Program Organization

Marcus Birkenkrahe

October 26, 2024

README

- This chapter covers program organization with regard to a program's chunks (modules, functions), objects (variables), and memory.
- In class, we'll be working with a wiki-type summary and interactive exercises. This section is accompanied by one bonus and by one mandatory programming assignment, as well as an online test.
- Much of this is based on King, C Programming (2008), chapter 10.

Memory organization

STACK	`func(int i);`	local variable
HEAP	`(int *) malloc(sizeof(int))`	pointer variable
GLOBALS	`int glob = 42;`	external variable
CONSTANTS	`char *str = "hello";`	string literal
CODE	`str db 'hello', 0`	assembly code

- **Stack:** The stack is the section of memory used for local variable storage. When you call a function, all of its local variables are created on the stack. When you leave the function, they are destroyed.
- **Heap**: The heap is for dynamic memory, data that are created ("allocated") when the program is running, and then hang around for a long time (until they are "freed" or deallocated).
- **Globals:** A global variable lives outside of all functions and is visible to all of them. They can be updated anywhere anytime. Created when the program runs for the first time.

- **Constants**: Created when the program first runs but stored in *read-only* memory. Examples are *string literals*.
- **Code:** This is where the code that the compiler has assembled, is located. It is also read-only, and sits at the lowest memory addresses.
- For example, the assembly code to define the string literal `char *str = "hello";` would look like this (in x86):

```
section .data
    str db 'hello', 0  ; Define a string 'hello' with a null terminator

section .bss
    ptr resb 4  ; Reserve 4 bytes for the pointer (assuming a 32-bit system)

section .text
    global _start

_start:
    ; Load the address of the string into the pointer
    mov eax, str      ; Move the address of 'hello' into EAX
    mov [ptr], eax    ; Store the address in the variable 'ptr'

    ; The rest of your code would go here

    ; Exit system call (Linux)
    mov eax, 1        ; sys_exit
    xor ebx, ebx      ; Exit code 0
    int 0x80
```

Storage and Scope

- A variable declared in the body of a function is *local* to the function.
- Which of the variables in the following example are local?

```
int sum_digits(int n)
{
  int sum = 0;

  while n > 0; {
    sum += n % 10;
    n /= 10;
  }
  return sum;
}
```

Answer: `sum` and `n`.

- Local variables have **automatic storage duration** - the storage is allocated when the function is called and deallocated upon return.
- Local variables have **block scope**: they are visible from the point of its declaration to the end of the enclosing function body.
- In C/C++ you can declare variables anywhere you like so the scope can be minimal. In the weird example below, `i` can never be used.

```
void f(void)
{
  // ...
  int i;   // variable with minimal scope
}
```

- Adding `static` to a local variable declaration gives it **static storage duration**: it now has permanent storage and retains its value throughout the execution of the program:

```
void func(void)
{
  static int i; // static local variable
  // ...
```

- Function **parameters** have the same properties - automatic storage duration and block scope - as local variables.

Type conversion

- For the computer to perform an arithmetic operation, the operands must be of the same size (same number of bits) and be stored in the same way (same data type).

- C allows the basic types to be mixed in expressions: you can combine `int` and `float`, and even `char` in a single expression. Can you explain the following `result`?

```
int i = 1;
float x = 1.f;
char c = 'a'; // ASCII value is 97
double result = i + x + c;
printf("%g\n", result);
```

```
99
```

- Explanation:

  > To compute `double result`, `i`, `x`, and `c` are implicitly converted to `double`. This is a *widening* conversion for the numeric variables `i` and `x`, and a conversion from `char` to `int` and from `int` to `double` for `c`. For the conversion to `int`, the numeric ASCII value of `c` is used.

```
char c = 'a'; // ASCII value is 97
printf("ASCII value of %c = %d\n",c,c);
```

```
ASCII value of a = 97
```

- C also allows *explicit conversion* using the *cast* operator, and explicit *type definitions* with `typedef`.

Implicit Arithmetic Conversions

- Because there are so many different arithmetic types[1]=, implicit conversion rules are complicated. A conversion is performed:

  1. When the operands don't have the same type (the usual case): `float f; int i = 2 * f;`

[1]Why does C have so many types? Because it works so close to machines and especially machine memory: to make good use of the machine, you need to capitalize on its architecture.

