Overview of C

Structure of a C Program

```
/* File: powertab.c
 * ----------------
 * This program generates a table comparing values
 * of the functions n^2 and 2^n.
 */

#include <stdio.h>

/*
 * Constants
 * ---------
 * LowerLimit - Starting value for the table
 * UpperLimit - Final value for the table
 */

#define LowerLimit 0
#define UpperLimit 12

/* Function prototypes */

int RaiseIntPower(int n, int k);

/* Main program */

int main()
{
   int n;

   printf("    |   2 |     N\n");
   printf("  N |  N  |    2\n");
   printf("----+-----+------\n");

   for (n = LowerLimit; n <= UpperLimit; n++){
      printf(" %2d | %3d | %4d\n", n,
             RaiseIntPower(n, 2),
             RaiseIntPower(2, n) );
   }
}
```

```
/*
 * Function: RaiseIntPower
 * This function returns n to the kth power.
 */

int RaiseIntPower(int n, int k)
{
   int i, result;

   result = 1;

   for (i = 0; i < k; i++){
      result *= n;
   }
   return (result);
}
```


Variables, values and types

A ***variable*** can be thought of as a named box, or cell, in which one or more data values are stored and may be changed by the algorithm.

The act of creating a variable is called ***declaring the variable***. For every variable that is declared it must be explicitly typed. In other words, each variable has an associated ***data type***.

Examples:

```
int age;
int test_score;
float average;
double result = 0.0;
```

An ***identifier*** is simply the algorithmic terminology for a name that we make-up to "identify" the variable. **Every** variable must be given a ***unique identifier*** so that there will be no ambiguity as to which piece of data we are referencing.

Rules for Variable Identifiers (in C)

- A sequence of letters, digits, and the special character _.
- A letter or underscore must be the 1st character of an identifier.
- C is *case-sensitive*: Apple and apple are two different identifiers.

Data Types

A data type is defined by two properties:

- a *domain*, which is a set of values that belong to that type
- a *set of operations*, which defines the behavior of that type

e.g.

Type `int` includes all integers (… -2, -1, 0, 1, 2, …) up to the limits established by the hardware of the computer.
The set of operations includes the standard arithmetic operations like addition, multiplication.

C includes several fundamental types that are defined as part of the language. These types are called ***atomic types***.

Atomic types can be grouped into 3 categories: integer, floating point and character.

- **Integer Types:**
  - **short**: 2 bytes
  - **int**: 4 bytes
  - **long:** 4 bytes
  - **unsigned:** 4 bytes
- **Floating-point Types:**
  - **float**: 4 bytes
  - **double**: 8 bytes
  - **long double:** 8 bytes
  - **signed/unsigned**

The range of values for each type depends on the particular computer's hardware.

- **Characters:**
  - **char:** 1 byte

The domain of type `char` is the set of symbols that can be displayed on the screen or typed on the keyboard: the letters, digits, punctuation symbols, spacebar, Return key, etc.
In addition to the standard characters, C allows you to write special characters in a two-character form beginning with a backslash (\)

Examples:

```
char grade;
char first, mid, last;

int age, year;
double tax_rate;


grade = 'A';
age = 20;
mid = '\0';
```

Expressions

For example:

(-b + sqrt(b * b – 4 * a * c)) / (2 * a)

Arithmetic Operators

Basic operations include the four basic operations and the modulus operator:

- **Addition (+)**
- **Subtraction (-)**
- **Multiplication (*)**
- **Division (/)**
- **Modulus (%)**

`x % y` produces the remainder when `x` is divided by y.
e.g.

```
11 % 5 = 1
20 % 3 = 2
```

⇨ **Operator Precedence:**

`x = 1 + 2 * 3;` **(What is the value of x?)**

`x = 1 + (2*3);` x is 7?
OR
`x= (1+2) * 3;` x is 9?

