Last time:

- Choose good **data structures**
- Be aware of how you use the **heap**
- Use **compiler optimization flags**
- Using the stack: function call overhead and **function inlining**
- Doing things on the cheap — **strength reduction**
This time:

- Side effects and **pure** functions
- **const** and **restrict** keywords
- Less known compiler hints
Function side effects are **bad**

- A function side effect occurs when you modify something other than the return value of the function
  - Global variables
  - Function arguments

- Function side effects are bad for flexibility, and bad for performance.
More on function side effects

int func1(x) {
    return f(x) + f(x) + f(x) + f(x);
}

int func2(x) {
    return 4 * f(x);
}

• Are these two equivalent?
Do you know? Does the compiler know?
char *str = "Hello World!";
int i;
for (i = 0; i < strlen(str); i++)
{
    if (str[i] == '!') str[i] = '?';
}

• What’s the complexity?
• Can you improve it?
• Can the compiler improve it?
• Does the compiler know the implementation of strlen?
The "const" keyword

- The keyword “const” in front of an argument means it is read-only
- Can also be used for variables to make them into constants
- The memory object will never be written to
- This helps the compiler in analysis
Pure functions

- Pure functions have no side effects and are allowed to be called fewer times than the program specifies.

In gcc function declaration:

```c
double square(double* x) __attribute__((pure));
```

- Const functions are pure and read only their arguments (i.e. no global data and no heap).

```c
double square(double x) const;
```
The von Neumann Model

- Fetches instructions from the memory referenced by the program counter (PC) and computes results based on the data the instruction specified.
- The Arithmetical-Logic Unit (ALU) does the actual work.
- In this simple model access times to memory are uniform.
Aliasing

- Any pointer of unknown origin can reference a value that is accessed through another variable
- Any pointer might be used as an array
- Multiple “aliases” for the same memory location makes compile-time optimization very hard and may cause bugs!

```c
void xorSwap (int *x, int *y) {
    *x ^= *y;
    *y ^= *x;
    *x ^= *y;
}
```
• Constant propagation:

\[
\begin{align*}
\ldots \\
x &= 5; \\
*y &= 10; \\
z &= x^2; \\
\end{align*}
\]

Is \(z = 25\) or \(z = 100\)?
The "strict aliasing" rule

- Default mode in C99 and recent GCC
- Pointers of different types should not refer to the same memory
- Not a problem until you start being “clever”
- Significant compilation benefits
The "restrict" keyword

- restrict is another element of the C99 standard
- Available in many C/C++ compilers, including recent gcc (sometimes as __restrict)
- Within this context, any memory locations accessed by a restricted (pointer) variable will only be accessed through that pointer
- E.g. strcpy(char * restrict dest, char * restrict from)
More compiler hints

Function always returns a non-aliased pointer:

double ** myMatrixAllocation(int) __attribute__((malloc))

Save memory in a struct:

```c
struct foo{
    char a;
    int x[2] __attribute__((packed));
};
```
Summary of hardware independent optimization

- Choose data structures carefully
- Explore compiler optimization flags
- Avoid too many small function calls
- Move computations out of inner loops whenever possible
- Avoid function side effects
- const and restrict