2. When the type on the left doesn't match the type on the right of an assignment operator: `float f; int i = f;`
3. When the type of an argument in a function call does not match the type of the corresponding parameter:

   ```
   void f(int i) {};
   float x;
   f(x);
   ```

4. When the type of the expression in a `return` statement does not match the functions `return` type:

   ```
   int f(int i) { return 0.f; };
   float x;
   f(x);
   ```

Type Definitions

- You can create a Boolean type `BOOL` with a macro, which then allows you to define variables of that type:

  ```
  #define BOOL int

  BOOL T = 1;
  BOOL F = 0;
  ```

- A better way is to use a *type definition*:

  ```
  typedef int Bool; // defines type ‘Bool‘ as ‘int‘

  Bool flag;
  ```

- To the compiler, `flag` is nothing but an `int` variable.
- What's the point?

  1. Type definitions make a program more understandable provided you've chosen meaningful names.
  2. Type definitions can make a program easier to modify.
  3. Type definitions help making programs portable.

- Example: variables `cash_in` and `cash_out` are used to store dollar amounts.

  1. Declaring a `Dollars` type is more informative than `float`:

     ```
     typedef float Dollars; // declare ‘Dollars‘ type
     Dollars cash_in, cash_out;
     ```

  2. If you later decide that `Dollars` is should be `double` instead of `float`, you only have to change `typedef double Dollars`.

- Portability is a big issue: types may have different ranges on different machines. The statement `int i = 100000;` works on a 32-bit machine, but fails on a 16-bit machine[2]

- Example: for a large warehouse program, you need variables capable of storing product quantities in the range [0,50000]. We could use `long` variables for that[3] but operations on `int` are faster and they take up less space. To use `int`, we define our own type:

  ```
  typedef int Quantity;

  Quantity q;
  ```

- When moving to a machine with shorter integers, change the definition:

  ```
  typedef long Quantity;
  ```

- The C library uses `typedef` to create names for types that can vary from one C implementation to another - e.g. `typedef unsigned long int size_t;` The `_t` signifies that these types can vary from machine to another.

- The `stdint.h` header uses `typedef` to define names for integers with a particular number of bits, e.g. `int32_t` is a signed integer type with exactly 32 bits to make programs more portable.

[2]Why does a 16-bit system difficulty with `int i = 100000;`? Because the largest signed whole number that can be represented with 16 bits is 2^16 = 65536, since every decimal number is represented as a sequence of bits or multiples of 2.

[3]Why would `long` work here? Because a `long int` variable can represent unsigned integers in the range [0,2^32] or [0,4294967296].

- Do you remember how to determine exactly how much memory is required to store values of a particular type?

  > The `sizeof` operator returns the number of bytes needed to store values of its argument type, e.g. if `i` and `j` are integers, then `sizeof(i)` is 4 on a 32-bit machine, as is `sizeof(i+j)`.

- Printing a memory value requires care because its type is `size_t` and depends on implementation. It is guaranteed to be an unsigned integer type. To be safe, cast it and print it as cast:

  ```
  printf("Size of int: %lu\n", (unsigned long) sizeof(int));
  ```

  ```
  Size of int: 4
  ```

- The `printf` function in C99 and later can display `size_t` directly:

  ```
  printf("Size of int: %zu\n", sizeof(int));
  ```

  ```
  Size of int: 4
  ```

External variables

- Functions can communicate through

  1. passing variables (see "storage and scope")
  2. external (or *global*) variables that are declared outside the body of any function. They have `static` storage duration (don't disappear until the program is finished), and *file scope* (rather than *block scope*): they can be accessed by all functions after the declaration of the external variable.

Organizing Programs

Our programs are going to get larger - and we've touched upon all of these structural components of programs[4]:

[4]In fact, we've not discussed *inline functions*, which are typically put into header files.

```
#include directives
#define directives
Type definitions
Declarations of external variables
Prototypes for functions other than main
Definition of main
Definitions of other functions
```

Precede each function definition by a boxed comment that gives the name of the function, explains its purposes, discusses the meaning of each parameter, describes its return value (if any), and lists any side effect it has (such as modifying external variables).

