CISC 372: Parallel Computing

C, part 2

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

siegel@udel.edu

Pointer arithmetic

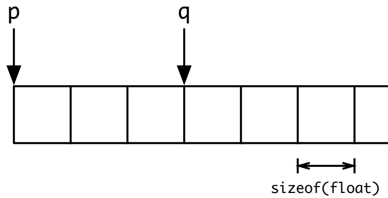
if all of the following hold

- p is an expression of type pointer-to-T and T is a complete type (size of T is known!!)
- i is an expression of integer type

then

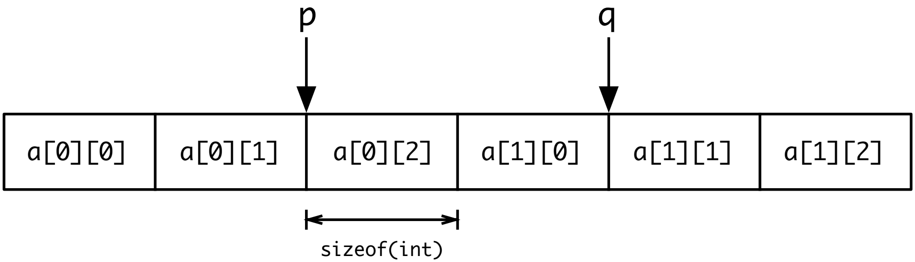
- `p+i` (= `i+p`) is an expression of type pointer-to-T
- it points to the address that is i T's past p
- if `sizeof(T)` is n bytes, then `p+i` is $i * n$ bytes after p

```
float a[10];
float *p = &a[0], *q = p+3, *r = q+7;
```



Pointer arithmetic within a 2d-array

```
int a[2][3];
int *p = &a[0][2];
int *q = p+2; // q == &a[1][1]
```



The real meaning of the index operator [..]

The meaning of `x[y]`:

- `x[y]` is syntactic sugar for `*(x+y)`
- if `p` is a pointer-to-T, then p[i] means `*(p+i)`
  - recall: this can be used to read or write to location `p+i`

Example: index operator and pointers

```c
#include <stdio.h>

/* assigns val to p[i], ..., p[i+n-1] */
void set_range(int *p, int n, int val) {
  for (int i=0; i<n; i++) p[i] = val;
}

/* prints p[0], ..., p[n-1] */
void print(int *p, int n) {
  for (int i=0; i<n; i++) printf("%d ", p[i]);
  printf("\n");
}

int main() {
  int a[10];
  set_range(&a[0], 10, 0); // a[0..9]=0
  print(&a[0], 10);
  set_range(&a[3],  5, 8); // a[3..7]=8
  print(&a[0], 10);
}
```

```
basie:c siegel$ cc ptr1.c
basie:c siegel$ ./a.out
0 0 0 0 0 0 0 0 0 0
0 0 0 8 8 8 8 8 0 0
basie:c
```

The type `void*`

- there is a special pointer type named `void*`

The type `void*`

- there is a special pointer type named `void*`
- the type pointed to could be anything

The type `void*`

- there is a special pointer type named `void*`
- the type pointed to could be anything
- a supertype of all pointer types

The type `void*`

- there is a special pointer type named `void*`
- the type pointed to could be anything
- a supertype of all pointer types
- any pointer type can be converted to `void*`
- any `void*` type can be converted to any pointer type

The type `void*`

- there is a special pointer type named `void*`
- the type pointed to could be anything
- a supertype of all pointer types
- any pointer type can be converted to `void*`
- any `void*` type can be converted to any pointer type
- converting from `T*` to `void*` then back to `T*` yields the original pointer

The type `void*`

- there is a special pointer type named `void*`
- the type pointed to could be anything
- a supertype of all pointer types
- any pointer type can be converted to `void*`
- any `void*` type can be converted to any pointer type
- converting from `T*` to `void*` then back to `T*` yields the original pointer
- this is necessary in order to design generic functions
  - consume a pointer to different kinds of data

The type `void*`

- there is a special pointer type named `void*`
- the type pointed to could be anything
- a supertype of all pointer types
- any pointer type can be converted to `void*`
- any `void*` type can be converted to any pointer type
- converting from `T*` to `void*` then back to `T*` yields the original pointer
- this is necessary in order to design generic functions
  - consume a pointer to different kinds of data
- restrictions
  - a `void` pointer can not be dereferenced (why?)

