Pointers

pointers =~= arrays

- **an array expression <u>usually</u> "decays" into a pointer to element zero**

```
void display(size_t size, wibble * first);
void display(size_t size, wibble first[]);
```

these two declarations are equivalent*

```
wibble table[42] = { ... };
```

```
display(42, table);
display(42, &table[0]);
```

these two statements are equivalent

array decay

- 6.3.2.1 Lvalues, arrays, and function designators
  - para 3 - Except when it is the operand of
    - the sizeof operator
    - or the unary & operator,
    - or is a string literal used to initialize an array,
  - an expression that has type "array of type" is converted to an expression with type "pointer to type" and points to the initial element of the array object and is not an lvalue

There are three exceptions

- what does this program print?
  - assume sizeof(char*) == 8

```
typedef struct wibble
{
    char x[42];
} wibble;

wibble f(void);

void array_decay(void)
{
    char x[42];

    printf("%zd\n", sizeof(f().x));

    printf("%zd\n", sizeof(x));

    printf("%zd\n", sizeof(0, x));
}
```

exercise

answers

```c
typedef struct wibble
{
    char x[42];
} wibble;

wibble f(void);

void array_decay(void)
{
    char x[42];

    printf("%zd\n", sizeof(f().x));      → 42

    printf("%zd\n", sizeof(x));          → 42

    printf("%zd\n", sizeof(0, x));       → 8
}
```

string literals

- strings are arrays of char
  - automatically terminated with a null character, '\0'
  - a convenient string literal syntax

```
char greeting[] = "Bonjour";
```

equivalent to

```
char greeting[] =
    { 'B', 'o', 'n', 'j', 'o', 'u', 'r', '\0' };
```

There is no decay in this case

- what does this program print?

```
typedef struct wibble
{
    char x[42];
} wibble;

wibble f(void);

void array_decay(void)
{
    char x[42];

    printf("%zd\n",
        (char*)( x+1) - (char*) x);

    printf("%zd\n",
        (char*)(&x+1) - (char*)&x);

    printf("%zd\n",
               (&x+1) -        &x);
}
```

exercise

answers

```
typedef struct wibble
{
    char x[42];
} wibble;

wibble f(void);

void array_decay(void)
{
    char x[42];

    printf("%zd\n",
        (char*)( x+1) - (char*) x);    → 1

    printf("%zd\n",
        (char*)(&x+1) - (char*)&x);    → 42

    printf("%zd\n",
               (&x+1) -        &x);    → 1
}
```

restrict

- C99 library now uses restrict in its declarators where appropriate
  - says in code what would otherwise be said only in comments