Using External Variables to Implement a Stack

- A stack, like an array, can store multiple data items of the same type. Remember this is where local variables are stored transiently.
- The only allowed operations are: `push` an item onto the stack (add it at one end, the stack top), or `pop` it from the stack (remove it from teh same end).
- Examining or modifying an item that is not at the top of the stack is forbidden.
- We implement a stack in C as an array `contents` with an integer variable `top` that marks the position of the stack top. An empty stack has `top = 0`.
- To push an item onto the stack, we store it in `contents` at the position `top`, then increment `top`. To pop an item, decrement `top`, then use it as an index to `contents` to fetch the item to be popped.
- Code for the stack implementation:

  ```
  #include <stdbool.h> // adds data type ‘bool‘
  #define STACK_SIZE 20

  /* external variables */
  int contents[STACK_SIZE];
  int top = 0;
  ```

```
/* function prototypes */
void make_empty(void); // set top to 0
bool is_empty(void); // check if top is 0 (stack empty)
bool is_full(void); // check if top is STACK_SIZE (stack full)
void push(int i); // push i onto stack (if not full)
int pop(void); // pop top stack element
int stack_overflow(void); // stack overflow (push to full stack)
int stack_underflow(void); // stack underflow (pop from empty stack)
int print_stack(); // print stack as array
/* main program */
int main(void)
{
  // push number onto stack
  push(1);
  print_stack();
  push(1);
  print_stack();
  //print_stack();
  // pop number from stack
  pop();
  print_stack();
  // pop number from stack (trigger underflow)
  //pop();
  // push STACK_SIZE numbers onto stack
  for (int i=0; i < STACK_SIZE; i++)
    push(1);
  // push number onto stack (trigger overflow)
  print_stack();
  //push(100);
  //push(101);
  return 0;
}
/*************************************************************/
// make_empty: set stack top index to zero
// no parameters (void), no return (void)
// modifies external variable ‘top‘
/*************************************************************/
void make_empty(void)
{
  top = 0;
}
/****************************************************************/
// is_empty: check if stack is empty and return Boolean
// no parameters (void), returns ‘bool‘
// checks if external variable ‘top‘ is zero
/****************************************************************/
bool is_empty(void)
{
  return top == 0;
}
/****************************************************************/
// is_full: check if stack is full and return Boolean
// no parameters (void), returns ‘bool‘
// checks if external variable ‘top‘ is STACK_SIZE (constant)
/****************************************************************/
bool is_full(void)
{
  return top == STACK_SIZE;
}
/****************************************************************/
// push: add integer to top of non-full stack
// takes integer parameter (int), returns nothing (void)
// uses external ‘top‘ as index for stack array ‘contents‘
// if stack is full, call stack_overflow
/****************************************************************/
void push(int i)
{
  if (is_full())
    stack_overflow();
  else
    contents[top++] = i;
  //return is_full() ? stack_overflow() : contents[top++]=i;
}
/****************************************************************/
// pop: extract integer from top of non-empty stack
// no parameters (void), returns integer
// uses external ‘top‘ as index for stack array ‘contents‘
// if stack is empty, call stack_underflow
/****************************************************************/
```

```
int pop(void)
{
  if (is_empty())
    stack_underflow();
  else
    return contents[--top];
}
/******************************************************************/
// stack_overflow: exits with 1 and aborts if stack is_full
// no parameter (void), returns integer (EXIT_FAILURE)
/******************************************************************/
int stack_overflow(void)
{
  printf("Stack overflow!\n");
  return EXIT_FAILURE;
}
/******************************************************************/
// stack_underflow: exits with 1 and aborts if stack is_empty
// no parameter (void), returns integer (EXIT_FAILURE)
/******************************************************************/
int stack_underflow(void)
{
  printf("Stack underflow!\n");
  return EXIT_FAILURE;
}
/******************************************************************/
// print_stack: print stack as array ‘contents‘; constant
// no parameter (void), returns integer from ‘contents‘ (external)
/******************************************************************/
const int print_stack(void)
{
  printf("top = %d\n",top);
  for (int i=0;i<STACK_SIZE;i++)
    printf("%d ", contents[i]);
  puts("");
}
```

```
top = 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
top = 2
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
top = 1
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Stack overflow!
top = 20
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

Guessing a number

- We'll write a simple game-playing program: the program generates a random number between 1 and 100, which the user attempts to guess in as few tries as possible.

  Sample output:

  ```
  A new number has been chosen.
  Enter guess: 55
  Too low; try again.
  Enter guess: 65
  Too high; try again.
  Enter guess: 60
  Too high; try again.
  Enter guess: 58
  You won in 4 guesses!