The type void*

- there is a special pointer type named void*
- the type pointed to could be anything
- a supertype of all pointer types
- any pointer type can be converted to void*
- any void* type can be converted to any pointer type
- converting from T* to void* then back to T* yields the original pointer
- this is necessary in order to design generic functions
  - consume a pointer to different kinds of data
- restrictions
  - a void pointer can not be dereferenced (why?)
  - you can not do pointer arithmetic on a void pointer (why?)

The type void*

- there is a special pointer type named `void*`
- the type pointed to could be anything
- a supertype of all pointer types
- any pointer type can be converted to `void*`
- any `void*` type can be converted to any pointer type
- converting from `T*` to `void*` then back to `T*` yields the original pointer
- this is necessary in order to design generic functions
  - consume a pointer to different kinds of data
- restrictions
  - a `void` pointer can not be dereferenced (why?)
  - you can not do pointer arithmetic on a `void` pointer (why?)
  - if you want to do these things, first cast to a non-void-pointer
    - ```
      void *p; ...
      int *q = (int*)p; // better be sure this is OK
      *q = *q + 10;
      ```
      ```

Example: void*

```
#include <assert.h>
int main() {
  int x = 5;
  int *p = &x;
  double y = 3.1415;
  double *q = &y;
  void *r;
  r = p; // conversion from int* to void*
  p = r; // conversion back to int*
  assert(*p == 5);
  r = q; // conversion from double* to void*
  q = r; // conversion back to double*
  assert(*q == 3.1415);
}
```

C's array-pointer "pun"

In most contexts:

- any expression of type *array-of-T* is automatically converted to an expression of type *pointer-to-T*
  - pointing to the first (i.e., 0-th) element of the array

C's array-pointer "pun"

In most contexts:

- any expression of type *array-of-T* is automatically converted to an expression of type *pointer-to-T*
  - pointing to the first (i.e., 0-th) element of the array
- i.e. `a` and `&a[0]` denote the same thing
  - the pointer to element 0 of array `a`

C's array-pointer "pun"

In most contexts:

- any expression of type *array-of-T* is automatically converted to an expression of type *pointer-to-T*
  - pointing to the first (i.e., 0-th) element of the array
- i.e. `a` and `&a[0]` denote the same thing
  - the pointer to element 0 of array `a`

```
#include <assert.h>
int main() {
  int a[10];
  int *p;
  p = a; // same as p=&a[0]
  assert(a[3] == *(p+3));
  assert(a[3] == *(a+3));
}
```

C's array-pointer "pun"

In most contexts:

- any expression of type *array-of-T* is automatically converted to an expression of type *pointer-to-T*
  - pointing to the first (i.e., 0-th) element of the array
- i.e. `a` and `&a[0]` denote the same thing
  - the pointer to element 0 of array `a`

```
#include <assert.h>
int main() {
  int a[10];
  int *p;
  p = a; // same as p=&a[0]
  assert(a[3] == *(p+3));
  assert(a[3] == *(a+3));
}
```

Exceptions: `sizeof` and a few other places

C's array pointer pun, cont.

- any formal parameter in a function header of type *array-of-T* is converted to type *pointer-to-T*

C's array pointer pun, cont.

- any formal parameter in a function header of type *array-of-T* is converted to type *pointer-to-T*
- example: the following all mean exactly the same thing:
  - `int f(double *a);`

C's array pointer pun, cont.

- any formal parameter in a function header of type *array-of-T* is converted to type *pointer-to-T*
- example: the following all mean exactly the same thing:
  - `int f(double *a);`
  - `int f(double a[]);`

C's array pointer pun, cont.

- any formal parameter in a function header of type *array-of-T* is converted to type *pointer-to-T*
- example: the following all mean exactly the same thing:
  - `int f(double *a);`
  - `int f(double a[]);`
  - `int f(double a[1000]);`
    - the `1000` is simply ignored
    - no reason to do this, unless it is as documentation

C's array pointer pun, cont.