⇨ **Associativity: (*left to right*)**

```
10 + 3 + 7  =>  20
10 - 3 + 7  =>  14
15 / 5 * 2  =>   6
```


Assignment Operator

variable = expression

- The expression can simply be a constant or a variable:

```
int x, y;
x = 5;
y = x;
x = 6;
```

- The expression can be an arithmetic expression:

```
x = y + 1;
y = x * 2;
x = x + 10;
```

- Embedded assignments:

```
z = (x = 6) + ( y = 7);
n1 = n2 = n3 = 0;
```

- Shorthand assignments:

```
x += y;
z -= x;
y /= 10;
```

Boolean Operators

C defines three classes of operators that manipulate Boolean data:

- **relational operators**
  - Greater than >
  - Greater than or equal >=
  - Less than or equal <=
  - Less than <
  - Equal to ==
  - Not equal to !=

- **logical operators**
  - AND : &&
    (TRUE if both operands are TRUE)
    ((5 > 4) && (4 > 7)) is FALSE

  - OR : ||
    (TRUE if either or both operands are TRUE)
    ((5 >4) || (4 > 7)) is TRUE

  - NOT: !
    (TRUE if the following operand is FALSE)
    ! (4 > 7) is TRUE

- **?: operator**
  (*condition*) ? *expr1* : *expr2*;

```
max = (x>y) ? x : y;
```


Input and Output Operators

I/O Operators: allow us to communicate with the "outside world," i.e, the real world beyond the algorithm

scanf: obtains (reads) an input value

each *read* operation does 2 things:

1. obtains next value from "outside"
2. stores it in the specified variable

e.g.

```
scanf("%d", &num);
```

printf: sends out an output value

e.g.

```
printf("%d", num);
printf("Hello world!");
```

- **scanf** and **printf** may take any number of arguments of the existing types.

General format:

scanf(*format control string*, *input var list*);
printf(*format control string, output var list*);

- number of conversion characters = number of arguments.

Conversion characters for different types:

%c	char
%d	int
%f	float, double (for printf)
%lf	double (for scanf)
%Lf	long double (for scanf)

Statements

Simple Statement

expression **;**

The expression can be a function call, an assignment, or a variable followed by the ++ or – operator.

Blocks

A block is a collection of statements enclosed in curly braces:

```
{
    statement_1
    statement_ 2
    ...
    statement_n
}
```


The `if` statement

It comes in two forms:

```
if   (condition)
    statement
```

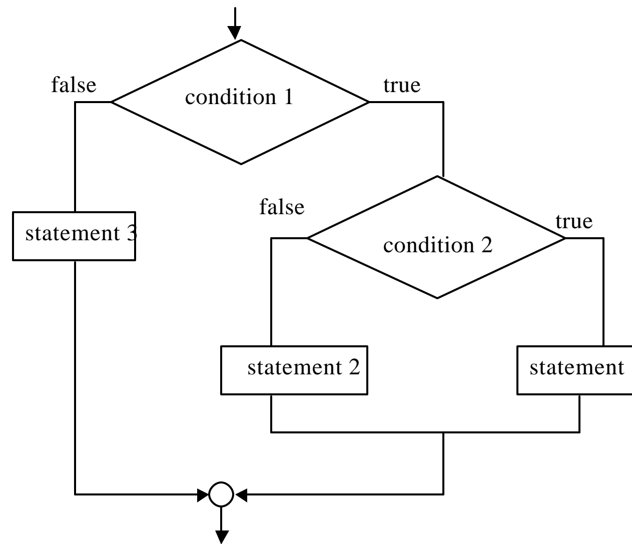
```
if   (condition)
    statement
else
    statement
```

An example program:

```
main()
{   int n;

    printf("This program labels a number as "
            "even or odd.\n");
    printf("Enter a number: ");
    n = GetInteger();
    if (n%2) == 0)
        printf("That number is even.\n");
    else
        printf("That number is odd.\n");
}
```

Nested `if` Statements



if (*condition1*)
    if (*condition 2*)
        statement 1
    else
        statement 2
else
    statement 3

The else-if Statement

if (*condition1*)
  statement
else if (*condition2*)
  statement
else if (*condition3*)
  statement
else
  statement

If Statement Exercise

Write a C program that reads in the dimensions of a room (length x width) and also reads in the size of a desk (length x width) and determines if the desk can fit in the room with each of it's sides parallel to a wall in the room. Assume the user enters positive numbers for each of the four dimensions.

```
#include <stdio.h>
int main() {

     int roomlen, roomwid;
     int desklen, deskwid;

     printf("Enter the length and width of the "
            "room.\n");
     scanf("%d%d",&roomlen, &roomwid);
     printf("Enter the length and width of the "
            "desk.\n");
     scanf("%d%d",&desklen, &deskwid);

     if ((deskwid <= roomwid) && (desklen <= roomlen))
        print("The desk will fit in the room.\n");
     else
        print("The desk will not fit in the room.\n");

     return 0;

}
```

