Lab Session 2 (Using C)

Data Types

The Importance of Data Types

- Variable Declarations:
  - Reserve an appropriate amount of memory.
  - Ensure correct operations on variables.
  - eg: int a; printf(“%f”,a);

- Expressions: (res=a+b or ch==‘Y’)
  - Combinations of constants, variables, and function calls.
  - Have value and type.

Fundamental data types

Fundamental data types: Long Form		
`char`	`signed char`	`unsigned char`
`float`	`signed short int`	`unsigned short int`
`double`	`signed int`	`unsigned int`
`long double`	`signed long int`	`unsigned long int`

- Abbreviations:
  - `signed int` ⇔ `int`
  - `short int` ⇔ `short`
  - `long int` ⇔ `long`
  - `unsigned int` ⇔ `unsigned`

Fundamental data types

Fundamental data types: Short Form		
`char`	`signed char`	`unsigned char`
`float`	`short`	`unsigned short`
`double`	`int`	`unsigned`
`long double`	`long`	`unsigned long`

Fundamental data types grouped by functionality			
Integral Types	`char`	`signed char`	`unsigned char`
	`short`	`int`	`long`
	`unsigned short`	`unsigned`	`unsigned long`
Floating Types	`float`	`double`	`long double`
Arithmetic Types	***Integral types + Floating types***		

printf(), fprintf(), sprintf() Conversion

Conversion characters		
`char %c`	`signed char %d`	`unsigned char %u`
`float %f`	`short int %hd`	`unsigned short %hu`
`double %f`	`int %d`	`unsigned %u`
`long double %Lf`	`long %ld`	`unsigned long %lu`

- Different integer base: `%x, %X, %o`
- Optional flags:
  - `+, -, 0, '`
  - `e.g. "%+d %-7d %07d %'d"`
- Floating-point formatting:
  - `e.g. printf( "%+-9.3f %+08.2f", 3.14, 2.78 );`
  - `e.g. fprintf( file, "%*.*f", 9, 3, 3.14159 );`

scanf(), fscanf(), sscanf() Conversion

Conversion characters		
`char` `%c`	`signed char`	`unsigned char`
`float` `%f`	`short int` `%hd`	`unsigned short` `%hu`
`double` `%lf`	`int` `%d`	`unsigned` `%u`
`long double` `%Lf`	`long` `%ld`	`unsigned long` `%lu`

- Different integer base: **`%x, %X, %o`**

- **`signed char`** should be treated as "signed byte" (8-bit)
- **`unsigned char`** should be treated as "unsigned byte" (8-bit)

- **`scanf()`** of **`signed char/ unsigned`** char should be done through **`int/ unsigned`**

1. The Data Type char

- **`char`**
  - One of the fundamental data types in C.
  - Representing a single character.
  - Enclosed inside **`' '`**.
  - Examples are **`'A'`**, **`'d'`**, **`'9'`**, **`'+'`**, **`'='`**, ...

- Internal Representation of a char
  - A char can be thought of as an integer value (compatible with BUT **not the `int` data type**).
  - For example,
    - **`'A'`** is stored as integer value 65.
    - **`'+'`** is stored as integer value 43.

The 7-bit ASCII Table

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
10	NL	VT	NP	CR	SO	SI	DLE	DC1	DC2	DC3
20	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
30	RS	US	SP	!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	I	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL		

Non-printable / *hard-to-print* / *hard-to-type* characters		
Name of character	**Written in C**	**ASCII Value**
alert	`\a`	7
backslash	`\\`	92
backspace	`\b`	8
carriage return	`\r`	13
double quote	`\"`	34
form feed	`\f`	12
horizontal tab	`\t`	9
newline	`\n`	10
null character	`\0`	0
single quote	`\'`	39
vertical tab	`\v`	11

- The Back-Slash was chosen as the *Escape character* (`\`)
- The above is a list of Escape sequences (e.g. `\n`)

Some Examples

`printf ("%c", 'a');`	**a**
`printf ("%d", 'a');`	**97**
`printf ("%c", 97);`	**a**
`printf ("%d", 'a' * 2 - 7);`	***187***
`printf ("H\tello");`	**H ello**
`printf ("\07"); // in octal!`	***beep***
`printf ("\"Hello!\"");`	**"Hello!"**

2. The Data Type **`int`**

- …, –3, –2, –1, 0, +1, +2, +3, …
- Computers can store only a finite portion of integral numbers.
- I.e., upper and lower limits exist.
- Why?
- An **`int`** is stored in 2 bytes, 4 bytes, or even larger (locker sizes).

