void*)0`**
- Can turn the **`NULL`** pointer into any other pointer type:
- **`(int*)NULL`**
- **`(double*)NULL`**

3



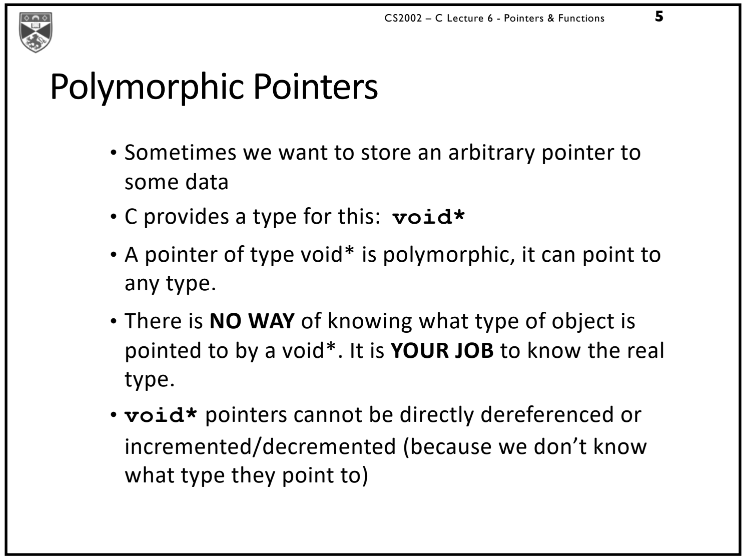
Casting Pointers

- Pointers can be cast from one type to another.

```
int i;
int *pi = &i;
char *pc = &i;
pi = pc; // Redundant
printf("%d", *(int*)pc);
```

- In this code, **`*pc`** does not give a sensible value.

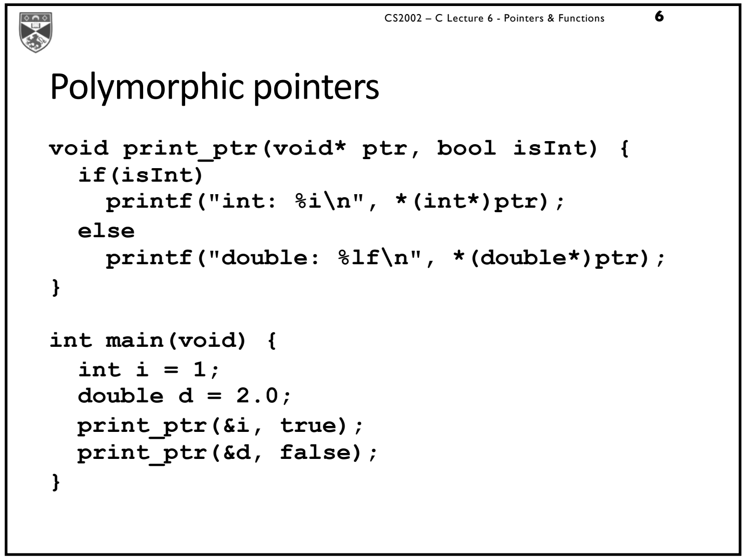
4



Polymorphic Pointers

- Sometimes we want to store an arbitrary pointer to some data
- C provides a type for this: `void*`
- A pointer of type void* is polymorphic, it can point to any type.
- There is **NO WAY** of knowing what type of object is pointed to by a void*. It is **YOUR JOB** to know the real type.
- `void*` pointers cannot be directly dereferenced or incremented/decremented (because we don't know what type they point to)