- any formal parameter in a function header of type *array-of-T* is converted to type *pointer-to-T*
- example: the following all mean exactly the same thing:
  - `int f(double *a);`
  - `int f(double a[]);`
  - `int f(double a[1000]);`
    - the 1000 is simply ignored
    - no reason to do this, unless it is as documentation
- one difference: an array can not occur on left side of =

```
int a[10];
int b[10];
int *p;
p = a; // yes
p = b; // yes
a = p; // no!
a = b; // no!
```

Allocating sequences of data

Multiple ways:

1. `double a[10];`
   - in the file scope
   - allocates an array that persists for the entire life of the program
   - can be accessed in any scope
   - length must be a constant expression
   - cannot be used if length is unknown at compile time

Allocating sequences of data

Multiple ways:

1. `double a[10];`
   - in the file scope
   - allocates an array that persists for the entire life of the program
   - can be accessed in any scope
   - length must be a constant expression
   - cannot be used if length is unknown at compile time
2. `double a[n];`
   - in a local scope
   - allocates an array that persists until the end of that scope is reached
   - can be accessed in that scope and sub-scopes, and through pointers
   - length can be any integer expression

Allocating sequences of data

Multiple ways:

1. `double a[10];`
   - in the file scope
   - allocates an array that persists for the entire life of the program
   - can be accessed in any scope
   - length must be a constant expression
   - cannot be used if length is unknown at compile time
2. `double a[n];`
   - in a local scope
   - allocates an array that persists until the end of that scope is reached
   - can be accessed in that scope and sub-scopes, and through pointers
   - length can be any integer expression
3. `malloc` and `free`
   - dynamic memory allocation
   - memory allocated in the heap
   - programmer controls when allocation and deallocation occur
   - all accesses through pointers

Heap allocation: `malloc` and `free`

- `malloc` and `free` are functions defined in `stdlib`

Heap allocation: `malloc` and `free`

- `malloc` and `free` are functions defined in `stdlib`
- `malloc`
  - consumes argument of integer type
    - the number of bytes to allocate

Heap allocation: `malloc` and `free`

- `malloc` and `free` are functions defined in `stdlib`
- `malloc`
  - consumes argument of integer type
    - the number of bytes to allocate
  - allocates that many bytes in the heap

Heap allocation: `malloc` and `free`

- `malloc` and `free` are functions defined in `stdlib`
- `malloc`
  - consumes argument of integer type
    - the number of bytes to allocate
  - allocates that many bytes in the heap
  - returns `void*`
    - address of first byte allocated
    - typically, this is converted immediately into a non-void pointer type

Heap allocation: malloc and free

- malloc and free are functions defined in stdlib
- malloc
  - consumes argument of integer type
    - the number of bytes to allocate
  - allocates that many bytes in the heap
  - returns void*
    - address of first byte allocated
    - typically, this is converted immediately into a non-void pointer type
  - example
    - `int *p = (int*)malloc(10*sizeof(int));`
    - allocates space for 10 ints and returns pointer to beginning of that region

Heap allocation: `malloc` and `free`

- `malloc` and `free` are functions defined in `stdlib`
- `malloc`
  - consumes argument of integer type
    - the number of bytes to allocate
  - allocates that many bytes in the heap
  - returns `void*`
    - address of first byte allocated
    - typically, this is converted immediately into a non-void pointer type
  - example
    - `int *p = (int*)malloc(10*sizeof(int));`
    - allocates space for 10 `int`s and returns pointer to beginning of that region
- `free`
  - consumes a `void*` pointer previously produced by `malloc`
  - deallocates the object

Heap allocation: example

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

void print(int *p, int n) {
  for (int i=0; i<n; i++) printf("%d ", p[i]);
  printf("\n");
}

int main(int argc, char * argv[]) {
  int n = atoi(argv[1]); // converts first command-line arg to int
  int * p = malloc(n*sizeof(int));
  assert(p); // check that malloc succeeded
  for (int i=0; i<n; i++) p[i] = i;
  print(p, n);
  free(p);
}
```

