Tutorial on C Programing

TELE402

What Does a C Program Look like?

```
/* a simple C program (add,c)  */
#include <stdio.h>
#define  LAST   10
void Incr (int *num, int i);
int main ( )
{
      int  i,  sum =0;
      for ( i =1; i<=LAST; i++ ) {
            Incr (&sum, i ); /* add i to sum */
      }
      printf (“sum = %d\n”, sum);

      return 0;
}

void Incr (int *num, int i) {

    *num = *num +i;

}
```

Preprocessing

Function declaration

Variable declaration

I/O functions

the “**main**” function, an executable program must have one

Function definition

Data Types

- Four basic data types
  - char -- a single byte, holding one character
  - int -- an integer, natural size of integers on host machine
  - float -- single-precision floating point
  - double -- double-precision floating point
- Note that there is no *boolean* type in standard C
- Qualifiers applicable to basic types
  - short -- 16 bits (short int)
  - long -- 32 bits (long int)
  - signed or unsigned -- (signed char -128 to 127)

Variables

- Variable Names
  - Are made up of letters and digits
  - The first character must be a letter
  - "_" as a letter to improve readability (e.g num_packets)
  - Upper and lower case letters are distinct (**case-sensitive**)
- All variables must be **declared before use**
  - int step, count; /* declare multiple vars of the same type */
  - char esc='\\'; /* initialization in declaration */
  - Use *const qualifier* to declare variables with no changeable values

    const char msg[] = "warning: ";
  - Variables must be declared at the beginning of a block (C89).

Operators

- Arithmetic operators
  - \+ - * / %
- Relational and logical operators
  - > >= < <=
  - == !=
  - && ||
- Increment and Decrement operators
  - ++ -- (note that *x*++ and ++*x* *are different*)
- Bitwise operators
  - & bitwise AND
  - | bitwise OR
  - ^ bitwise exclusive OR
  - << left shift
  - >> right shift
  - ~ one's complement
- Assignment operator (=)

Control Flow

- If-Else
  if (expression) statement1; *else* statement2;
- Else-if: combined with If-Else
- Switch:

```
switch (expression) {
        case const-expr: statements;
        case const-expr: statements;
        default:  statements;
}
```

- While

```
while (expression)
        statements;
```

- For

```
for (expr1; expr2; expr3)
        statements;
```

- Do-while

```
do
        statements
while (expression);
```

Functions (1)

- Not a good idea to implement everything in the main function
- Function is the main abstraction which represents an operation to be performed more than once.
- Function definition

Return type

Argument declarations

```
int  cal_sum ( int  a, int  b)
{
      int sum=0;
      sum = a + b;
      return sum;
}
```

Function name

Declarations and statements

Functions (2)

- Every C program must have a function named "**main**". This function is called when the program runs.
- There are many standard functions defined as part of the C language. These functions are stored in **libraries**
  - <stdio.h> defines the standard input/output functions.
  - <string.h> defines the standard functions for manipulating strings.
  - …

Functions (3)

- Two approaches to define and declare functions
  - Define it first, and then use it
  - Forward declaration: declare it before giving a complete definition

```
 /* a simple C program (add,c)  */

#include <stdio.h>
#define  LAST   10

void Incr (int *num, int i);

int main ( )
{
        int  i,  sum =0;
        for ( i =1; i<=LAST; i++ ) {
            Incr (&sum, i ); /* add i to sum */
        }
        printf (“sum = %d\n”, sum);

        return 0;
}

void Incr (int *num, int i) {

      *num = *num +i;

}
```

Functions (4)

- Scope of the functions or variables is the part of the program within which they can be used
  - **automatic** variables declared at the beginning of a function: the scope is the function in which they are declared
  - local variables of the same name in different functions are unrelated
  - **external** variables or functions last from the point they are declared.
- For **external** variables: **declaration ≠ definition**
  - declaration announces the properties (primarily the type)
  - definition also causes storage to be set aside
  - **extern** declaration is mandatory for external variable to be referred to before definition, or is defined in a different source file.

```
int sp;
double val[MAXVAL];
```

```
extern int sp;
extern double double val[];
```