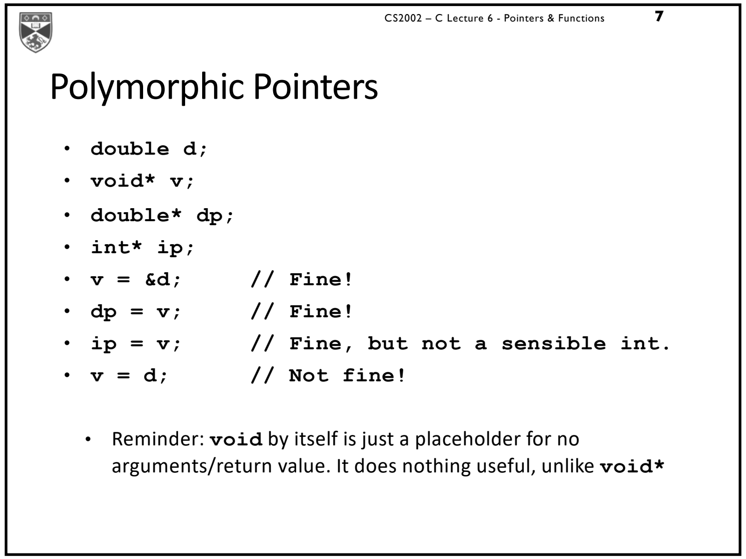
5



Polymorphic pointers

```
void print_ptr(void* ptr, bool isInt) {
  if(isInt)
    printf("int: %i\n", *(int*)ptr);
  else
    printf("double: %lf\n", *(double*)ptr);
}

int main(void) {
  int i = 1;
  double d = 2.0;
  print_ptr(&i, true);
  print_ptr(&d, false);
}
```

6

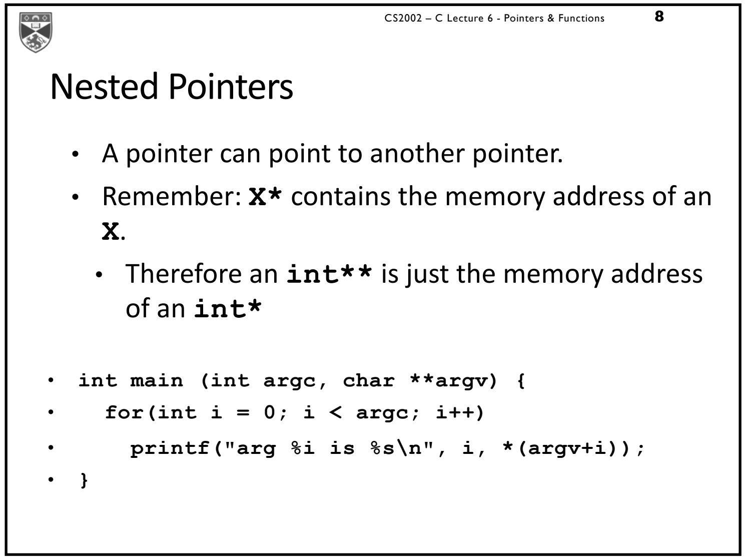


Polymorphic Pointers

```
• double d;
• void* v;
• double* dp;
• int* ip;
• v = &d;      // Fine!
• dp = v;      // Fine!
• ip = v;      // Fine, but not a sensible int.
• v = d;       // Not fine!
```

- Reminder: `void` by itself is just a placeholder for no arguments/return value. It does nothing useful, unlike `void*`

7

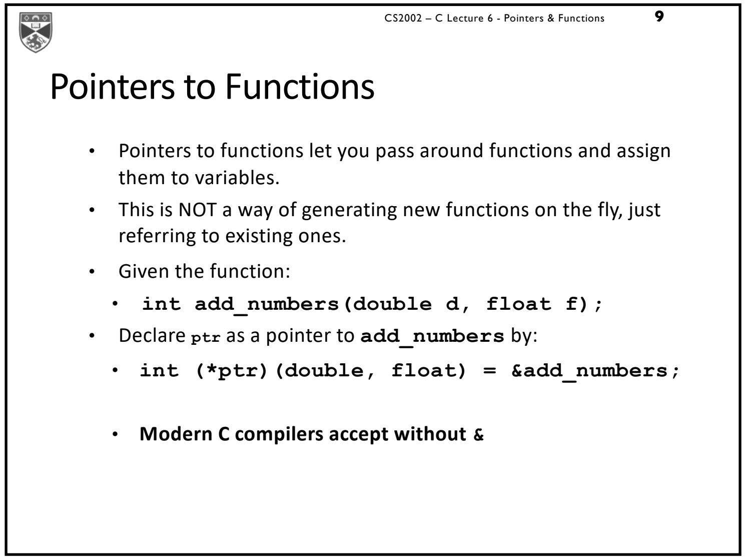


Nested Pointers

- A pointer can point to another pointer.
- Remember: `X*` contains the memory address of an `X`.
  - Therefore an `int**` is just the memory address of an `int*`