2-byte **`int`**

- Two bytes , 16 bits
- 2^{16} distinct values can be stored.
- Half for negative integers, and half for non-negative integers.

$-(2^{15}), -(2^{15}-1), \ldots, -2, -1, 0, 1, 2, \ldots, 2^{15}-1$

- That is,

$-32768, -32767, \ldots, -2, -1, 0, 1, 2, \ldots, 32767$

4-byte `int`

- Four bytes , 32 bits
- 2^{32} distinct values can be stored.
- Half for negative integers, and half for non-negative integers.

$-(2^{31}), -(2^{31}-1), \ldots, -2, -1, 0, 1, 2, \ldots, 2^{31}-1$

- That is,

–2147483648, …, –2, –1, 0, 1, 2, …, 2147483647

<u>Exercise</u>
Find out the range of values that can be stored if a certain computer uses 8 bytes to store one single integer value.

Figuring out the size of an integer

`sizeof( int )`

- Can be used to find out the number of bytes used for storing an **`int`**, ***on a particular computing platform***.
- Example below.

```c
#include <stdio.h>

int main(void)
{
    printf("size of int   = %d\n", sizeof( int ) );
    return (0);
}
```

```
size of int   = 4
```

The Overflow Problem

```c
#include <stdio.h>
#define BIG 2000000000

int main(void)
{
     int a, b=BIG, c=BIG;

     printf("Size of int = %d bits\n", sizeof(int) * 8);

     a = b + c;
     printf ("a=%d, b=%d, c=%d\n", a, b, c);
     return(0);
}
```

- `(b + c)` > $(2^{31} – 1) \approx$ 4 billions
- An 32-bit integer variable cannot hold this large value.
- Variable `a` ***overflows***.

How to solve the overflow problem?

```
Size of int = 32 bits
a=-294967296, b=2000000000, c=2000000000
```

`int` variation – `short int`

- May take less number of bytes than **`int`**.
- May thus store a smaller integer value.
- For example,
  - Suppose your computer uses 4-byte **`int`**, ranging from –2147483648 to +2147483647.
  - A **`short int`** *may occupy* 2 bytes, ranging from –32768 to +32767.
  - Perhaps? May be?

`int` variation – `long int`

- May take more number of bytes than **`int`**.
- May thus store a larger integer value.
- For example,
  - Suppose your computer uses 4-byte **`int`**, ranging from –2147483648 to +2147483647.
  - A **`long int`** *may occupy* 8 bytes, ranging from -2^{63} to $2^{63}-1$.
  - Perhaps? May be?

`int` variation – `unsigned int`

- **`unsigned int`** takes same number of bytes as **`int`**.
- But stores non-negative integers ***only***.
- Range: 0 → ($2^{\textit{number of bits}}$ – 1)
- 2-byte **`unsigned int`**:
  - 0 to $2^{16}–1$
  - 0 to 65535
- 4-byte **`unsigned int`**:
  - 0 to $2^{32}–1$
  - 0 to 4294967295

`int` variation – `unsigned int`

- Note that combinations of **`short`** and **`long`** with **`unsigned`** are possible, leading to:
  - **`unsigned short int`**
  - **`unsigned long int`**

<u>Exercise</u>

- Find out the range of values that can be represented for
  - 2-byte **`unsigned short int`**
  - 8-byte **`unsigned long int`**

```c
#include <stdio.h>
#define BIG 2000000000

int main(void)
{
    unsigned int a, b=BIG, c=BIG;

    printf("Size of unsigned int = %d bits\n",
            sizeof(unsigned int) * 8);
    a = b + c;
    printf ("a=%u, b=%u, c=%u\n", a, b, c);
    return(0);
}
```