```
basie:c siegel$ cc malloc1.c
basie:c siegel$ ./a.out 10
0 1 2 3 4 5 6 7 8 9
basie:c siegel$
```

Pointer types revisited

- declaration
  - if $T(x)$ declares x to have type T
  - then $T($`*p`$)$ declares `p` to have type *pointer-to-T*

Pointer types revisited

- declaration
  - if $T(x)$ declares x to have type T
  - then $T($`*p`$)$ declares `p` to have type *pointer-to-T*
- declaration examples
  - `double *p`
    - $T(x) =$ `double` x
    - $T($`*p`$) =$ `double *p`
    - `p` has type *pointer-to-double*

Pointer types revisited

- declaration
  - if $T(x)$ declares x to have type T
  - then $T($`*p`$)$ declares `p` to have type *pointer-to-T*
- declaration examples
  - `double *p`
    - $T(x) =$ `double` x
    - $T($`*p`$) =$ `double *p`
    - `p` has type *pointer-to-double*
  - `double (*p)[10]`
    - $T(x) =$ `double` x`[10]`
    - $T($`*p`$) =$ `double (*p)[10]`
    - `p` has type *pointer-to-array-of-length-10-of-double*

Pointer types revisited

- declaration
  - if $T(x)$ declares x to have type T
  - then $T($`*p`$)$ declares `p` to have type *pointer-to-T*
- declaration examples
  - `double *p`
    - $T(x) =$ `double` x
    - $T($`*p`$) =$ `double *p`
    - `p` has type *pointer-to-double*
  - `double (*p)[10]`
    - $T(x) =$ `double` x`[10]`
    - $T($`*p`$) =$ `double (*p)[10]`
    - `p` has type *pointer-to-array-of-length-10-of-double*
- the parentheses around `*p` are necessary
  - `[]` binds more tightly than `*`
  - `*a[]` = `*(a[])` : `a` has type *array-of-pointer-to-...*
  - `(*p)[]` : `p` has type *pointer-to-array-of-...*

Reading type declarations

- the rules above means types are specified "from the inside, out"

Reading type declarations

- the rules above means types are specified “from the inside, out”
- think of declaration as a sequence of unary operations applied to variable of form `[]` and `*`

Reading type declarations

- the rules above means types are specified "from the inside, out"
- think of declaration as a sequence of unary operations applied to variable of form `[]` and `*`
- Example: what is the type of `a` declared by: `double a[n][m]`
  - array-of-length-n-of-(array-of-length-m-of-double)
  - written hierarchically:

```
array-of-length-n-of
  array-of-length-m-of
    double
```

Reading type declarations

- the rules above means types are specified “from the inside, out”
- think of declaration as a sequence of unary operations applied to variable of form `[]` and `*`
- Example: what is the type of `a` declared by: `double a[n][m]`
  - array-of-length-n-of-(array-of-length-m-of-double)
  - written hierarchically:

```
array-of-length-n-of
  array-of-length-m-of
    double
```

- Example: what is the type of `p` declared by : `float **p`
  - pointer-to-(pointer-to-float)
  - ```
    pointer-to
      pointer-to
        float
    ```
    ```

Exercises: name the type

1. `char *p[n]`
2. `short (*p)[n]`
3. `unsigned int *p[n][m]`
4. `unsigned long int *(*p[n])`
5. `long *((*p)[n])`
6. `long *(*p)[n]`

Exercises: name the type

1. `char *p[n]`
   - array-of-length-*n*-of-pointer-to-`char`
2. `short (*p)[n]`
3. `unsigned int *p[n][m]`
4. `unsigned long int *(*p[n])`
5. `long *((*p)[n])`
6. `long *(*p)[n]`

Exercises: name the type

1. `char *p[n]`
   - array-of-length-*n*-of-pointer-to-`char`
2. `short (*p)[n]`
   - pointer-to-array-of-length-*n*-of-`short`
3. `unsigned int *p[n][m]`
4. `unsigned long int *(*p[n])`
5. `long *((*p)[n])`
6. `long *(*p)[n]`

Exercises: name the type

1. `char *p[n]`
   - array-of-length-*n*-of-pointer-to-`char`
2. `short (*p)[n]`
   - pointer-to-array-of-length-*n*-of-`short`
3. `unsigned int *p[n][m]`
   - array-of-length-*n*-of-array-of-length-*m*-of-pointer-to-`unsigned-int`
4. `unsigned long int *(*p[n])`
5. `long *((*p)[n])`
6. `long *(*p)[n]`

Exercises: name the type

1. `char *p[n]`
   - array-of-length-*n*-of-pointer-to-`char`
2. `short (*p)[n]`
   - pointer-to-array-of-length-*n*-of-`short`
3. `unsigned int *p[n][m]`
   - array-of-length-*n*-of-array-of-length-*m*-of-pointer-to-`unsigned-int`
4. `unsigned long int *(*p[n])`
   - array-of-length-n-of-pointer-to-pointer-to-`unsigned-long-int`
5. `long *((*p)[n])`

