Pointers

- Every variable—even a `const`—has an address ("lvalue") and value ("rvalue"), e.g., (which used where?): `int a = 4; a ++;`

- Definition, initialization and assignment:

```
int  some_info, *in_ptr = &some_info, **int_buffer_ptr;
char ch, *char_index;    // allocation for _any_ pointer:
char_index = &ch;        //   (sizeof (int)) bytes (why?)
```

  note "`*`" in initialization, but not in assignment (why?)

- Dereferencing and pointer arithmetic:

```
*in_ptr += 2;    // == some_info += 2;
in_ptr += 2;    // == in_ptr increases by 2 * sizeof (int);
```

- Dereferencing $\Rightarrow$ point to something or set to 0 *at all times*

  Primarily for free store, unnamed memory allocation.

Constant Pointers

```
float *fp;
const float *cfp;    // cfp points to a "const float"
// fcp is a const pointer, to a float
float *const fcp = &some_float;
const float *const cfcp = &some_const_float;
```

	can change what it points to	can point to a `const float`	needs initialization (& cannot be changed)
`fp`	$\surd$	$\times$	$\times$
`cfp`	$\times$	$\surd$	$\times$
`fcp`	$\surd$	$\times$	$\surd$
`cfcp`	$\times$	$\surd$	$\surd$

Pointers to `const` are mainly for function arguments.

Arrays and Pointers

```
long id_numbers [5];
long *id_num_ptr = &(id_numbers [3]);
```

- Similarities
  - Both tags alone refer to address of its first element

    ```
    if (id_num_ptr == id_numbers) { /* ... */ }
    ```
  - $\Rightarrow$ both can use array and pointer syntax

    ```
    id_num_ptr [1] = *(id_numbers + 2);
    ```
- Differences
  - `id_numbers` only has an rvalue: *refers* to address of beginning of array and cannot be changed
  - `id_num_ptr` also has an lvalue: an extra `(long *)` is allocated and can be set to address of a `long`

1–D Pointer and Array Definitions

```
double *dp = 1000, da [5];
```

- Assume allocated at 4000 and 5000, resp.

- Bytes allocated
  - `dp`: `sizeof (void *)` [4]
  - `da`: `5 * sizeof (double)` [40]

- `sizeof (dp)` and `sizeof (da)` return these numbers

- Note: initialization to 1000 is a bad idea (why?); better:
  - `dp` gets set to address of `double`, e.g., `&(da [3])`
  - `dp` gets return value of `new()` (later)

1–D Pointers

syntax	type	lvalue	rvalue	comments
`dp`	`double *`	4000	1000	name alone
`dp + 2`	`double *`	—	1016	pointer arithmetic
Rule: in bytes: `dp + 2 * sizeof (double)` (i.e., remove 1 `*`)				
`*dp`	`double`	1000	$r(1000)^*$	pointer dereferencing
Rule: add a `*` of dereferencing $\Rightarrow$ drop a `*` of the type				
`dp [3]`	`double`	1024	$r(1024)^*$	array syntax
Rule: `foo [n]` is just `*(foo + n)` (a combination)				

$^*r(n)$ means whatever resident at memory location n

1–D Arrays

syntax	type	lvalue	rvalue	comments
`da`	`double []`	—	5000	name alone
Rule: array without `[]` is *rvalue* of array beginning address				
Rule: `double []` is *like* a `double *` (but not exactly)				
..... same as pointer				

- Note: the size of 5 `doubles` was *only* used for
  - initial allocation
  - future invocations of `sizeof()`

Now for 2–D arrays ....

2–D Pointer and Array Definitions

```
double **dpp = 2000, dm [7][11];
```

- Assume allocated at 6000 and 7000, resp.
- Bytes allocated
  - `dpp`: `sizeof (void *)` [4]
  - `dm`: `7 * 11 * sizeof (double)` [616]
- `sizeof (dpp)` and `sizeof (dm)` return these numbers
- Again, initialization to 2000 is a bad idea
- Now for same analysis, with (almost) same rules

2–D Pointers

syntax	type	lvalue	rvalue	comments
`dpp`	`double **`	6000	2000	name alone
`dpp + 2`	`double **`	—	2008	pointer arithmetic
`*dpp`	`double *`	2000	$r(2000)^*$	pointer dereferencing
`**dpp`	`double`	9788	$r(9788)$	multiple dereferencing
`dpp [3]`	`double *`	2012	$r(2012)^+$	array syntax
Rule: `foo [n]` is just `*(foo + n)`				
`dpp [3][5]`	`double`	8884	$r(8884)$	multi-array syntax
Rule: `foo [n][m]` is just `*(*(foo + n) + m)`				

*say, 9788; +say, 8844

2–D Arrays

syntax	type	lvalue	rvalue	comments
`dm`	`double [][11]`	—	7000	name alone
Rule: compiler needs to know how to jump, e.g., `dm [1][0]`				
Rule: array type retains all dimensions except first				
`dm + 2`*	`double [][11]`	—	7176	partial pointer syntax
Rule: `dm + n` is just rvalue of `&(dm [n][0])`				
`dm [2]`*	`double []`	—	7176	partial array syntax
Rule: `dm [n]` is just rvalue of `&(dm [n][0])`				
`dm [2][4]`	`double`	7208	$r(7208)$	full array syntax

*note the same values, but different types (and ∴ arithmetic)