- Note the use of **`unsigned`** in declaring variables **`a, b, c`**.
- Note the use of **`%u`** in the **`printf`**.
- Overflow is avoided. Why?

```
Size of unsigned int = 32 bits
a=4000000000, b=2000000000, c=2000000000
```

3. The Floating Types

- What is a Floating-Point value?
  - Store real values such as 0.1, –2.4, 3.14159
  - Exponential notation:
  - **1.234567e5** = **1.234567** $\times 10^{5}$
    = 123456.7
  - **1.234567e-3** = **1.234567** $\times 10^{-3}$
    = 0.001234567

Precision and Range

- Precision – number of significant decimal places.
- Range – largest and smallest values that can be stored.
- Example, **`float`** ***on a particular machine***:

  Precision: 6 significant figures

  Range: 10^{-38} to 10^{+38}

$$0.d_1d_2d_3d_4d_5d_6 \times 10^n, \ -38 \le n \le +38$$

`float` variation – `double`

- Example, **`double`** on a particular machine:

  Precision: 15 significant figures

  Range: 10^{-308} to 10^{+308}

  $0.d_1d_2d_3d_4\ldots d_{14}d_{15} \times 10^n$, $-308 \leq n \leq +308$

- Example, assuming precision as 15 significant figures

  x = 123.45123451234544444;

  results in

  $0.123451234512345 \times 10^3$

Important

- Not all real numbers are representable.
- Floating-point arithmetic operations need not be exact.
- For very large computations, rounding errors may accumulate and become significant.

4. The **`sizeof`** Operator

- Find the number of bytes needed to store a piece of data.

- General form,

```
sizeof( data_type_or_name )
```

- **`data_type_or_name`** can be
  - a valid data type (such as **`int`**, **`float`**) or,
  - a variable name that has already been declared.

- E.g.

```
sizeof( long double )
sizeof( i )
```

```c
#include <stdio.h>

int main(void)
{
  printf("\n");
  printf("Here are the sizes of some fundamental types:\n\n");
  printf("       char:%3d byte \n", sizeof(char));
  printf("      short:%3d bytes\n", sizeof(short));
  printf("        int:%3d bytes\n", sizeof(int));
  printf("       long:%3d bytes\n", sizeof(long));
  printf("   unsigned:%3d bytes\n", sizeof(unsigned));
  printf("      float:%3d bytes\n", sizeof(float));
  printf("     double:%3d bytes\n", sizeof(double));
  printf("long double:%3d bytes\n", sizeof(long double));
  printf("\n");

  return (0);
}
```

```
Here are the sizes of some fundamental types:

        char:  1 byte
       short:  2 bytes
         int:  4 bytes
        long:  4 bytes
    unsigned:  4 bytes
       float:  4 bytes
      double:  8 bytes
long double: 16 bytes
```

Note: Execution results vary with different machines and compilers.

5. Type Casting

- Examples,

```
double   US_rate = 7.78932;
unsigned exchange;
long     approximate;

exchange = (unsigned) (US_rate * 109700);
approximate = (long) US_rate;
```

- Examples,

```
printf("%f\n", 14   / 3);          /* incorrect */
printf("%f\n", 14.0 / 3);          /* 4.666667 */
printf("%f\n", (float) 14 / 3);    /* 4.666667 */
```

Default Type in C

- **Assumed type** of integer constants in a C program is *normally* **`int`**.

  e.g. **`a = 3 + 109700;`** // both **`3`** and **`109700`** are **`int`**

- **Assumed type** of real number constants in a C program is **`double`**.

  e.g. **`d = 2.0;`** // the number **`2.0`** is a **`double`**

Suffixes

- A suffix can be appended to a numerical constant to specify its type explicitly.