6. `long *(*p)[n]`

Exercises: name the type

1. `char *p[n]`
   - array-of-length-*n*-of-pointer-to-`char`
2. `short (*p)[n]`
   - pointer-to-array-of-length-*n*-of-`short`
3. `unsigned int *p[n][m]`
   - array-of-length-*n*-of-array-of-length-*m*-of-pointer-to-`unsigned-int`
4. `unsigned long int *(*p[n])`
   - array-of-length-n-of-pointer-to-pointer-to-`unsigned-long-int`
5. `long *((*p)[n])`
   - pointer-to-array-of-length-*n*-of-pointer-to-`long`
6. `long *(*p)[n]`

Exercises: name the type

1. `char *p[n]`
   - array-of-length-*n*-of-pointer-to-`char`
2. `short (*p)[n]`
   - pointer-to-array-of-length-*n*-of-`short`
3. `unsigned int *p[n][m]`
   - array-of-length-*n*-of-array-of-length-*m*-of-pointer-to-`unsigned-int`
4. `unsigned long int *(*p[n])`
   - array-of-length-n-of-pointer-to-pointer-to-`unsigned-long-int`
5. `long *((*p)[n])`
   - pointer-to-array-of-length-*n*-of-pointer-to-`long`
6. `long *(*p)[n]`
   - pointer-to-array-of-length-*n*-of-pointer-to-`long`

Construct the declaration for the given type name

1. declare a to have type

```
array of length n of
  pointer to
    array of length m of
      double
```

2. declare b to have type

```
array of length n1 of
  array of length n2 of
    pointer to
      array of length n3 of
        pointer to
          int
```

Construct the declaration for the given type name

1. declare `a` to have type

```
array of length n of
  pointer to
    array of length m of
      double
```

```
double (*a[n])[m]
```

2. declare `b` to have type

```
array of length n1 of
  array of length n2 of
    pointer to
      array of length n3 of
        pointer to
          int
```

Construct the declaration for the given type name

1. declare a to have type

```
array of length n of
  pointer to
    array of length m of
      double
```

```
double (*a[n])[m]
```

2. declare b to have type

```
array of length n1 of
  array of length n2 of
    pointer to
      array of length n3 of
        pointer to
          int
```

```
int *(*b[n1][n2])[n3]
```

C type names

- sometimes you need to name a type without declaring any variable
- `sizeof(int)`
- casts: `(int*)x`
- the type name is obtained by writing a variable delcaration and then erasing the variable
- `double (*a[n])[m]` $\rightarrow$ `double (*[n])[m]`

Heap-allocated 2d arrays: array of pointers

Heap-allocated 2d arrays: array of pointers

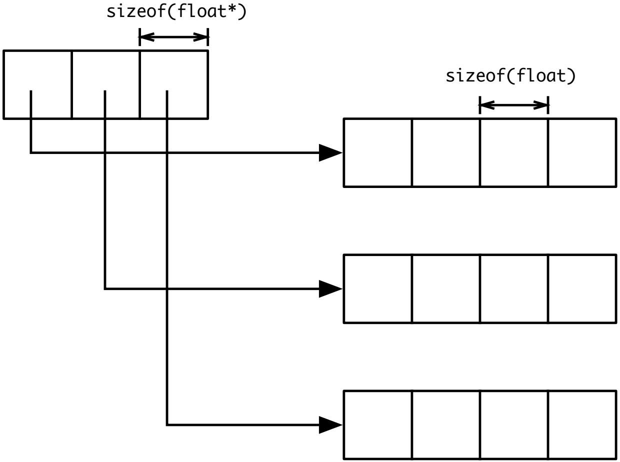
- problem: allocate on heap a 3×4 array of `float`s

Heap-allocated 2d arrays: array of pointers

- problem: allocate on heap a 3×4 array of `float`s
- solution: an array of length 3 of pointers
  - each pointer points to an array of length 4 of `float`s (one row)