Will this work in every situation? Why or why not?


```
#include <stdio.h>
int main()
{
  int roomlen, roomwid;
  int desklen, deskwid;
  int temp;

  // Read in room and desk dimensions.
  printf("Enter the length and width of the room.\n");
  scanf("%d%d",&roomlen, &roomwid);
  printf("Enter the length and width of the desk.\n");
  scanf("%d%d",&desklen, &deskwid);

  // Check both ways of putting the desk in the room
  // and output the result.
  if ((deskwid <= roomwid) && (desklen <= roomlen))
     print("The desk will fit in the room.\n");
  else if ((deskwid<=roomlen) && (desklen<=roomwid))
     print("The desk will fit in the room.\n");
  else
     print("The desk will not fit in the room.\n");

  return 0;

}
```

The `switch` statement

General Form:

```
switch (e) {
   case c1 :
         statements
      break;
   case c2 :
         statements
      break;

    more case clauses

   default:
         statements
      break;
}
```

Example:

```
int MonthDays (int month, int year)
{
   switch (month){
      case 9:
      case 4:
      case 6:
      case 11: return 30;
      case 2:
         return (IsLeapYear(year))? 29: 28);
      default :
         return 31;
   }
}
```


Iterative Statements

The `while` Statement

General form:

```
   while (conditional expression){
      statements
   }
```

Example:

```
/* This function computes the sum of the
 * digits in an integer.
 */
int DigitSum (int n)
{
   int sum;

   sum = 0;

   while (n > 0) {
      sum += n % 10;
      n /= 10;
   }

   return (sum);
}
```

⇨ Many programming problems do not fit easily into the standard while loop structure. The most common example of such problems are those that read in data from the user until some special value, or sentinel, is entered to signal the end of the input.

One way to solve this problem is to use the `break` statement that has the effect of immediately terminating the innermost enclosing loop:

```
while (TRUE) {
      Prompt the user and read in a value.
      if ( value == sentinel ) break;
      Process the data value.
}
```

Example:

```
/* This program add a list of numbers */
#define sentinel 0

main()
{
   int value, total =0;
   printf("This program add a list of numbers.\n");
   printf("Use %d to signal the end of list.\n",
           sentinel);
   while (TRUE) {
      printf(" ? ");
      value = GetInteger();
      if (value == sentinel) break;
      total += value;
   }
   printf("The total is %d.\n", total);
}
```


Example: Menu driven program set-up

```
int main() {

  int choice;

  while (TRUE) {

    Print out the menu.

    scanf("%d",&choice);

    if (choice == 1) {
        Execut e this option
    }
    else if (choice == 2) {
        Execute this option
    }
    ...
    else if (choice == quitting choice)
       break;
    else   {
       That's not a valid menu choice!
    }
  }
  return 0;
}
```

The `for` Statement

General form:

`for` (*initialization*; *loopContinuationTest*; *increment*) {
statements
}

which is equivalent to the `while` statement:

initialization;
`while` (*loopContinuationTest*) **{**
statements
increment;
}

Example: Finding the sum 1+3+…+99:

```
int main() {

  int val ;
  int sum = 0;

  for (val = 1; val < 100; val = val+2) {
       sum = sum + val;
  }

  printf("1+3+5+...+99=%d\n",sum);
  return 0;
}
```


The `do/while` Statement

General Form:

`do {`
statements
`} while (` *condition* **`);`**

Example 1:

```
counter = 10;
do {
    printf("%2d\n", counter );
    counter -= 1;
} while (counter >= 0);
printf("Liftoff!\n");
```

Example 2:
Write a loop to enforce the user to enter an acceptable answer of Yes or No.

```
do {
    printf("Do you want to continue? (Y/N)");
    scanf("%c",&ans);
} while (ans != 'Y' && ans != 'y' &&
         ans != 'N' && ans != 'n');
```

Nested Control Structures

Flow of control statements such as if, for, while etc. can be nested within themselves and within one another.

When the body of a loop includes another loop construct this is called a *nested loop*. In a nested loop structure the inner loop is executed from the beginning every time the body of the outer loop is executed.

Example 1:

```
value = 0;
for (i=1; i<=10; i=i+1)
   for (j=1; j<=5; j=j+1)
      value = value + 1;
```

The outer loop is executed ____ times. The inner loop is executed ____ times every time the outer loop is executed. (i.e. ____ times in total)

Example 2:

```
value = 0;
for (i=1; i<=10; i=i+1)
   for (j=1; j<=i; j=j+1)
      value = value + 1;
```

Similar to the one above. How many times the inner loop is executed?

