```
void g(struct part part1)
{
  struct part part2 = part1;
  ...
}
```

The initializer doesn't have to be a variable or parameter name, although it does need to be an expression of the proper type. For example, `part2`'s initializer could be `*p`, where `p` is of type `struct part *`, or `f(part1)`, where `f` is a function that returns a `part` structure.

Uninitialized Variables

In previous chapters, we've implied that uninitialized variables have undefined values. That's not always true; the initial value of a variable depends on its storage duration:

- Variables with *automatic* storage duration have no default initial value. The initial value of an automatic variable can't be predicted and may be different each time the variable comes into existence.
- Variables with *static* storage duration have the value zero by default. Unlike memory allocated by `calloc`, which is simply set to zero bits, a static variable is correctly initialized based on its type: integer variables are initialized to 0, floating variables are initialized to 0.0, and pointer variables contain a null pointer.

`calloc` function ➤ *17.3*

As a matter of style, it's better to provide initializers for static variables rather than rely on the fact that they're guaranteed to be zero. If a program accesses a variable that hasn't been initialized explicitly, someone reading the program later can't easily determine whether the variable is assumed to be zero or whether it's initialized by an assignment somewhere in the program.

18.6 Inline Functions (C99)

C99 function declarations have an additional option that doesn't exist in C89: they may contain the keyword `inline`. This keyword is a new breed of declaration specifier, distinct from storage classes, type qualifiers, and type specifiers. To understand the effect of `inline`, we'll need to visualize the machine instructions that are generated by a C compiler to handle the process of calling a function and returning from a function.

At the machine level, several instructions may need to be executed to prepare for the call, the call itself requires jumping to the first instruction in the function, and there may be additional instructions executed by the function itself as it begins to execute. If the function has arguments, they'll need to be copied (because C passes its arguments by value). Returning from a function requires a similar