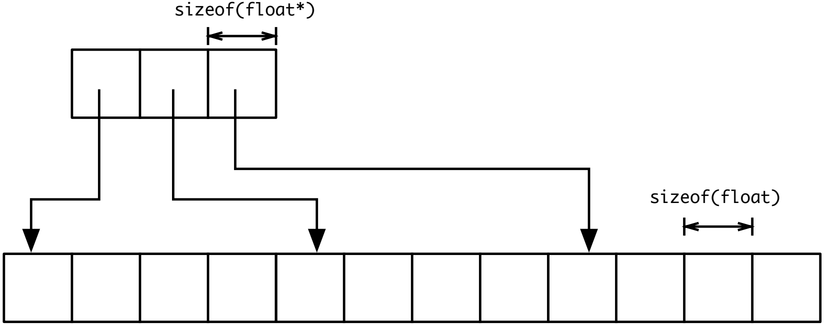
Heap-allocated 2d arrays: array of pointers

- problem: allocate on heap a 3×4 array of `floats`
- solution: an array of length 3 of pointers
  - each pointer points to an array of length 4 of `floats` (one row)



2d arrays: array of pointers: single allocation

- even better: allocate all rows at once in single `malloc` (see `array2d.c`)



Structures

The following defines a new type named `struct Show`:

```
struct Show {
  int channel; // this is an int field
  char * name; // this is a string field
  double cost; // this is a double field
};
```

```
struct Show show;
show.channel = 10;
show.name = "The 372 Show";
show.cost = 100000.00;
```

- `struct Show` is a type just like any other type
- can be used to declare variables, as function parameter type, can be returned by a function, . . .

Structures, cont.

It may be convenient to give the new type a shorter name:

```
typedef struct _show {
  int channel; // this is an int field
  char * name; // this is a string field
  double cost; // this is a double field
} Show;
```

- now you can just use `Show` instead of `struct _show`
- note: you can use the same name for the struct and the new type
  - `typedef struct X { ...} X;`

Structures and pointers

- structures are often manipulated using pointers
- functions consuming a structure typically consume a pointer to the structure
- functions returning structures typically return a pointer to a structure

Structures and pointers

- structures are often manipulated using pointers
- functions consuming a structure typically consume a pointer to the structure
- functions returning structures typically return a pointer to a structure

```
int getChannel(Show * show) {
  return (*show).channel;
}

void setChannel(Show * show, int c) {
  (*show).channel = c;
}
```

Structures and pointers

- structures are often manipulated using pointers
- functions consuming a structure typically consume a pointer to the structure
- functions returning structures typically return a pointer to a structure

```
int getChannel(Show * show) {
  return (*show).channel;
}

void setChannel(Show * show, int c) {
  (*show).channel = c;
}
```

- this pattern is so popular that C provides a shortcut
  - `s->x` is syntactic sugar for `(*s).x`

Structures and pointers, cont.

OK:

```
int getChannel(Show * show) {
  return (*show).channel;
}

void setChannel(Show * show, int c) {
  (*show).channel = c;
}
```

Structures and pointers, cont.

OK:

```
int getChannel(Show * show) {
  return (*show).channel;
}

void setChannel(Show * show, int c) {
  (*show).channel = c;
}
```

Better:

```
int getChannel(Show * show) {
  return show->channel;
}

void setChannel(Show * show, int c) {
  show->channel = c;
}
```

Arrays of structures

- one can create an array of structures, or
- one can create an array of pointers to structures.

Each has advantages (and disadvantages).

```
Show *shows[n]; // array of pointer to Show
for (int i=0; i<n; i++) {
  Show * s = (Show*)malloc(sizeof(Show));
  s->channel = i;
  shows[i] = s;
}
```

Type definitions, revisited

- `typedef` provides a way to give a type a name
- the name can be used wherever a type is called for
- a long or complicated type name can be given a simple short name
  - for convenience and readability
- a type that you may want to change in the future will only have to be changed in one place
- syntax: just like declaring a variable of that type, but add "`typedef`"
-
```
typedef unsigned long int nat;
nat x=0, y=0;
```
  - `nat` stands for the type unsigned-long-int

Type definitions, revisited, cont.

- ```
  struct node_s {
    int data;
    struct node_s *nxt;
  };
  typedef struct node_s * Node;
  ```
  - `Node` stands for the type pointer-to-struct-node_s
- ```
  typedef struct node_s {
    int data;
    struct node_s *nxt;
  } * Node;
  ```
  - same as above, just more condensed form