Chapter 8: Intraprogram Communication
Topics

- Storage Types
- Global and Local Variables
- Modules and External Variables
- Typedefs: Renaming Types
- Review: Functions
- Pointers to Functions
Storage Types

• How the memory as a variable is handled during an execution is called its storage type
Storage Types

• The variable properties can be different in depending on:
  • the time the variable exists in memory
  • the location of the variable in memory
  • the location of the variable in the program structure
Storage Types

• The storage type specification comes before the data type in declaration

• `<storage type> <data type> <identifier>`
Storage Types

• Two storage types define the life-span of a variable in C
  • automatic
  • static
Automatic Storage

- The default method of data storage is **automatic storage**
- The keyword for this type of storage is **auto**
Automatic Storage

- Automatic Storage means the variable is created and memory allocated for it at the time a function is called.

- Example

  ```
  auto int IntVar=0;
  ```

- is equivalent to

  ```
  int IntVar=0;
  ```
Automatic Storage

- When a variable has automatic storage, the memory and value of that variable ends when the program leaves the code block where it was defined.
Automatic Storage

• The following function is called 4 times. What is the output?

```c
void AutoFunction()
{
    auto int counter=0;
    counter++;
    printf("Auto %d\n",counter);
}
```

• Auto 1
• Auto 1
• Auto 1
• Auto 1
Static Storage

- Another method of storing data is *static storage*
- The keyword for this type of storage is *static*
Static Storage

• Static storage means that the variable exists and retains its value for the entire run of the program.

• Example

  static int IntVar;

• or

  static int IntVar=5;
Static Storage

- Unless otherwise initialized, static variables automatically are set to 0.
- Static variables retain their value after a function ends, rather than being erased.
The following function is called 4 times. What is the output?

```c
void StaticFunction1()
{
    static int counter;
    counter++;
    printf("Static 1 %d\n", counter);
}
```
Static Storage

• The following function is called 4 times. What is the output?

```c
void StaticFunction2()
{
    static unsigned long double counter=3;
    counter=2*counter+1;
    printf("Static 2 %g\n",counter);
}
```

• Static 2 7
• Static 2 15
• Static 2 31
• Static 2 63
Storage Types

• A storage type that defines where a variable is stored is called **register storage**
Register Storage

• The keyword for this type of storage is `register`

• Example:
  ```
  register int LoopVar;
  ```
  or
  ```
  register int LoopVar=0;
  ```
Register Storage

- Register storage is supposed to move the variable into a register and allow it a compiler to generate a more efficient program.
Register Storage

• Is not guaranteed in the C language.
• Useful for variables that are used more than others, such as the loop control for a large array
Register Storage

• The following function is called once.

Void RegisterFunction()
{
    register int LoopVar;
    for(LoopVar=0;LoopVar<100000;LoopVar++)
    {
        printf("%d%%r",LoopVar/1000);
    }
    printf("\n");
}
Scope

- The **scope** of a variable is defined on where the variable is created.
- The properties of the variable is altered by where it is declared in the structure of the program.
The *scope* of a variable is where inside the structure of a program a variable can be read, used, and changed.
Variables are generally referred to having two types of scope.

- Local
- Global
Local Variables

- Local variables are declared inside a code block.
- Local variables are visible with reference to their code blocks.
Global Variables

- Global Variables are declared outside of all functions and
- Global variables are declared before they are used inside of functions
Global Variables

- A global variable is visible to all blocks of code
- A variable declared before all the blocks in a source file can be seen by all the functions of that source file.
Global Variables

- Global variables exist during the entire execution of a program.
- **Global variables initialize to 0**
Global Variables

- Global variables are quick to use.
- Global variables are easy to declare and get working quickly.
- Global variables can be seen throughout the program.