  Play again? (Y/N) Y

  A new number has been chosen.
  Enter guess: 78
  Too high; try again.
  Enter guess: 34
  You won in 2 guesses!
  ```

- Which tasks will this program have to carry out? Each of them will be put in a function:

  1. Initialize the random number generator
  2. Choosing a secret number randomly
  3. Read user guess, compute and print answers

- We write each of these functions first and then create the `main` program with the external variables, prototypes, function calls.
- `stdlib` has several functions to generate random numbers. We use `srand` from `stdlib.h`. The function requires a seed. It does not return any value, and it takes any number as seed value[5].
- As seed, we use the `time` function from `time.h`, which returns the current time encoded in a single number, the number of seconds that have elapsed since *the Epoch*, the starting point for Unix system time, defined as 00:00:00 UTC on January 1, 1970.

  ```
  #include <time.h>

  time_t current_time; // ‘time_t‘ is a portable data type for time data
  current_time = time(NULL); // return the current time but don’t store it anywhere
  printf("The current time in seconds since the Epoch: %ld\n", current_time);
  ```

  ```
  The current time in seconds since the Epoch: 1729971415
  ```

- Now the function to initialize the number generator (requires to `include stdlib.h` and `time.h`). We give it a `#+name` to use it with `noweb` later[6]:

  ```
  /*************************************************************/
  // initialize_number_generator: Initializes the random number
  //                              generator using the time of day.
  /*************************************************************/
  void initialize_number_generator(void)
  {
    srand ( (unsigned) time(NULL));
  }
  ```

[5]What is a "seed" value? Computer-generated random numbers aren’t actually random but pseudorandom numbers generated by some algorithm, that is they repeat albeit after a very long period. With a seed, the starting point for the number generation can be fixed so that you get the same set of random numbers every time. This is useful and necessary to obtain the sample test data.

[6]You’ve seen this in class: the Org-mode code block header argument `:noweb yes` allows you to use any named code block as a macro: whatever is in the code block named `#+name: code` will be copied to the location where `<<code>>` appears.

- To choose a secret number randomly, generate a random number with `stdlib::rand(void)`: this function chooses a pseudo-random integer in the range `[0,RAND_MAX]`, where `RAND_MAX` is defined in `stdlib`:

```
#include <stdlib.h>
printf("%d\n",RAND_MAX); // (2^32)/2 - 1
```

```
2147483647
```

- Here is an example. The first loop generates random numbers, and the second loop scales them to `[1,100]`:

```
#include <stdlib.h>
#include <time.h>
#define MAX_NUMBER 100;
int i, r1[10], r2[10];
srand(time(NULL)); // this guarantees different sets of random numbers
for (i=0;i<10;i++) {
  r1[i]=rand();
  r2[i]=r1[i] % MAX_NUMBER + 1;
  printf("%12d => %4d\n",r1[i],r2[i]);
 }
printf("%d",RAND_MAX);
```

```
   420753090 =>   90
  1486456311 =>   11
  1448943174 =>   74
   799568499 =>   99
   729487741 =>   41
   946063588 =>   88
   522468840 =>   40
  1717796384 =>   84
  2089874773 =>   73
  1082889291 =>   91
2147483647
```

- A function to choose a secret number and store it in an *external* variable `secret_number`:

```
/******************************************************************/
// choose_secret_number: Randomly selects number between 1 and
//                       MAX_NUMBER and stores it in secret_number
/******************************************************************/
#define MAX_NUMBER 100;
void choose_new_secret_number(void)
{
  int secret_number = rand() % MAX_NUMBER + 1;
}
```

- Our last function is a function to read the guesses and compare them to the secret number.