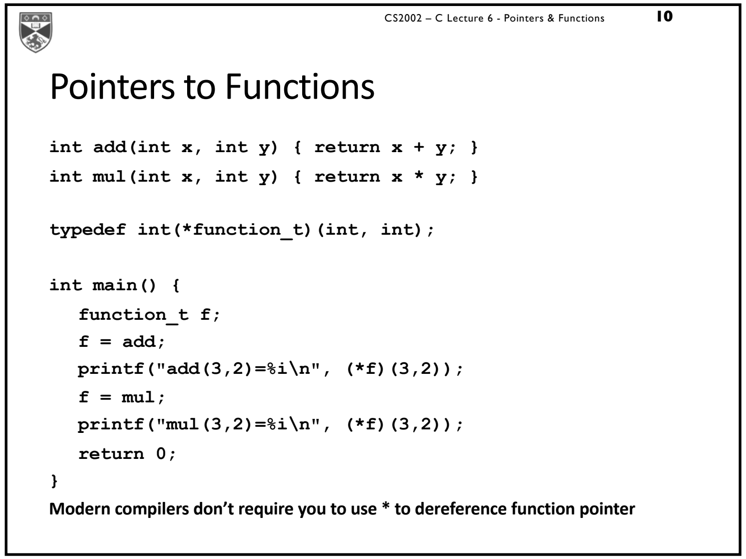
```
• int main (int argc, char **argv) {
•   for(int i = 0; i < argc; i++)
•     printf("arg %i is %s\n", i, *(argv+i));
• }
```

8



Pointers to Functions

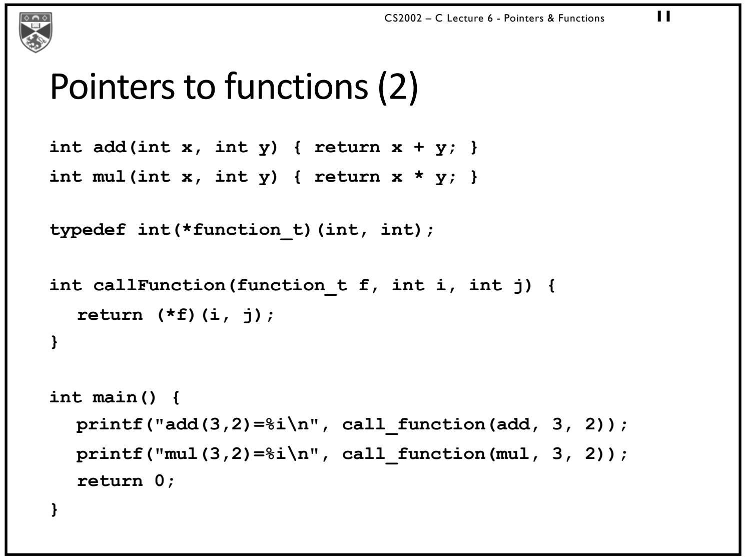
- Pointers to functions let you pass around functions and assign them to variables.
- This is NOT a way of generating new functions on the fly, just referring to existing ones.
- Given the function:
  - `int add_numbers(double d, float f);`
- Declare `ptr` as a pointer to `add_numbers` by:
  - `int (*ptr)(double, float) = &add_numbers;`
  - **Modern C compilers accept without `&`**