**THAT IS THE PROBLEM**

- Global variables can easily cause a side effect.
Side Effects

- The properties that make a global variable easy to use also make it easy to cause an error.
Side Effects

- A line of code in one function forgotten by a programmer can cause a fault that goes throughout the entire program due to the scope of a global variable.
Global Properties of Local Variables

• A local variable can behave as a global variable with respect to a nested code block
Global Properties of Local Variables

```c
int w=0;
void LocalScopes()
{
    {
        int x=5; int y=10;
        {
            int x = 0; int z=15;
            printf("%d,%d,%d,%d\n",w,x,y,z);
            x=3;y=6;z=9;w=81;
            printf("%d,%d,%d,%d\n",w,x,y,z);
        }
        printf("%d,%d,%d\n",w,x,y);
    }
    printf("%d,%d\n",w,x);
}
```
Modules and External Variables

- A module is a file containing source code in a program that all work together to perform a related task.
Modules and External Variables

• Each source module is compiled first, and then can be linked afterwards.
• Breaking large programs into modules allow for better organization.
Modules and External Variables

- In order to make functions work with each other in different modules, global variables may need to be used.
- Also, some functions may also have to be given extra behavior to cross between module files.
• The special keyword for having a variable go across a module boundary is `extern`.

• A global variable
  
  `extern int globalvar;`

• Would be visible across multiple files
• Typically, the original variable is declared without an extern, and is in a user defined header file, or in the C file with the main.

    int globalvar;
To block this ability to jump between modules, the variable can be declared \textit{static}.

\begin{verbatim}
static int globalvar;
\end{verbatim}
Typedef

- A typedef is a special storage type, like static or register.
- Typedef is used to associate an identifier with a known data type.
- Typedef does not allocate storage, it just renames a data type.
Typedef

- Portability: Moving code from one machine to another may require a specific type of data field. The short type on one machine may be equal to the int type on another
Typedef

- Readability: There are a limitless number of combinations of ints and floats and other values. Giving a different name is easier to recognize or remember.

typedef long big_int;
typedef struct {
    float radius;
    float center;
} CIRCLE;
Function Review

• Function Definition:
• A Function definition is the main header of the function and its code block

```c
type identifier (int param 1, int param 2)
{
    /*Code here*/
}
```
Function Review

• Function Types-The return type of a function is the function type.
• In communicating between pieces of a program, the return type can be used for communication between functions.
Function Review

- Function Declaration
- The function declaration is the prototype of a function.

```
return_type identifier(type1,type2);
```
Function Review

- Function Call
- To call a function is to use the function name with its parameter list, and then to get its return type if necessary.
Function Review

- data=function_1();
- data=function_2(paramter);
- function_3();
- function_4(paramater);
Function Pointer

• Functions are identifiers in C
• Like all such routines, it is possible to have a pointer to a function just as a pointer to a variable.
• The notation for such a pointer is

  return type (* function_pointer_identifier) (parameter list)
Function Pointers

- Function pointers can be useful
- Combining different routines to achieve efficiency
- Doing a set of commands in different orders
Function Pointer

```c
int PtrTest1(int B)
{
    int result=5;
    int LoopControl=0;
    int LoopControl2=0;
    printf("Pointer Test 1\n");
    for(LoopControl=0;LoopControl<B;LoopControl++)
    {
        for(LoopControl2=0;LoopControl2<LoopControl;LoopControl2++)
        {
            printf("*");
        }
        printf("\n");
    }
    printf("\n");
    return result;
}
```
Function Pointer

```c
int PtrTest2(int C)
{
    int result=0;
    printf("Pointer Test 2\n");
    printf("The parameter is %d\n",C);
    printf("\n");
    return result;
}
```
void FunctionPointerTest()
{
    int (*fnptr)(int)=NULL;

    fnptr=PtrTest1;
    fnptr(7);

    fnptr=PtrTest2;
    fnptr(7);
}

Function Pointer

Pointer Test 1

*  
**  
***  
****  
*****  
******

Pointer Test 2
The parameter is 7