```
/********************************************************************/
// read_guesses: Repeatedly reads user guesses and tells the user
//               whether each guess is too low, too high, or correct.
//               When guess is correct, prints total number of guesses
//               and returns.
/********************************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0, secret_number;

  for (;;) { // infinite loop left if guess correct
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d",&guess);
    if (guess == secret_number) {
printf("You won in %d guesses!\n\n",num_guesses);
return; // only allowed in void function if no argument given
    } else if (guess < secret_number)
printf("Too low; try again.\n");
    else
printf("Too high; try again.\n");
  }
}
```

- Let's put this together:

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>

#define MAX_NUMBER 100 // secret number between 1 and MAX_NUMBER

int secret_number; // external variable

void initialize_number_generator(void);
void choose_new_secret_number(void);
void read_guesses(void);

int main(void)
{
  char command; // flag for quitting the game
  printf("Guess the secret number between 1 and %d.\n\n", MAX_NUMBER);
  initialize_number_generator();
  do {
    choose_new_secret_number();
    printf("A new number has been chosen.\n");
    read_guesses();
    printf("Play again? (Y/N) ");
    scanf(" %c", &command);
    printf("\n");
  } while (command == 'y' || command == 'Y');

  return 0;
}

/**************************************************************/
// initialize_number_generator: Initializes the random number
//                              generator using the time of day.
/**************************************************************/
void initialize_number_generator(void)
{
  srand ( (unsigned) time(NULL));
}
/****************************************************************/
// choose_secret_number: Randomly selects number between 1 and
//                       MAX_NUMBER and stores it in secret_number
/****************************************************************/
```

```
#define MAX_NUMBER 100;
void choose_new_secret_number(void)
{
  int secret_number = rand() % MAX_NUMBER + 1;
}
/****************************************************************/
// read_guesses: Repeatedly reads user guesses and tells the user
//               whether each guess is too low, too high, or correct.
//               When guess is correct, prints total number of guesses
//               and returns.
/****************************************************************/
void read_guesses(void)
{
  int guess, num_guesses = 0, secret_number;

  for (;;) { // infinite loop left if guess correct
    num_guesses++;
    printf("Enter guess: ");
    scanf("%d",&guess);
    if (guess == secret_number) {
printf("You won in %d guesses!\n\n",num_guesses);
return; // only allowed in void function if no argument given
    } else if (guess < secret_number)
printf("Too low; try again.\n");
    else
printf("Too high; try again.\n");
  }
}
```

```
Guess the secret number between 1 and 100.

A new number has been chosen.
Enter guess: You won in 1 guesses!

Play again? (Y/N)
```

- Practice: alter `choose_new_secret_number` and `read_guesses` so that `secret_number` does not have to be an external variable.

- Challenge: Can you figure out how to make the numbers repeatable so that you can repeat a game?

Review questions

1. What are the properties of *local* variables?

    - Automatic storage duration (storage is allocated when function is entered, and deallocated when it is exited)
    - Block scope (from the point of declaration to the end of the function block)

2. What's the effect of the `static` keyword on a variable type declaration?

    The variable retains its value in storage until the program is finished. This implies that it retains its value between subsequent function calls while going in and out of scope.

3. What does `typedef` do, and when and why is it important?

    - `typedef` allows you to define your own type based on a basic type, e.g. `typef int Bool` to define a `Bool` type, or `typedef float Dollar` to define a `Dollar` type of integers.
    - It is used to improve readability, maintainability, and portability.

4. Program organization has seven different categories - what are they and what is their preferred order?

    #include directives #define directives Type definitions Declarations of external variables Prototypes for functions other than main Definition of main Definitions of other functions

5. What do you know about storage and scope of *external* variables?

    They have `static` storage (they are only deleted when the program is finished), and *file scope* (accessible by all functions).

6. Order the five parts of the computer's memory organization, starting with the storage space for local variables.

(a) STACK
(b) HEAP
(c) GLOBALS
(d) CONSTANTS
(e) CODE

7. Which storage duration modes, and which scopes do you know? Give examples.

   (a) Automatic storage duration: local variables e.g. inside a function. 'Automatic' means that the computer decides how to organize the memory, which is allocated when a function is called, and deallocated when it returns.
   (b) Static storage duration: `static` variables have permanent storage and retain their values throughout the duration of the program.
   (c) Block scope: variable is visible from its declaration to the end of the enclosing function body.
   (d) File scope: variable is visible from its declaration everywhere, and available to all functions.