Functions (5)

- **Automatic** variables appear/disappear automatically when a function is called.
- "Call-by-value" parameter passing
  - The values of function parameters are automatic
  - They are copy of the values of the variables that were passed to the function

```
#include <stdio.h>
int sum( int a, int b);
// function prototype at the start of the file

int main()
{
    int x = 4, y = 5;
   int total = sum( x, y );   // function call
   printf( "The sum of 4 and 5 is %d", total);
}

int sum( int a, int b )   // call-by-value
{
   return( a + b );
}
```

Functions (6)

- Static variables

```
#include <stdio.h>
void func() {
  static int x = 0;
  printf("%d\n", x);
  x = x + 1;
}
int main(int argc, char * const argv[]) {
        func();
        func();
        func();
        return 0;
}
```

  - Are allocated statically with lifetime across the entire run of the program
  - Will be **initialized only once**
  - Apply to both external and internal variables
- Register variables
  - Are variables that will be heavily used
  - Are to be placed in machine registers, resulting in faster programs
  - e.g., `register int x;`

Pointers (1)

- Memory and Addresses
  - Most modern computers are *byte-addressable*

```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'z';
```

(the addresses are at 2100, 2104, etc.)

5	10	12.5	9.8	z
2100	2104	2108	2112	2116

Pointers (2)

- A pointer is a variable that contains the address of a variable

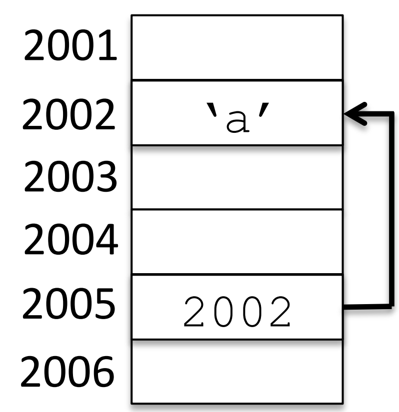
```
char x;     // data variable
char *xaddr;        // pointer variable
```

- Address-of operator &

```
xaddr= &x;
```

- Dereference (indirection) operation *

```
*xaddr= 'a';
```



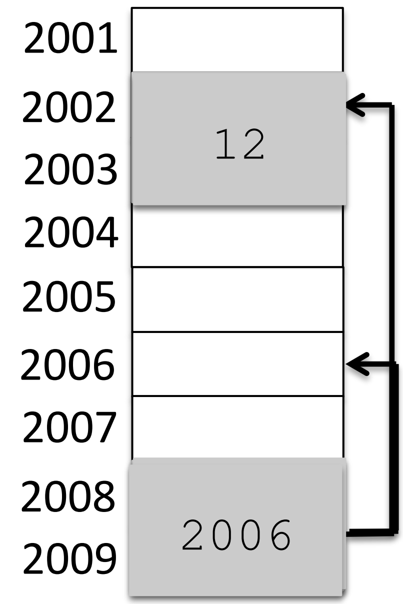
Pointers (3)

```
int x = 1, y = 8;
int *ip, *iq;

ip = &x;
y = *ip;
*ip = 6;
iq = ip;
```

Pointers (4)

- Pointer arithmetic

```
int i=10;
int *ip;

ip = &i;
*ip = *ip + 2;
ip = ip + 2;
```



The addition of 2 to a pointer increases the value of the address it contains by the size of two **objects of its type.**

Pointers (5)

- Why use pointers?

```
#include <stdio.h>

void swap( int, int );

main()
{
  int num1 = 5, num2 = 10;
  swap( num1, num2 );
  printf(“num1 = %d and num2 = %d\n”, num1, num2);
}

void swap(int n1, int n2)
  int temp;

  temp = n1;
  n1 = n2;
  n2 = temp;
}
```

What is the problem?