Functions

Functions can be categorized by their return types:

- Function returning a value – A function that performs some subtask that requires returning some value to the calling function.

- Void function – A function that performs some subtask that does not require a single value to be returned to the calling function.

Function definitions and prototypes

A function definition has the following syntactic form:

result-type name **(** *parameter-list* **)**
{
    body of the function
}

Before you use a function in a C program, it is a good practice to declare it by specifying its *prototype*. A function's prototype gives us all the information needed to know how to use the function and no more.
In C, the general form for a function prototype is:

result-type name **(** *parameter-list* **);**

For example,

```
double pow(double base, double exponent);
```

No information about how the function performs its task. Nonetheless, the reader has enough information to invoke or use the function.

C Library Functions

Library: ctype.h

```
int isalpha(int c);
int isdigit(int c);
int isupper(int c);
int islower(int c);
int tolower(int c);
int toupper(int c);
```

Library math.h

```
double cos(double x);
double acos(double x);
double cosh(double x);
double exp(double x);
double ceil(double x);
double floor(double x);
double fabs(double x);
double pow(double x, double y);
double sqrt(double x);
```

Library stdlib.h

```
int rand(void);
void srand(unsigned seed);
```

General Structure of a C Program

- A program is made up of one or more functions; one of them must be **`main( ).`**
- Each function must have been defined before it is called. The best way to organize modules in a program is to list the prototypes of all functions before the main function. Their definition can be placed after the main function.

```
preprocessor directives

function prototypes

int main () {
.
.
.
}

function-1
function-2
.
.
.
function-n
```


Returning results from functions

- Functions can return values of any type.

Example:

```
int IsLeapYear(int year)
{
   return ( ((year % 4 == 0) && (year % 100 != 0))
             || (year % 400 == 0) );
}
```

This function may be called as:

```
if (IsLeapYear(2003))
   printf("29 days in February.\n");
else
   printf("28 days in February.\n");
```

`void` Functions

A void function does not return a value to the calling algorithm. We don't need to include a return statement in a void function. After the last statement in the void function is executed, the control of execution is returned to the calling point in the program.

Format of a `void` function

```
void function_name (formal parameters) {
.
.
.
}
```

Example:

```
void displayResults(int result1, int result2,
                    int result3)
{
    printf("The first result is %d\n", result1);
    printf("The second result is %d\n", result2);
    printf("The third result is %d\n", result3);
}
```

Calling the function `displayResults`:

```
displayResults(r1, r2, r3);
```


- Another common example of a void function is one that prints out a menu:

```
void menu() {

  printf("Please choose one of the following.\n");
  ....
}
```

You can call this function as follows:

```
menu();
```

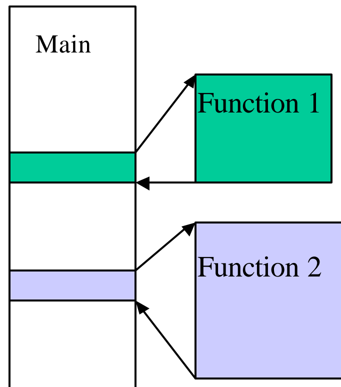
- Another situation where `void` functions might be useful is in a menu driven program where the menu choices are completely unrelated:

```
int main()
{
   int choice;

   while (1) {
     menu();
     scanf("%d", &choice);
     if (choice == 1)
        function1();
     else if (choice == 2)
        function2();
     else if (choice == 3)
        function3();
     else if (choice == 4)
        break;      //quit option is selected
     else
        printf("Invalid option. Please enter your"
               "choice again.\n");
   }
}
```

Calling a Function

- When a program encounters a function, the function is called or invoked.
- The control of execution is transferred to the body of the function.
- The function has its own memory space.
- After the function does its work, program control is returned to the calling function, where program execution continues.



- When a return statement is executed, program control is immediately passed back to the calling environment
- If an expression follows the keyword return, the value of the expression is returned to the calling environment as well.

Value and Reference Parameters

There are two ways to pass parameters to functions in many programming languages:

1. Call-by-value
2. Call-by-reference

- When an argument is passed *call-by-value*, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller.

- When an argument passed *call-by reference*, the caller actually allows the called function to modify the original variable's value.

➢ In C, all calls are call-by-value. However it is possible to simulate call by reference by using pointers. (Address operators (&) and indirection operators (*))