Suffix	Type	Example
`u` or **`U`**	**`unsigned`**	**`37U`**
`l` or **`L`**	**`long`**	**`37L`**
`ul` or **`UL`**	**`unsigned long`**	**`37UL`**
`f` or **`F`**	**`float`**	**`3.7F`**
`l` or **`L`**	**`long double`**	**`3.7L`**

Data Type Definition: **`typedef`**

What is **`typedef`**?

- Allows programmers to give a new name (*alias*) to an existing type.
- Usage:

```
typedef    ExistingTypeName    NewName;
```

- Example:

```
typedef    int    color_t;
```

- Identifier **`color_t`** is now ***synonymous*** with **`int`**.

Data Type Definition: **typedef**

Program type_1.c

```c
#include <stdio.h>

typedef   int   color_t;        // a "new" type name

int main()
{
  color_t   red, blue, green; // declare 3 variables

  typedef   double   GPA;      // a "new" type name
  GPA  michael = 2.89;         // declare a variable
  red   = 15;
  green = 27;
  blue  = 29;
  return 0;
}
```

`color_t` is a Global type name.
`GPA` is a Local type name.

Advanced Usage

- Array Type Definition

```
typedef   int  IntArray[100];
IntArray  mark_sheet1;  // variable declaration
IntArray  mark_sheet2;  // variable declaration
// TWO array variables of 100 int
```

- Enumeration Type – Another User-Defined Type

```
enum fruit  {apple, orange, banana};
typedef enum fruit  FRUIT_TYPE;
FRUIT_TYPE  myFavourite;   // var. decl.
myFavourite = apple;       // assign
if (myFavourite != orange) // compare
```

What is an *Array*

- An array is a fixed-size, sequenced collection of storage elements of the same data type.
  - One single name (identifier).
  - Indexible (access with an index).
  - Stored contiguously.

1. One-Dimensional (Simple) Array

- Name of the array is **`grade`**.
- All storage elements share the same name, **`grade`**.
- All storage elements are of the *same type*.
  - **`int`** in this particular example
- The integral values in [ ] are *subscripts/ indexes*.
  - indicating the position of an element

grade[0]	88
grade[1]	78
grade[2]	93
grade[3]	92
grade[4]	80
…	…
…	…
grade[97]	65
grade[98]	97
grade[99]	83

1. One-Dimensional (Simple) Array

- The array **`grade`** can hold 100 *distinct* integer values simultaneously.
- The array elements are referred to as **`grade[0], grade[1], grade[2], ..., grade[99]`**.
- Note carefully array subscript *starts with 0* (ZERO) in C.
  - Thus the last available element is **`grade[100 – 1]`**.

1. One-Dimensional (Simple) Array

Program array_2.c

```
#include <stdio.h>
int main(void) {
  double list[4], sum = 0;
  int i;
  for (i = 0; i < 4; i++) {
    printf("number = ? ");
    scanf("%lf", &list[i]);
  }
  for (i = 90; i < 94; i++)
    sum += list[i - 90];
  printf("sum = %f\n", sum);
  return 0;
}
```

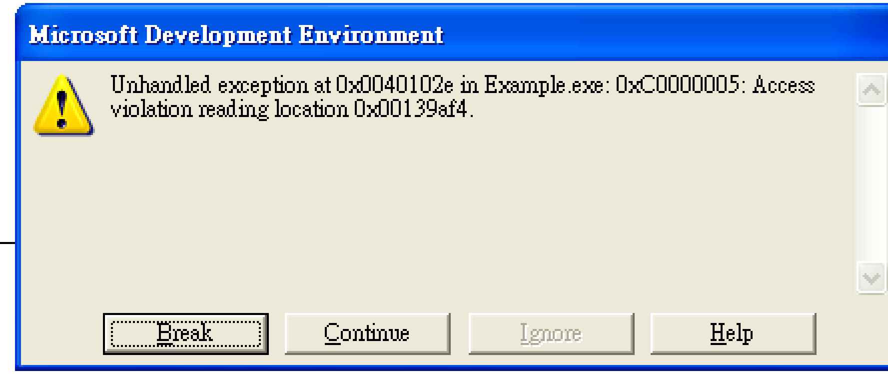
We usually use loops to process the elements in arrays.