Pointers (6)

- Using of pointers solves the problem

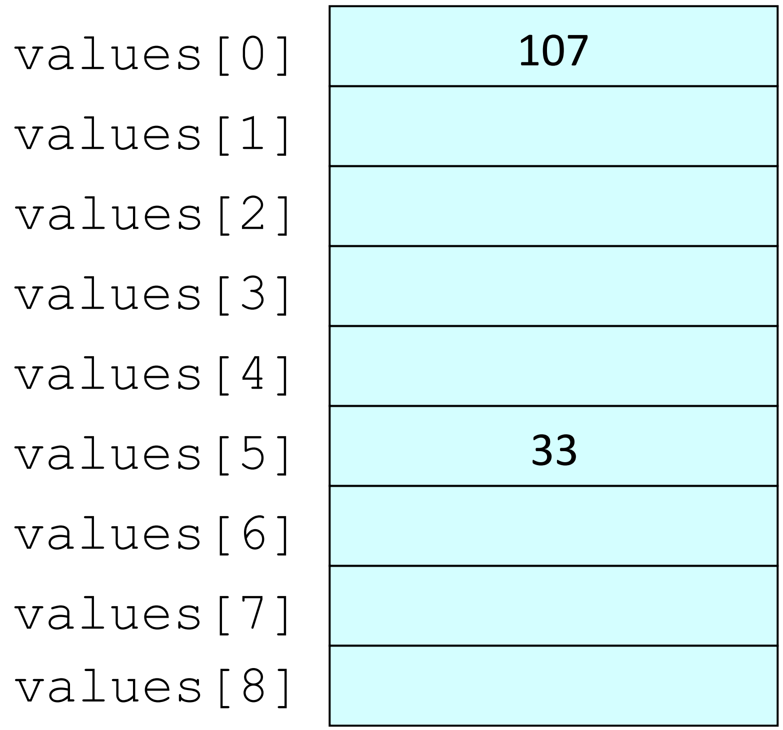
```
#include <stdio.h>

void swap( int*, int* );

main()
{
  int num1 = 5, num2 = 10;
  swap( &num1, &num2 );
  printf("num1 = %d and num2 = %d\n", num1, num2);
}

void swap( int *n1, int *n2 ) // passed by 'reference'
{
  int temp;

  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

Array (1)

- An array is a collection of ordered data items, all belonging to the same type.
- For example, declare:

```
int values[9];
```

```
values[0] = 107;
values[5] = 33;
```



- The first array index is always 0.

```
values[9]= 10;
```

- C performs no array boundary checks.

This means that you can overrun the end of an array, without the compiler complaining.

This is the cause of many runtime errors!!!

Array (2)

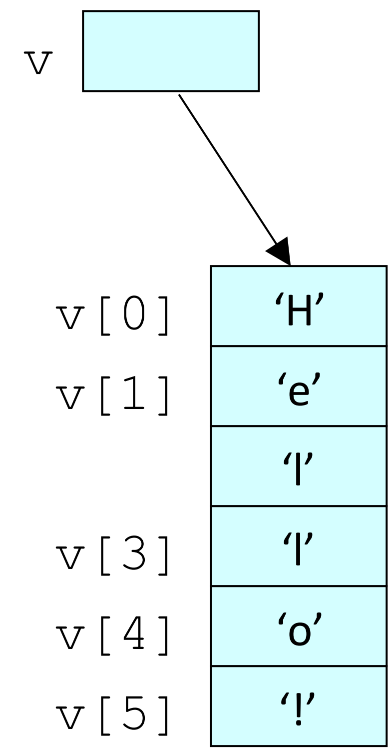
- An example

```
#include <stdio.h>

void main(void)
{
  int myary[12];          // 12 cells
  int index, sum = 0;

   // Initialize array before use
  for (index = 0; index < 12; index++)
   {
     myary[index] = index;
   }

  for (index = 0; index < 12; index++)
  {
    sum += myary[index];  // sum array elements
  }
  printf( “The sum is %d”, sum );
}
```

Array (3)

- Strong relationship between pointers and arrays
  - The name of an array is like a pointer to the first element of the array
  - In most cases array names are *converted* to pointers

```
char v[6];
char *p;

p = &v[0]; equivalent to p = v;


v[i]  equivalent to  *(v+i);
```



  - One difference – an array name is not a variable.