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

char * strcpy(char * s1,
              const char * s2);
```

7.21 <string.h>

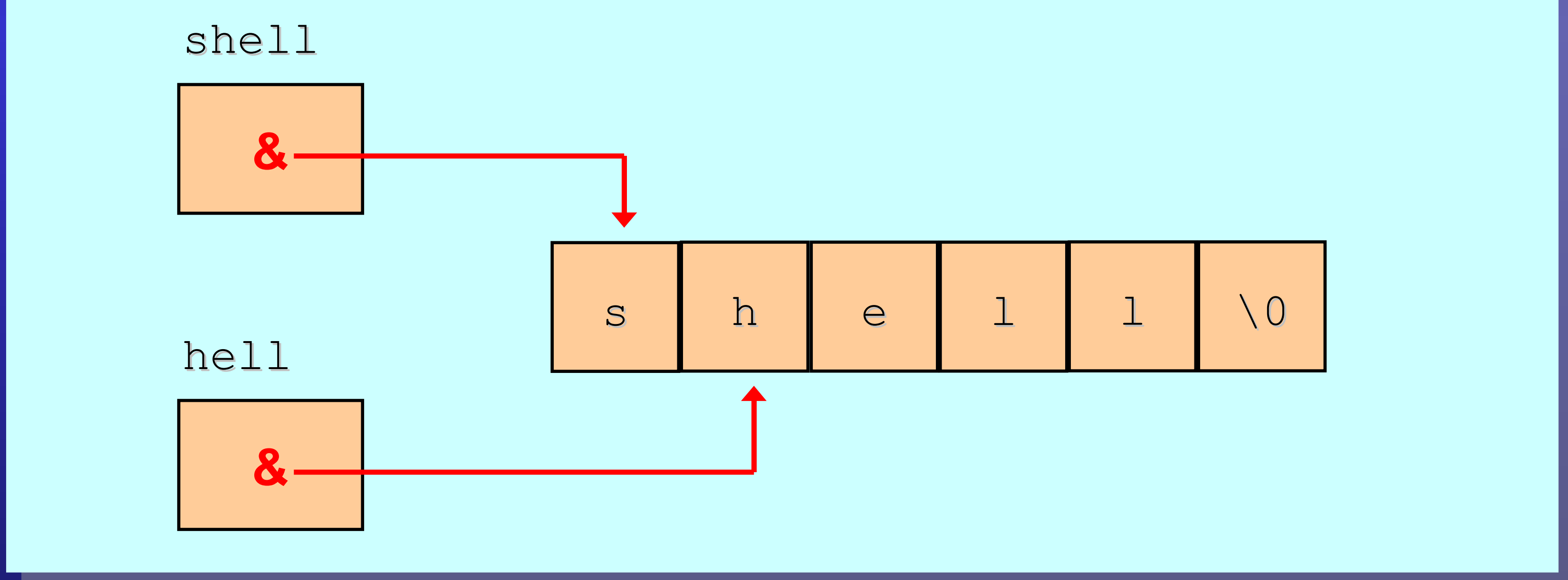
```
void * memcpy(void * restrict s1,
              const void * restrict s2,
              size_t n);

char * strcpy(char * restrict s1,
              const char * restrict s2);

If copying takes place between objects that
overlap the behaviour is undefined.
```

- duplicate string literals can be assigned to the same (static) storage location
  - be careful with restrict

```
const char * shell = "shell";
const char *  hell =  "hell";
```



s-hell

subscripting

- **6.5.2.1 Array subscripting**
  - para 2 – The definition of the subscript operator [ ] is that E1[E2] is identical to (*((E1)+(E2)))
- **6.5.6 Additive operators**
  - para 8 – If both the pointer operand and the result point to elements of the same array object, or one past the last element of the array object, the evaluation shall not produce an overflow; otherwise, the behavior is undefined.

Any pointer arithmetic that takes a pointer outside of the pointed to object...is undefined behavior.
There is no requirement that the pointer be dereferenced.

- which of these are conforming?

```
int array[10];

int * p1 = (array + 20) - 19;

int * p2 = array + 20 - 19;

int * p3 = array + (20 - 19);

int * p4 = array + 1;
```

exercise

answer

```
int array[10];
int * p1 = (array + 20) - 19;
int * p2 = array + 20 - 19;
int * p3 = array + (20 - 19);
int * p4 = array + 1;
```

question

- **typedef'ing a pointer...**

?

```
typedef struct date * date;

bool question(const date d);
```

...what is const?

is it the date pointer?
or
is it the date pointed to?

- the pointer is const!
- ***not*** the date pointed to!

```
typedef struct date * date;
```

```
bool question(const date d);
```

equivalent

```
bool question(struct date * const d);
```

answer

pointer typedef?

- so how about this...?
  - one typedef for plain ptr
  - another typedef for ptr to const

```
typedef       struct date *  date;
typedef const struct date * cdate;
```

```
void hide_ptr(date delay);

void hide_const_too(cdate delay);
```

- NO! it's a ***bad*** ***bad*** idea
  - hiding the pointer inside a typedef is not a good idea

```
typedef struct date
{
    int day;
    int month;
    int year;
} * date;
```

```
bool date_equal(const date lhs, const date rhs)
{
    return lhs->day == rhs->day &&
           lhs->month == rhs->month &&
           lhs->year = rhs->year;
}
```

✓ ☹

answer