Pointers to Functions

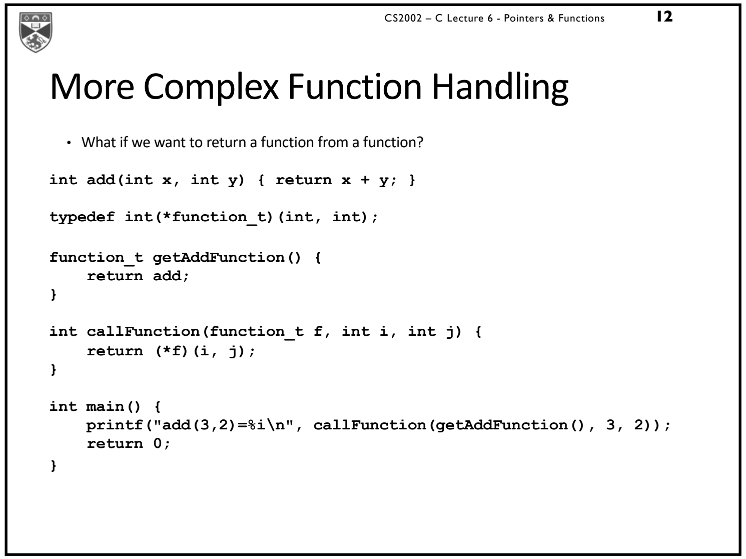
```
int add(int x, int y) { return x + y; }
int mul(int x, int y) { return x * y; }

typedef int(*function_t)(int, int);

int main() {
   function_t f;
   f = add;
   printf("add(3,2)=%i\n", (*f)(3,2));
   f = mul;
   printf("mul(3,2)=%i\n", (*f)(3,2));
   return 0;
}
```

Modern compilers don't require you to use * to dereference function pointer



Pointers to functions (2)

```
int add(int x, int y) { return x + y; }
int mul(int x, int y) { return x * y; }

typedef int(*function_t)(int, int);

int callFunction(function_t f, int i, int j) {
   return (*f)(i, j);
}

int main() {
   printf("add(3,2)=%i\n", call_function(add, 3, 2));
   printf("mul(3,2)=%i\n", call_function(mul, 3, 2));
   return 0;
}
```



More Complex Function Handling

- What if we want to return a function from a function?

```
int add(int x, int y) { return x + y; }

typedef int(*function_t)(int, int);

function_t getAddFunction() {
    return add;
}

int callFunction(function_t f, int i, int j) {
    return (*f)(i, j);
}

int main() {
    printf("add(3,2)=%i\n", callFunction(getAddFunction(), 3, 2));
    return 0;
}
```

Changing Variables in Functions

- Remember: Arguments to functions are passed by value. To change a X, you need to pass an X*.

```
void ptrchange(int* i) {
  *i = 2;    // Changes outside of fn
  i = NULL; // Does nothing outside of fn
}
```

Call by "Reference"

- Pointer arguments can be used to return multiple results from a function.

```
void set_ints(int *i1, int *i2, int v) {
  *i1 = v;
  *i2 = 2 * *i1; // same as 2*v
}

int main() {
  int i, j;
  set_ints(&i, &j, 3);
  printf("\ni=%i\tj=%i\n", i, j);
}
```

Call by Reference

This is (part) of how the scanf function works — all its arguments are passed by pointer.

```
void read2ints(int *i1, int *i2) {
  int i;
  scanf("%i %i", &i, i2);
  *i1 = i;
}
```

Enums

- Enums give a way of defining a set of constants.

```
enum tag { INT, DOUBLE };
```

Similar to:

```
#define INT 0
#define DOUBLE 1
```

Can also give explicit values. Values carry on:

```
enum tag { R = 2, D, F, S = -6, L };
```