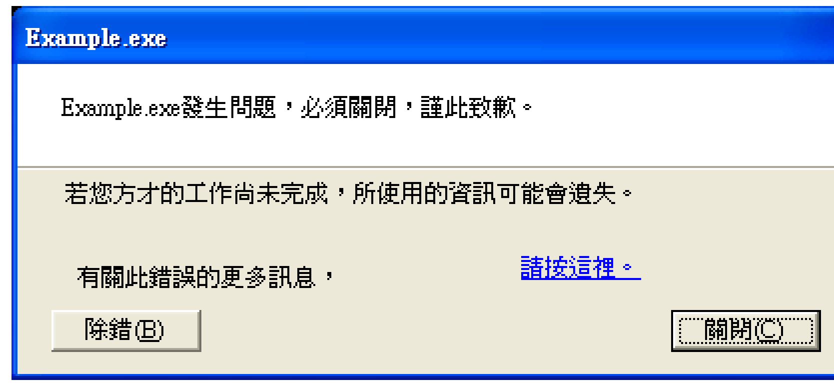
```
number = ? 78.5
number = ? 89
number = ? 95
number = ? 67.5
sum = 330.000000
```

1.1 Array Bounds

Program array_3.c

```c
#include <stdio.h>
int main(void) {
  double list[4], sum = 0;
  int i;
  for (i = 0; i < 4; i++) {
    printf("number = ? ");
    scanf("%lf", &list[i]);
  }
  for (i = 9990; i < 9994; i++)
    sum += list[i - 90];
  printf("sum = %f\n", sum);
  return 0;
}
```






```
number = ? 78.5
number = ? 89
number = ? 95
number = ? 67.5
sum = -78344.296783
```

1.1 Array Bounds

- The value of an array subscript must be in the range 0 to 3 in the above example.
- An array subscript value outside this range will cause a run-time error -- *array out of bound*.
- The effect of the error is unpredictable in C !
- It is the programmer's responsibility (not the compiler's) to ensure correct value of array subscripts.

2. Two-Dimensional Array (Table)

	Col 0	Col 1	Col 2	Col 3	Col 4
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

```
short a[3][5];  // a table with 3 rows,
                                5 columns
```

- The first dimension denotes the *row-index*, which starts with 0.
- The second dimension denotes the *column-index*, which also starts with 0.

2. Two-Dimensional Array (Table)

Program array_4.c

```
#include <stdio.h>
int main(void) {
  int a[3][5], row, col, sum=0;
  /* fill the array */
  for (row = 0; row < 3; ++row)
    for (col = 0; col < 5; ++col)
      a[row][col] = row + col;
  /* print and sum values in the array */
  for (row = 0; row < 3; ++row) {
    for (col = 0; col < 5; ++col) {
      printf("a[%d][%d] = %d  ", row, col, a[row][col]);
      sum += a[row][col];
    }
    printf("\n");
  }
  printf("\nsum = %d\n\n", sum);
  return 0;
}
```

We use nested-loops to process the rows and the columns.

2. Two-Dimensional Array (Table)

```
a[0][0] = 0  a[0][1] = 1  a[0][2] = 2  a[0][3] = 3  a[0][4] = 4
a[1][0] = 1  a[1][1] = 2  a[1][2] = 3  a[1][3] = 4  a[1][4] = 5
a[2][0] = 2  a[2][1] = 3  a[2][2] = 4  a[2][3] = 5  a[2][4] = 6

sum = 45
```

- In processing array elements of a two-dimensional array, each dimension requires a single loop.

- Therefore, a two-level nested-loop is necessary.

3. Multi-Dimensional Array (N-Dim)

Program array_4s.c

```
#include <stdio.h>

int main(void)
{
  long double sum = 0;     // a single variable
  long double a[5];        // a one-dimensional array
  long double b[9][5];     // a two-dimensional array
  long double c[7][9][5]; // a three-dimensional array
  int page, row, col;      // loop variables for indexes

  for (page = 0; page < 7; ++page)
    for (row = 0; row < 9; ++row)
      for (col = 0; col < 5; ++col)
        sum += c[page][row][col] = 9999;

  return 0;
}
```

3. Multi-Dimensional Array (N-Dim)

- The size (number of elements) of an array *grows exponentially* as the number of dimensions (pairs of brackets) increases!