```
v++;    v=p;
```

Array (4)

- Multidimensional arrays

```
int points[3][4];
points[1][2]=3;
```

- What is `points[1,2]`?
- Array initialization

```
int counters[5] = { 0, 0, 0, 0, 4 };
char letters[] = { 'a', 'b', 'c', 'd', 'e' };
```

Note that, in the second example, you don't have to specify the array size. The compiler will figure it out.

Array (5)

- Character strings
  - A *string constant* is an array of characters.
  - An array is terminated by with the *null* character `'\0'` so that the program can find the end.

What is the length of a string?

What is the length of a string array?

- We can write

```
   char  word[]= {'H','e','l','l','O','!'};
OR
   char  word[]= {'H','e','l','l','O','!','\0'};
OR
   char  word[]= "Hello!";
```

Array (6)

- When an array (or character string) is passed to a function, it is **passed by reference**.

```
int main()
{
  void concat(char result[],char str1[],char str2[];
  char s1[] = { “Hoo “};
  char s2[] = { “Ha!” };
  char s3[20];
  concat(s3, s1, s2);
  printf(“%s\n”, s3);
}
```

Pointers in C vs. References in Java

- **References might be implemented by storing the address.**
  - They may be using an additional layer of indirection to enable easier garbage collection. But in the end it will (almost always) boil down to (C-style) pointers being involved in the implementation of (Java-style) references.
- **You can't do pointer arithmetic with references.**
  - The most important difference between a pointer in C and a reference in Java
  - In Java, you can't just ask it to point to "the thing after the original thing".
- **References are strongly typed.**
  - In C you can have an int* and cast it to a char* and just re-interpret the memory at that location.
  - That re-interpretation doesn't work in Java: you can only interpret the object at the other end of the reference as something that it already is (i.e. you can cast a Object reference to String reference only if the object pointed to is actually a String).
- **Those differences make C pointers more powerful, but also more dangerous.**
  - Both of those possibilities (pointer arithmetic and re-interpreting the values being pointed to) add flexibility to C and are the source of some of the power of the language.
  - But it's pretty easy to use them incorrectly.

Structs (1)

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- Structs are like records (in Pascal) or classes in Java (without methods, though).

```
#include <stdio.h>
struct birthday
{
  int month;
  int day;
  int year;
}; // note semicolon

int main() {
  struct birthday bd;
  struct birthday *pbd = &bd;
  bd.day=1; bd.month=1; bd.year=1977;
  printf("My birthday is %d/%d/%d\n", bd.day, bd.month, bd.year);
  printf("My birthday is %d/%d/%d", pbd->day, pbd->month, pbd->year);
}
```

Structs (2)

- Embedded structure

```
struct person
{
  char name[80];   // elements can be different types
  int age;
  float height;
  struct      // embedded struct
  {
    int month;
    int day;
    int year;
  } birth;
};

struct person me;
struct person class[60];

me.birth.year=1977;
class[0].name="Jack";
class[0].birth.year = 1971;
. . .
```

Structs (3)

- Array of structure

```
struct birthday bdarr[10];

bdarr[0].date =1; bdarr[0].month = 1;
```

- Legal operations on a structure include
  - Copy or assign to it as a unit
  - Take its address with &
  - Access its members

- **Not that structures can not be compared**

- Three ways to pass a structure to a function
  - Pass the components separately
  - Pass the entire structures(passed by value, not reference)
  - Pass a pointer to it

Type Conversion (1)

- When an operator has operands of different types, they need to be converted to a common type according to a small number of rules.