Defines:

```
R=2, D=3, F=4, S=-6, L=-5
```

Names

- While you can have:

```
struct p { int i;};
typedef int p;
```

- You cannot have more than one of:

```
struct p;
union p;
enum p;
```

Improved struct-union Example

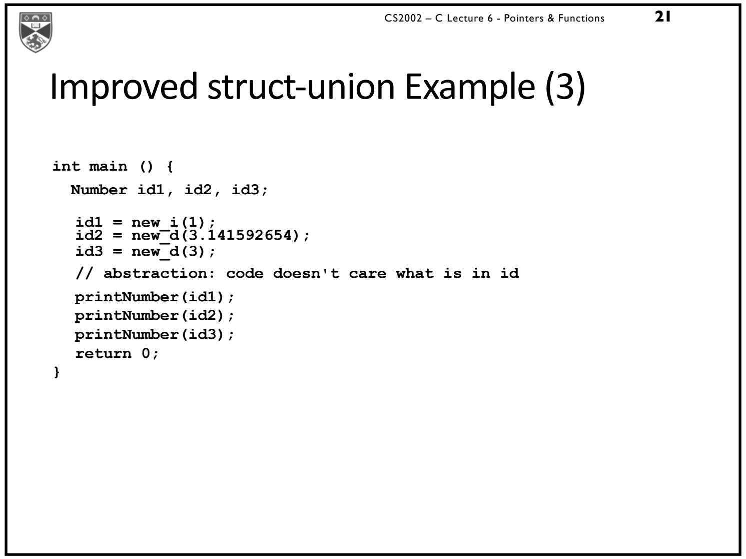
```
enum tag { INT, DOUBLE } ;
typedef enum tag Tag ;


typedef struct number {
  Tag tag ;
  union{ int i; double d; } val;
} Number;
```

Improved struct-union Example

```
enum tag { INT, DOUBLE } ;
typedef enum tag Tag ;

typedef struct number {                // define Number type
   Tag tag ;
   union{ int i; double d; } val;
} Number;

Number new_i ( int i )  {      // constructor for INT
   Number id;
   id.tag = INT;
   id.val.i = i;
   return id;
}

Number new_d( double d ) {     // constructor for DOUBLE
   Number id;
   id.tag = DOUBLE;
   id.val.d = d;
   return id;
}
```

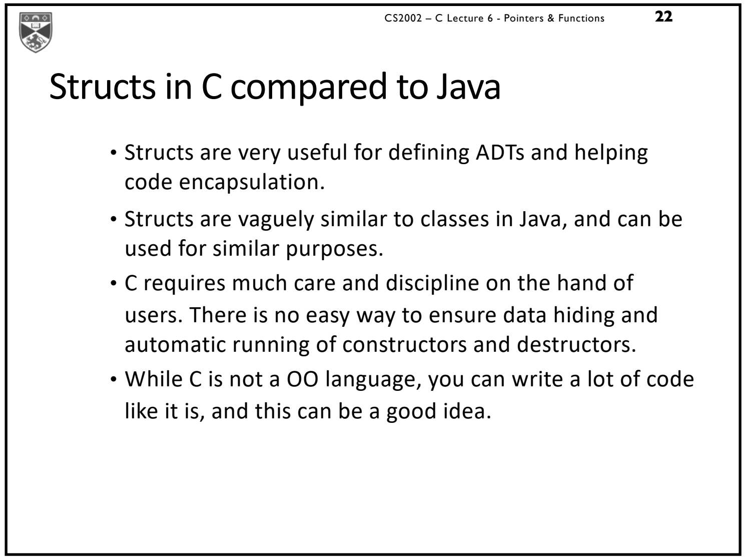
Improved struct-union Example (2)

```
void printNumber(Number id) {
   switch (id.tag) {
       case INT:
         printf("Int\t %i\n", id.val.i);
         break;
       case DOUBLE:
         printf("Double\t %.14f\n", id.val.d);
         break;
      default:
         printf("print_id: unknown tag = %i\n", id.tag);
   }
}
```



Improved struct-union Example (3)

```
int main () {
  Number id1, id2, id3;

  id1 = new_i(1);
  id2 = new_d(3.141592654);
  id3 = new_d(3);
  // abstraction: code doesn't care what is in id
  printNumber(id1);
  printNumber(id2);
  printNumber(id3);
  return 0;
}
```



Structs in C compared to Java

- Structs are very useful for defining ADTs and helping code encapsulation.
- Structs are vaguely similar to classes in Java, and can be used for similar purposes.
- C requires much care and discipline on the hand of users. There is no easy way to ensure data hiding and automatic running of constructors and destructors.
- While C is not a OO language, you can write a lot of code like it is, and this can be a good idea.