- What is the total number of elements?

- Each element is a **`long double`** in the example above.

4. Array Initialization for 1-D Array

```
int a[4] = { 3, 4, 5, 6 };
```

Initializers

3	4	5	6
a[0]	a[1]	a[2]	a[3]

```
int a[4];
a[0] = 3;
a[1] = 4;
a[2] = 5;
a[3] = 6;
```

Equivalent!

4. Array Initialization for 1-D Array

```
int a[ ] = { 3, 4, 5, 6 };
```

Initializers

3	4	5	6
a[0]	a[1]	a[2]	a[3]

```
int a[4];
a[0] = 3;
a[1] = 4;
a[2] = 5;
a[3] = 6;
```

C can "guess" the size of the array if we use initializers!

If a 1-D array is declared without a size, it is implicitly given the size of the *number of initializers*.

4. Array Initialization for 1-D Array

```
int a[4] = { 3, 4          };
```

Short!

a[0]	a[1]	a[2]	a[3]
3	4	0	0

```
int a[4];
a[0] = 3;
a[1] = 4;
a[2] = 0;
a[3] = 0;
```

When a list of initializers is shorter than the number of array elements to be initialized, the remaining elements are initialized to zero.

Points to Note

- C provides basically TWO kinds of array storage:
  - Fixed-size (~ automatic/ static/ call-stack-frame) array
    - i.e. programmer **pre-determined size** during *compile-time*
    - e.g. `double a[4000];                  // size = 4000`
    - e.g. `int  b[] = {1, 3, 5, 7, 9}; // size = 5`
    - e.g. `char c[] = "Hello";            // size = 6`
  - Dynamic (~ heap/ malloc) array
    - i.e. pointer-operated, **memory allocated** during *run-time*
    - *Programmer Allocates and Frees*, according to need
    - e.g. `double *p = malloc( sizeof(double) * n );`
    - e.g. `p[n - 1] = 3.14;`
    - e.g. `free(p);`

Practice 1

- Store 10 integers (0-9) into an Array and print them.

```
#include<stdio.h>
int main()
{
        int a[10];
        int i=0;
        do{
                a[i]=i;
                printf("%d\t",a[i]);
                i=i+1;
        }while(i<10);
        return 0;
}
```

Practice 2

- Store 10 integers (0-9) into an Array ;add them together; and print sum answer.

Practice 3 (challenging)

- Convert (decimal to base 2)
- 12 --- 1100
- 7 --- 111
- Algorithm:
- http://www.wikihow.com/Convert-from-Decimal-to-Binary

Steps of short division

- **Set up the problem**. For this example, let's convert the decimal number 156_{10} to binary.
- **Write the integer answer (quotient) under the long division symbol, and write the remainder (0 or 1) to the right of the dividend**.
- **Starting with the bottom remainder, read the sequence of remainders upwards to the top**. For this example, you should have 10011100. This is the binary equivalent of the decimal number 156. Or, written with base subscripts: $156_{10} = 10011100_2$
- **Continue downwards, dividing each new quotient by two and writing the remainders to the right of each dividend**. Stop when the quotient is 0.

Template:

```
#include <stdio.h>
int main()
{
        int i, input;
        int out[50];
        while(1)
        {
        //initialize the array, to do
        printf("input a positive integer:\n");
        scanf("%d",&input);
        i=0;
        do{
         //three lines to implement the short division method
        }while(input!=0);
        //use a loop to output the integers(1 or 0) stored in the array
        printf("\ndo you want to continue? input 1 or 0\n");
        scanf("%d",&i);
        if(i==0) {
                printf("get %d, exit the program\n",i);break; }
        else{
                 printf("get %d, continue\n",i);continue;}
        }
return 0;
}
```