Data Types in C

We often want computers to process large amounts of data, so we need ways to manipulate lists and other groupings of large amounts of data. To satisfy these needs, we require more than just the few basic data types that are built-into the language. We require constructs and methods that allow us to not only manipulate data but also create data abstractions.

Three constructors that allow the definition of complex structured types:

- pointers
- arrays
- records

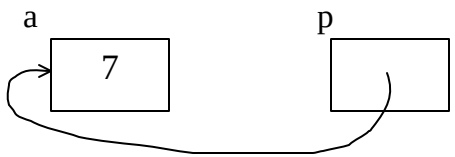
Pointers

A **pointer** is simply the internal machine address of a value inside the computer's memory.

```
int a;
int *p;

// p is a pointer to int.

a = 7;
p = &a;
```



We can reach the contents of the memory cell addressed by p:

```
printf("%d\n", *p);
```

Fundamental pointer operations

1. Assigning the address of a declared variable:

```
int a, *this, *that;
this = &a;
```

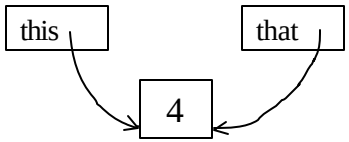
2. Assigning a value to a variable to which a pointer points.

```
*this = 4;
```

assigns 4 to the `int` variable to which the pointer variable refers.

3. Making one pointer variable refer to another:

```
that = this;
```



4. Creating a new variable. Given:

```
int  *this;
int * that;
```

then

```
this = malloc(sizeof(int));
```

allocates a new memory space for the pointer and makes the pointer variable refer to it. e.g.:

Addressing and Dereferencing

```
int a, b;
int *p, *q;

a=42; b=163;
p = &a;
q = &b;
```

```
printf("*p = %d, *q = %d\n", *p, *q);
```

```
*p = 17;
printf("a = %d\n", a);

p = q;
```

```
*p = 2 * *p – a;
printf("b = %d\n", b);
```

```
p = &a;
printf(“Enter an integer: “);
scanf(“%d”, p);
```

```
*q = *p;
```

```
double x, y, *p;

p= &x;
y = *p;

equivalent to

y = *&x;

or

y =x;
```


Passing parameters by reference

```
void SetToZero (int var)
{
   var = 0;
}
```

You would make the following call:

```
SetToZero(x);
```

This function has no effect whatever.
Instead, pass a pointer:

```
void SetToZero (int *ip)
{
   *ip = 0;
}
```

You would make the following call:

```
SetToZero(&x);
```

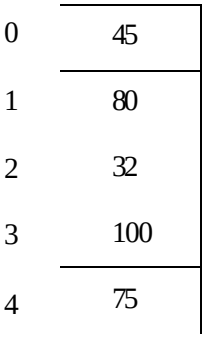
This is referred to as call-by-reference.

Arrays

- An array is a sequence of data items that are of the same type that can be indexed and that are stored contiguously.
- Arrays are a data type that is used to represent a large number of homogenous values.
- The elements of an array are accessed by the use of subscripts (also called index)
- Arrays of all types are possible, including arrays of arrays.

An array of 5 integers:

```
int grade[5];
```




```
grade[0] = 45;
grade[1] = 80;
grade[2] = 32;
grade[3] = 100;
grade[4] = 75;
```

- Indexing of array elements starts at 0.

Array declaration:

type identifier[*size of the array*];

i.e. *number of elements*: must be a positive constant integral expression

- A typical array declaration allocates memory starting from a base address. The array name is in effect a pointer constant to this base address.
- To store the elements of the array the compiler assigns an appropriate amount of memory, starting from the base address. (i.e. *identifier*)

Example:

```
int x[10];
double y[50];
```

- The identifier (e.g. x) is a constant which points to the base address of the array (it can be thought as a constant pointer).
- The elements are reached by using the *identifier* + index.

Initializing an Array

```
int y[5] = {1,3,5,7,9};

double x[10] = {1.1, 2.2, 3.3}; //rest is zero

int m[]={2,4,6,8,10}; //equivalent to m[5]
```

Subscripting

Suppose,

```
int i, a[size];
```

Then we can write:

```
a[i]
```

More generally:

```
a[expr]
```

➢ The value of an index must lie in the range 0 to ***size***-1

Effective and allocated sizes

- The number of array elements usually depends on the user's data.
- The size of the array specified in the declaration is called the **allocated size**.
- The number of elements actively in use is called the **effective size**.

Example:

```
#define MaxJudges 100

int scores[MaxJudges];
```

To keep track of the effective size, you would need to declare an additional variable as follows:

```
int nJudges;
```

Passing arrays as parameters

In a function definition, a formal parameter that is declared as an array is actually a pointer.

When an array is being passed, its base address is passed call-by-value. The array elements themselves are not copied.

