CS 201

Using External, Local, and Dynamic Variables

Gerson Robboy
Portland State University

In C we have 3 general ways of declaring or creating data

External or *statically allocated* variables

- Declared outside a function.

Local or *temporary* variables

- Declared inside a function

Dynamically allocated memory

- using the *malloc* family of functions

External or *"statically allocated"* variables

Declared outside of any function

Fixed size and declared data type

- **Can be structures, arrays, arrays of structures, …**
- **No limit on the size**

Statically allocated by the loader

Persists for the duration of the process

Two kinds:

- ***global* – scope is the entire program**
  - **Duplicate names in separate files is a bug**
- ***static* – scope is the C source file**

Local or *temporary* variables

Declared inside a function

Fixed size and declared data type

- **Can be structures, arrays, arrays of structures, …**
- **No limit on the size**

Created on the stack when a function is called

Persists until that function returns, then disappears

The scope is within that function only

- **Duplicate names in separate functions are distinct**

Function arguments are a variation of this

- **Also temporary, and also have limited scope**

Dynamically allocated memory

Allocated at run time using malloc, calloc, …

Variable size and no data type

Persists until it is explicitly freed

It has no name, no data type, and no scope.

- Depends on pointers to access it.
- The data type depends on the type of the pointer.

Temporary variables are our friends

Limited scope is good for structured programming

- hide information
- avoid name-space conflicts and side effects
- modularity makes the program easier to understand

Automatic garbage collection

- conserves memory
- no need to keep track of when and where to free allocated memory

Use temporary variables wherever possible

- More than that, design the program in order to use them
- When possible, pass information using function arguments rather than external data

External variables

When you need data to persist beyond one function.

When an object, such as a buffer, will be used repeatedly and it might as well just be declared externally once.

When data is needed in many different functions, and always passing it as an argument adds more complexity than it's worth.

- Especially constant data that will never be modified

Whenever possible, use "static" variables to limit the scope.

- Design the source file structure to make this possible.

Dynamic Allocation

When you don't know in advance how many things you will need.

When you don't know in advance how big it will be.

When you're going to use a big object for a while, but not forever

- Can free it to conserve memory

Reasons to avoid dynamic allocation

Pointer manipulation is prone to bugs

- Particularly hard ones to debug!

Lack of declared data type is also prone to bugs

Added complexity

- Keeping track of what's allocated and when to free it
  - Memory leaks
  - Avoiding freeing things twice
  - Avoiding losing the pointer to an allocated object

Example

A program like a shell has a main loop that looks something like this

```
main_loop(.. .)
{
    while(. . . ){
        get_command();
        parse_command();
        execute_command();
}
```

We need a buffer for the command lines. We don't know the length of each command until after we input it. What kind of data object should the buffer be? Where should it be declared?

Continuing with the shell example:

The function

```
char **parse_command(char *buf)
```

starts with a buffer full of text and returns an array of pointers to individual words in the text (*tokens*).

How and where do we declare the array of pointers to tokens? What about the content of the tokens themselves, that is, the actual strings?

Practice problem

Write the function

```
char **parse_command(char *buf)
```

Declare the variables you need.

Make these simplifying assumptions:

- **The buffer contains words separated by a single space**
- **The buffer contains at most 10 words**
- **We don't need to preserve the original contents of the buffer, and it can be modified.**

Example

A program is going to process some body of data and generate a list of data structures.

We know what the data structure looks like, but we don't know how many of them there will be. We will organize them in a linked list.

What kind of data object do we use and where do we declare it?

Example

One function of a program is going to process some body of data and generate a list of data structures.

We know what the data structure looks like, but we don't know how many of them there will be. Once this function has generated the list of structures, it can generate a result and then the list of structures is no longer needed.

What kind of data object do we use?

A function does calculations and returns a floating point value. What kind of thing should the variable be in which the value is returned?

A function does calculations and returns a pointer to an array of floating point values. What kind of thing should the variable be in which the return value (the pointer) is returned?

Practice Problem

Suppose a program needs to read lines of text from stdin, with no limit on how long a line may be.

Write a design for a function read_line, specifying what data objects it uses.

Write code for this function.

A certain function does calculations, generates an array of values, and returns a pointer to that array. The size of the array is constant. What kind of thing should the array be?

A certain function does calculations, generates an array of structures, and returns a pointer to that array. The size of the array is variable, and is known when this function executes. What kind of thing should the array be?

A certain function does calculations, generates an array of values, and returns a pointer to that array. The size of the array is variable and depends on input data. That is, the number of elements will be known only when the calculation is done. What kind of thing should the array be?

In the previous example, suppose we have an upper bound on the size of the array. Suppose we malloc the upper bound, even though it's way more memory than we're likely to need.

Is this a bad thing to do? Does this hog memory that other processes might use?

In the previous example, suppose there is no upper bound on the number of elements we may generate.

What might be a better way to organize the data than in an array?

Here's the beginning of a function that builds a binary tree of pointers to strings, sorting them into alphabetical order without copying strings.

```
typedef struct node Node;
struct node{
  char *string;
  Node *left;
  Node *right;
};
Node *Head = 0;
int insert(Node **node, char *string)
{
        int i;
        Node *n;

        if(*node == NULL){
                *node = (Node *)calloc(1, sizeof(Node));
```

Finish writing the function. The return value will be:

```
1 if the string was inserted in the tree
0 if the string was a duplicate, already in the tree
-1 on error
```

examples/insert.c

Review Problem

In one of our homework assignments, we had to pack two unsigned variables into a u_int64_t. Write code for this function:

```
u_int64_t u_to_ll(unsigned hi, unsigned lo);
```

Write three versions of the function:

- Using a union
- Using a pointer and no union
- Using casts and shifts but no pointers or union