Memory Model

Memory is like an array of bytes.

We say “addresses” for indexes.

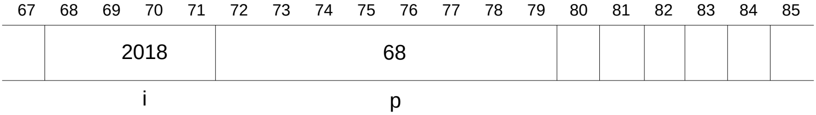
Refinement of A48 story: A variable may occupy multiple consecutive bytes, depending on type. Address refers to the first occupied byte.

“Pointer” = a variable/parameter that stores an address.

Confusing/Exciting: Since a pointer is a variable, it lives in memory and has its address!

Memory Model

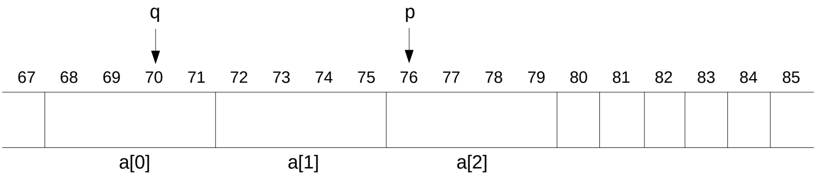
```
int i;
int *p;
i = 2018;
p = &i;
```



(Fictional addresses, inspired by true story.)

Memory Model: Array, Address Arithmetic

```
int a[3];
int *p = a + 2;          // 68 + 2*sizeof(int)
char *q = (char*)a + 2;  // 68 + 2*sizeof(char)
```



Compiler translates “2” to “2×sizeof(type)”.

(Fictional addresses, inspired by true story.)

Important Memory Regions

(On most platforms)

Some important memory regions:

- Text (code): Stores code. Function pointers usually point into here.
- Global: Stores global variables.
- Stack: For function calls. Holds local variables and return address. (Supports recursive calls.) Automatic allocation at call, deallocation at return.
- Heap: Manual allocation and deallocation, e.g., malloc, free. Good for dynamic data that needs to live beyond function return.
  (Unrelated to priorty queue's heap.)

Global Variables

Two kinds: top-level, function private.
Top-level subkinds: whole program, module only. (Difference when you split code into multiple files. Future lecture.)

```
int public_var = 10;
static int module_var = 50;

void f(void)
{
  static int private_var = 0;
  public_var++;
  private_var++;
...
```

Code: global.c

Integer Types

All combinations:
{signed, unsigned} $\times$ {char, short, int, long, long long}
Default signed, except char—depends on platform.
Abbreviations e.g., “unsigned” = unsigned int, “long” = long int.

Sizes and ranges depend on platform. On x86-64:

char (default signed)	1 byte
short	2 bytes
int	4 bytes
long	8 bytes
long long	8 bytes
~~ling ling~~	~~40 bytes~~

Code file: intsizes.c

Integer Literal Notation

example	type
3	int
'c'	int
3U	unsigned int
3L	long
3UL	unsigned long
3LL	long long
3ULL	unsigned long long

(Lowercase u and l also OK.)

Why important:
Good: `printf("%lu\n", 3UL);`
Bad: `printf("%lu\n", 3);`

Number Type Conversion

E.g., suppose `int i; char c; double d;`

`i = c;` (small integer to big integer)
Safe conversion.

`c = i;` (big integer to small integer)
Safe if actual value fits. (Detailed rules if not, I won't cover.)

`d = i;` (integer to floating-point)
Safe if within range. Approximation if can't be exact.

`i = d;` (floating-point to integer)
Truncate towards zero. Safe if truncated value fits.
Consider writing explicit `i = (int)d;` for human readers.

Applies to function calls too, e.g., `void f(int)` but you call `f(c)`.

Implicit Number Promotion

Applies to both integers and floating-point

E.g., `x/y` but `x` and `y` have different number types.

Pretty complicated rules. Approximately: convert “narrower (range)” operand type to match “wider” operand type. BUT: char and short are always promoted to at least int.

Example:

Suppose `double d; char i, j;`

`i/j` promote both to int, integer division.

`d/j` promote `j` to double, floating-point division.

Code: promote.c

Enumeration Types

```
enum rps { ROCK, PAPER, SCISSORS };
// ROCK=0, PAPER=1, SCISSORS=2
enum coin { HEAD, TAIL };
// HEAD=0, TAIL=1
```

“new” types and new integer constant names.

```
enum rps a;
enum coin c;

a = PAPER;
c = HEAD;
```

Code: enum.c

C Enumeration Types Are Fake

New integer constant names yes, new types no.

```
enum rps a;
enum coin c;
int i;
a = TAIL;
c = 10;
i = SCISSORS;
```

Bottomline: Enumeration “types” = int, mixable, not checked. Good for “meaningful” names only.