Example:

```
int Mean(double a[], int n)
{
    int j;
    double sum = 0;

    for (j=0; j < n ; j++)
        sum = sum + a[j];

    return (sum/n);
}
```

Note: a[] is a notational convenience. In fact

`int a[]` o **`int *a`**

Calling the function:

```
int total, x[100];

total = Mean(x, 100);
total = Mean(x, 88);
total = Mean(&x[5], 50);
```


Two-dimensional Arrays

The general syntax for declaring a two-dimensional array:

<type> <id_name>[row_size][col_size];

Example:

```
int table[5][10];

table[0][0] = 3;      //like an integer variable

table[1]  → one-dimensional array of 10 integers
```

Printing the contents of a 2-D Array:

```
int vals[5][5];
int i,j;

for (i=0; i<5; i++) {
   for (j=0; j<5; j++)
        printf("%d  ", vals[i][j]);
   printf("\n");
}
```

Pointers and Arrays

- Arrays are implemented as pointers.
- The operations on pointers make sense if you consider them in relation to an array.

Consider:

```
double list[3];
double *p;
```

`&list[1]` : is the address of the second element

`&list[i]` : the address of list[i] which is calculated by the formula

base address of the array + `i` * *8*

Pointer arithmetic

- If we have :

```
p = &list[0];        // Or: p = list;
```

then

`p + k` *is defined to be* `&list[k]`

- If

```
p = &list[1];
```

then

`p – 1` *corresponds to* `&list[0]`

`p + 1` *corresponds to* `&list[2]`

Pointer Arithmetic (cont.)

- The arithmetic operations *, /, and % make no sense for pointers and cannot be used with pointer operands.
- The uses of + and – with pointers are limited. In C, you can add or subtract an integer from a pointer, but you cannot, for example add two pointers.
- The only other arithmetic operation defined for pointers is subtracting one pointer from another. The expression

```
p1 – p2
```

where both `p1` and `p2` are pointers, is defined to return the number of array elements between the current values of `p2` and `p1`.

- Incrementing and decrementing operators:

`*p++` is equivalent to `*(p++)`

Example: Illustrates the relationship between pointers and arrays.

```
int SumIntegerArray(int *ip, int n)
{
   int i, sum;

   sum = 0;
   for (i=0; i < n; i++) {
      sum += *ip++;
   }
   return sum;
}
```

Assume

```
   int sum, list[5];
```

are declared in the `main` function. We can make the following function call:

```
   sum = SumIntegerArray(list, 5);
```


Records: `struct` Construct

```
struct party{
    int house_number;
    int time_starts;
    int time_ends;
};

struct party halloween_party,
             new_years_party;
```

Rather than a collection of 6 variables, we have:

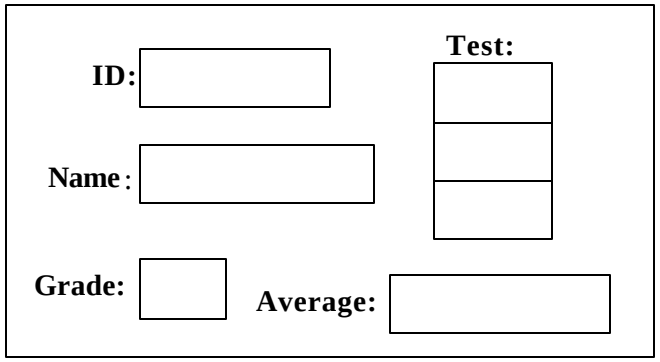
- 2 variables with 3 fields each.
- both are identical in structure

- We may have unlimited number of identical variables by declaring a single structure.
- Changes to declaration cascade to all variables of that structure:

```
struct party{
    int house_number;
    int time_starts;
    int time_ends;
    char police_came;
};
```

Another Example

Student Record




```
struct student {
    int ID;
    char name[20];
    int test[3];
    double average;
    char grade;
};
```

Variable declaration:

```
struct student s1, s2, s3,
              class[100],
              *sptr;
```


Accessing Structures

Two operators are used to access members of structures:

1. the structure member operator, also called the dot operator (`.`) and,
2. the structure pointer operator, also called the arrow opeartor (`->`).

Dot operator

```
s1.test[1] = 90;
s1.average = (s1.test[0]+ s1.test[1] +
              s1.test[2])/3.0;
s1.grade = 'A';
for(j=0; j<20; j++)
  printf("%c",s1.name[j]);
```

Arrow Operator

```
sptr = &s2;
sptr->grade = 'B';
```

`sptr->grade` is equivalent to `(*sptr).grade`

Parenthesis are necessary because the operators `->` and `.` have highest precedence.

`*sptr.grade` is equivalent to `*(sptr.grade)`
`&s2->grade` is equivalent to `&(s2->grade)`

Creating New Data Types

- The atomic types and arrays are “given” to us.
- We can use all these constructors to create new “user-defined” data types.
- Once we create a new data type, we can then declare variables to be of the new type, just as we can declare variables to be of the “built in” types.
- Creating variables of a new data type is a two-step process:
  1. Define the new type
  2. Declare variables of that type
- Creating the new data type does not provide any variables, only the template by which variables may then be declared.

The use of `typedef`

C provides `typedef` facility so that an identifier can be associated with a specific type.

Example 1:

```
typedef int color;

color red, blue;
```

Example 2:

```
typedef struct{
          int Day;
          int Month;
          int Year;
        } Date;

Date birthday, *d;
```

Example 3:

```
typedef int Vector[20];
typedef Vector Matrix[20];

Matrix m1;
Vector v1;
```

Example 4:

```
typedef int *pointerType;

pointerType p;
int x;

p = &x;
*p = x;
x = *p + 1;
```

Example 5:

```
typedef char String[20];

String First, Last;
```