Memory Management

malloc
free
returns a pointer
requires a pointer
pointers ⇒ a reference to memory

we've gotten a chunk of memory to store data

\[
\text{12 + 14} = \text{myvar}
\]

\(\text{data stored here}\)
to declare var we specify a type

int i

float d

char c

specify size & limit access
\texttt{sizeof} ( \texttt{name of a var type} )

\# of bytes to store variable

\texttt{Future proofs our code}
typedef struct record

malloc(10 * sizeof(record));

- safe even if change platform

- safe even if we redefining record
what about malloc's return type?

malloc → pointer

int* ipj
float* fpj

→ void pointer
```c
float* fp = (float*) malloc (sizeof(float));
```

explicitly cast

Alignment restrictions

usually don't need to worry

so the next address is a multiple of data size.
Byte-ordering

0x1234

\[
\begin{array}{c|c}
12 & 34 \\
\hline
34 & 12 \\
\end{array}
\]
```c
struct E
{
    int f1, j;
    int f2, j;
    char f3, j;
};
Palm pilot
  
  preference database
```
// read in

struct prefrec mypref;

var type var name

void* pp = NULL; // points to pref blob

memcpy(mypref, pp, sizeof(struct prefrec));

works until endianness changes
File I/O

#include <stdio.h>

FILE <-- data type declared for you

is distinct from the file name

fopen("name", S x x e )
fopen → FILE *
fprintf(FILE*, "%s", str);
fopen fclose
byte-oriented functions
→ fread fwrite
higher level
→ fprintf fputc
variable
```c
scanf(); formatted
fscanf(); reads

printf("This 'zs' is a string
    in quotes\n", str_data);
scanf("%s", str_data);
scanf("%d %d %d", &i1, &i2, &i3);
    # of items successfully read
```