8. When is implicit arithmetic conversion performed?

   (a) when the operands have different types
   (b) when type on left does not match type on right of `=`
   (c) when type of function argument does not match parameter type
   (d) when type of return expression does not math return type

9. What are, and how can you handle stack overflow and underflow?

   - When the `pop` function tries to remove a value from an empty stack, you get a "stack underflow", and when the `push` function tries to add a value to a full stack, you get a "stack overflow".
   - Call functions `stack_overflow()` and `stack_underflow()`, which `return EXIT_FAILURE`, a macro defined in `stdlib.h`, a non-zero integer, typically `1`.

10. How can you generate seeded pseudorandom numbers?

- You need to `include stdlib.h` for the `srand` and `rand` functions, and `time.h` for the `time` function to set the seed with `srand`.
- The function call `rand()` will give you always the same set of random numbers.
- The seed is the current time encapsulated in the number of seconds that have elapsed since the Epoch, the start of Unix system time on Jan 1, 1970.

Exercises

- **Exercise:** show how `static` local variables work by writing function with a `static int` variable.

  Solution:

  ```
  void add(void);

  int main(void)
  {
    int result;
    add();
    add();
    add();
    return 0;
  }

  void add(void)
  {
    static int i;
    i++;
    printf("add: %d\n", i);
  }
  ```

  ```
  add: 1
  add: 2
  add: 3
  ```

pp 237

Programming Projects

1. Modify the stack example so that it stores characters instead of integers. Next, add a `main` function that asks the user to enter a series of parentheses and/or braces, then indicates whether or not they're properly nested:

   ```
   Enter parentheses and/or braces: ((){}{()})
   Parentheses/braces are nested properly
   ```

   Hint: As the program reads characters, have it `push` each left parenthesis or left brace. When it reads a right parenthesis or brace, have it `pop` the stack and check that the item popped is a matching parenthesis or brace. (If not, the parentheses/braces aren't nested properly.) When the program reads the new-line character, have it check whether the stack is empty; if so, the parentheses/braces are matched. If the stack **isn't** empty (or if `stack_underflow` is ever called), the parentheses/braces aren't matched. If `stack_overflow` is called, have the program print the message `Stack overflow` and terminate immediately.

2. Some calculators (notably those from Hewlett-Packard) use a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In this notation, operators are placed **after** their operands instead of **between** their operands. For example, 1 + 2 would be written 1 2 + in RPN, and 1 + 2 * 3 would be written 1 2 3 * +. RPN expressions can easily be evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following actions:

   (a) When an operand is encountered, `push` it onto the stack.

   (b) When an operator is encountered, `pop` its operands from the stack, perform the operation on those operands and then `push` the result onto the stack.

   Write a program that evaluates RPN expressions. The operands will be single-digit integers, The operators are `+`, `-`, `*`, `/`, and `=`. The `=` operator causes the top stack item to be displayed; afterwards, the stack is cleared and the user is prompted to enter another expression. The process continues until the user enters a character that is not an operator or operand:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

If the stack overflows, the program will display the message ‘Expression is too complex‘ and terminate. If the stack underflows (because of an expression such as 1 2 + +), the program will display the message ‘Not enough operands in expression‘ and terminate.

Hints: Incorporate the stack code into your program. Use `scanf(" %c", &ch)` to read the operators and operands.

Glossary

Term	Explanation
Stack	Memory section for local variables, used during function calls.
Heap	Memory section for dynamic allocation, managed during runtime.
Globals	Variables accessible across functions, with static storage duration.
Constants	Read-only data like string literals, stored in a dedicated memory section.
Code	Memory section for executable instructions, read-only.
Static	Keyword for local variables to retain value across function calls.
Block Scope	Visibility of a variable from its declaration to the end of a block.
File Scope	Visibility of variables/functions throughout the file post-declaration.
Implicit Conversion	Automatic type conversion in mixed-type expressions.
typedef	Creates a new type name, enhancing code readability and portability.

Summary

- Explores memory sections: stack, heap, globals, constants, code.
- Stack stores local variables with automatic storage duration.
- Heap manages dynamic memory, manually allocated and freed.
- Globals have static storage duration, accessible across functions.
- Constants are read-only data, stored in a specific memory section.
- Code section holds the program’s executable instructions.

- `static` keyword makes local variables retain value across calls.
- Scope determines where a variable is visible in the code.
- Implicit conversions occur when operands in expressions differ in type.
- Emphasizes organized program structure using external variables, prototypes, and type definitions.
- Practical examples include stack implementation and a number-guessing game.