```
int i;
float d,f;
d = f + i;
i = d + f
```

- Some conversions are ‘safe’ and some are ‘unsafe’.
- Two types of conversions
  - Implicit (automatic) conversion
  - Explicit (forced) conversion – casting operation

Type Conversion (2)

- Implicit (automatic) conversions
  - Assigning a value to an object converts the value to the type of that object.

```
void ff( int );
int val = 3.14159;  // converts to int 3
ff( 3.14159 );// converts to int 3
```

  - The widest data type in an arithmetic expression is the target conversion type:

```
val + 3.14159; // → double
```

(but `val` is still an `int`)

Type Conversion (3)

- Explicit (forced) conversions - casting
- There are two ways to request a cast:
  - `type (expr)`
  - `(type) expr`
- What happens when we do this?
  - double (int (3.14159));
- Some conversions are safe on some machines, but not on others (it depends on the word size of the machine), because an `int` is usually the same size as either a `short` or a `long`, but not both.

Type Conversion (4)

- Pointer conversions

```
int ival;
int *pi = 0;
char *pc = 0;
void *pv; // can convert others to this,
     // but a void* pointer cannot
     // be dereferenced directly.
pv = pi;            // ok
pc = pv;            // ok
*pc = *pv;             // error
```

Memory Management (1)

- The C programming language manages memory **statically**, **automatically**, or **dynamically**.
  - **Static** variables are allocated in the **main memory**, usually along with the executable code of the program.
  - **Automatic** variables are allocated on the **stack**, and come and go as functions are called and returned.

The above two approaches are not adequate for all situations!

Automatic-allocated data cannot persist across multiple function calls.

Static data persists for the life of the program whether it is needed or not.

  - **Dynamic memory allocation** in which memory is more explicitly and flexibly managed, typically, by allocating it from the **heap**

Memory Management (2)

- The C dynamic memory allocation functions are defined in `<stdlib.h>` header.
- Functions
  - void* malloc(size_t size); //Allocates size bytes of uninitialized storage.
  - void* calloc(size_t num, size_t size); //Allocates memory for an array of num objects of size size and zero-initializes it
  - void *realloc(void *ptr, size_t new_size);//Reallocates the given area of memory. It must be previously allocated by malloc(), calloc(), or realloc() and not yet freed.
  - void free(void* ptr); //Deallocates the space previously allocated

Memory Management (3)

- Suppose you want to allocate enough memory to store 1,000 integers. You can

```
int *intptr
intptr = (int *) malloc (1000 * sizeof (int));
```

The cast

Standard Input and Output

- If you want to use the standard input/output library, you must have `#include <stdio.h>` before the first usage.
- The simplest function reads one character from the **standard input** (usually the keyboard):

```
int getchar( void );
```

  (It returns the next input character each time it is called.)

- The function to put the character c on the on the **standard output**, which is usually the screen.

```
int putchar( int c )
```

Formatted Input/Output (1)

- Formatted Output – Printf

  int printf (char *format, $\texttt{arg}_1$, $\texttt{arg}_2$, …)

  The format string contains two types of objects: ordinary characters and special conversion specifications. Each conversion begins with a ‘%’.

```
printf(“num1 = %d and num2 = %d\n”, num1, num2);
```

Character	Type
d, i	int; decimal
o	int; unsigned octal
x	int; unsigned hexadecimal
u	Int; unsigned decimal
c	Int; single character
s	Char *; string
f	Double;

Character	Type
\b	backspace
\n	Return and new line
\t	tab
\v	vertical tab
\\	\
\’ ’	‘

Formatted Input/Output (2)

- Formatted Input – Scanf