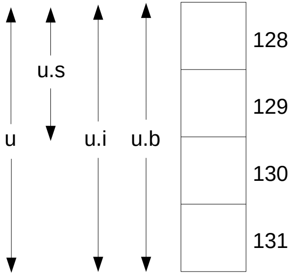
Code: enum.c

Advertisement: Scala, Rust, Haskell have **real** enumeration types, checked, not mixable.

Union Types

```
union my_union {
  unsigned short s;
  unsigned int i;
  unsigned char b[4];
};

union my_union u;
// use u.s, u.i, u.b[0] etc.
```



Use case: Your data have 3 mutually exclusive cases.

You need your own way to remember which case it is.

Tagged Union Idiom

I want an array in which some elements are int, others are double.

```
struct int_or_double {
  enum { INT, DOUBLE } tag;
  union {
    int i;
    double d;
  } data;
};
struct int_or_double a[10];
```

Idiom: Make an outer struct:

- tag field remembers which case you're in
- union of the cases

Set/check tag manually. Error-prone. Advertisement: Scala, Rust, Haskell do it for you, no bug.

Code: taggedunion.c

Type Alias: ‘typedef’

If you get tired of writing out ‘`struct node`’ all the time:

```
typedef struct node {
  int i;
  struct node *next;
  // "nodetype" not available here
} nodetype;

nodetype *p = malloc(sizeof(nodetype));
```

`typedef` is general, can also do e.g.

```
typedef double temperature;
typedef double *ptr_to_double;
typedef enum coin { HEAD, TAIL } cointype;
typedef union mu { ... } mutype;
```

Type Alias: ‘typedef’

If you also get tired of thinking up a 2nd name:

```
typedef struct node {
  int i;
  struct node *next;
} node;

typedef enum coin { HEAD, TAIL } coin;

node *p = malloc(sizeof(node));
coin c = HEAD;
```

No name clash. (Think about it.)

Type Alias: 'typedef'

Hell, this is legal too (DONT' DO IT):

```
typedef struct node {
  int i;
  struct node *next;
} coin;

typedef enum coin { HEAD, TAIL } node;
```

How to Read/Write Difficult typedefs

```
typedef double *pd;
```

How to figure out `pd` stand for pointer to double:

1. Ignore `typedef`, pretend var declaration
   `double *pd;`
2. What would be the type? Answer: pointer to double.
3. Put back `typedef`, conclude: `pd` stands for pointer to double.

Function Pointers

Variables `f` and `g` point to: function that takes 2 `char` parameters and returns `int`:

```
int (*f)(char, char);
int (*g)(char x, char y);
// param names optional and ignored
```

How to read/write:

f is a pointer	`(*f)`
to a function	`(*f)(...)`
2 char parameters	`(*f)(char, char)`
returns int	`int (*f)(char, char)`

Code: funptr.c

Exercise: What would `int *f(char, char);` mean? This explains parenthesizing.

Function Pointers (Epic)

Let's up the game! Suppose I have defined `F_out`:

```
typedef char (*F_out)(int);
```

Function Pointers (Epic)

Let's up the game! Suppose I have defined `F_out`:

```
typedef char (*F_out)(int);
```

- `h` is pointer to function
  `(*h)(...)`

Function Pointers (Epic)

Let's up the game! Suppose I have defined `F_out`:

```
typedef char (*F_out)(int);
```

- `h` is pointer to function
  `(*h)(...)`
- param `f` is pointer to function like last slide
  `(*h)(int (*f)(char, char))`
  (Note: name `f` is optional and ignored.)

Function Pointers (Epic)

Let's up the game! Suppose I have defined `F_out`:

```
typedef char (*F_out)(int);
```

- `h` is pointer to function
  `(*h)(...)`
- param `f` is pointer to function like last slide
  `(*h)(int (*f)(char, char))`
  (Note: name `f` is optional and ignored.)
- returns `F_out`, which is another kind of fun ptr:
  `F_out (*h)(int (*f)(char, char))`

Function Pointers (Epic)

Let's up the game! Suppose I have defined F_out:

```
typedef char (*F_out)(int);
```

- h is pointer to function
  `(*h)(...)`
- param f is pointer to function like last slide
  `(*h)(int (*f)(char, char))`
  (Note: name f is optional and ignored.)
- returns F_out, which is another kind of fun ptr:
  `F_out (*h)(int (*f)(char, char))`

Some people use typedef to break it up:

```
typedef int (*F_in)(char, char);
F_out (*h)(F_in f);
```