```
int scanf( char *format, arg1, arg2, … )
e.g. int day, year;
     char monthname[20];
     scanf("%d %s % d", &day, monthname,&year)
```

- The format string contains conversion specifications
  - Blanks or tabs are ingored
  - Ordinary characters (not %) are expected to match the next non-white space character of the input stream
  - The input arguments **must be a pointer** indicating where the converted input is supposed to be stored

File Operation (1)

- File pointer – points to a structure contains information about the file
  - The location of the buffer
  - The current character position
  - Being read or written
  - Whether errors or end of file occurs
- You don’t need to know the details, just declare a pointer with type of **FILE**

```
FILE  *fp;
```

File Operation (2)

- Basic functions for file operation
  - Open a file: `FILE *fopen(char *name, char *mode);`
    - `"r" – read`
    - `"w" – write`
    - `"a" – append`
    - `"b" – binary file` (UNIX does not distinguish txt and binary file)

```
fp = fopen("t.txt", "r");
```

  - Read data from an opened file
    - `size_t fread( void *buffer, size_t size, size_t count, FILE *stream );`

```
size = fread(buffer,1024,2,fp);
```

  - Write data into an opened file
    - `int fwrite( const void *buffer, size_t size, size_t count, FILE *stream );`

```
size = fwrite(buffer,512,2,fp);
```

  - Close an opened file

```
fclose(fp);
```

Preprocessing (1)

- The “definition” capability of C provides techniques for writing programs that are more portable, more readable and easier to modify reliably.
  - Include standard header files
  - Define Constants
  - Define Types
  - Define Macro Functions
  - Undefining
  - Conditional compilation
  - Conditional debugging

Preprocessing (2)

- Define Constants

```
#define BUFSIZE  512
#define NUMBUFS  10
```

- Define Types

Note: with “;” at the end

```
typedef unsigned short  ushort;

typedef int bool;
#define TRUE  1
#define FALSE 0
```

Preprocessing (4)

- Macro functions

```
if (x<y)
  return y;
else
  return x;
```

```
#define MAX(x,y) (((x)<(y))? (y):(x))
```

Preprocessing (5)

- Undefining

```
#define NUMBUFS  5
char b1[NUMBUFS][BUFSIZE];

#undef NUMBUFS
#define NUMBUFS  7
char b2[NUMBUFS][BUFSIZE];
```

Preprocessing (6)

- Conditional compilation

```
#ifdef  USHORT
  typedef unsigned short ushort;
#else
  typedef unsigned ushort; /*assumes 16-bit machine */
#endif
```

- Conditional debugging

```
 #ifdef  DEBUG
#define asserts(cond, str)\
 {if (!(cond) fprintf(stderr,”Assertion ‘%s’ failed\n”,
  str);}
#else
  #define asserts(cond, str)
#endif
```

Standard Library (1)

- Provides declarations of defined functions, types and macro definitions
  - <stdio.h> /*input and output */
  - <stdlib.h> /* utility functions */
  - <string.h> /* string functions */
  - <math.h> /* mathematical functions */
  - <time.h> /* date and time functions */
  - <assert.h> /* diagnostics */
  - …

Standard Library (2)

- <string.h>
  - char *strcpy(s, ct)
  - char *strncpy(s, ct, n)
  - char *strcat(s, ct)
  - int strcmp(s, ct)
  - int strncmp(s, ct)
  - size_t strlen(s)
  - …

C vs. Java

- **C is similar to Java**
  - C and Java have some similarities:
  - Statements continue until a semicolon (";") character
  - primitive data types (except C has no boolean)
  - if, while, for, and switch statements are very similar to Java
  - C functions resemble Java static methods
- **C differs from Java**
  - C precedes Java, and isn't object-oriented:
  - C doesn't have classes (C structs are a long way off), and no exception mechanism.
  - program structure differs (see below)
  - C has pointers (addresses) rather than references. These are more easily abused!
  - C has no booleans: 0 == false, !0 == true.
  - I/O is different
  - A simple (single-file) C program is like a Java program where all methods are static