So far

Some Primitive Data-types

Representation of numbers

Arrays

Character

Pretty Printing
The story so far ...

- We have written some non-trivial programs
- We have seen various control flows, and
- We have hopefully seen how everything really can be brought down to PCAL-code.

Arrays and the char data-type

Our objective is to understand two simple extensions to the data types that we know of as yet, viz., float and int. Again www.cplusplus.com/doc/tutorial for reference.
Some Primitive Data-types

We have seen the following data-types so far:

- **int**: integer.
- **float**: floating point real number.
- **long**: higher-precision integer.
- **double**: higher precision real.

We have seen that each of the basic data-types have operators on them such as comparisons, assignments, additions and others.

We now see a new data-type called **arrays** which is a systematic composition of the primitive data types.
Representation of numbers

- Internally, each register of the computer is a fixed width (say 32 or 64). Each place in this register is called a bit. Each bit can store either a 0 or a 1.

\[ m = \begin{array}{cccccccc} b_{31} & b_{30} & \ldots & b_3 & b_2 & b_1 & b_0 \end{array} \]

- Whence all data such as integers, reals, and (later) characters are coded as strings of 0’s and 1’s.

- Integers are represented either as \texttt{int} or \texttt{long}. The \texttt{int} means a 32-bit binary representation, while \texttt{long} is 64-bit. Positive numbers must have \( b_{31} = 0 \) and the value then equals

\[ \sum_i b_i 2^i \]

- Examples 00...01001 is 9, 000...0110 is 6 and so on.

- Negative numbers have \( b_{31} = 1 \) but there are many options of coding.
Positive real numbers are stored as

\[ r = m \times 2^e \]

where \( 0 \leq m < 1 \) and \( e \) is an integer.

Thus a real is stored in two memory locations: the mantissa \( m \) and the exponent \( e \).

Negative reals are coded similar to negative integers.
Positive real numbers are stored as

\[ r = m \times 2^e \]

where \( 0 \leq m < 1 \) and \( e \) is an integer.

Thus a real is stored in two memory locations: the mantissa \( m \) and the exponent \( e \).

Negative reals are coded similar to negative integers.

Different Data types have different encodings.

Operations are designed around this encoding.
Arrays

A Question

How many 0-1 sequences are there of length 50 in which there are no two consecutive zeros?

- Let $a_n$ be the sequences as above, but ending in zero.
- Let $b_n$ be the sequences as above, but ending in one.

It is clear that:

\[
\begin{align*}
    a_{n+1} &= b_n \\
    b_{n+1} &= a_n + b_n
\end{align*}
\]

This recurrence coupled with:
$a_1 = b_1 = 1$ solves the problem.
Arrays

A Question

How many 0-1 sequences are there of length 50 in which there are no two consecutive zeros?

- Let $a_n$ be the sequences as above, but ending in zero.
- Let $b_n$ be the sequences as above, but ending in one.

It is clear that:

\[
\begin{align*}
    a_{n+1} &= b_n \\
    b_{n+1} &= a_n + b_n
\end{align*}
\]

This recurrence coupled with:

\[
\begin{align*}
    a_1 &= b_1 = 1
\end{align*}
\]

solves the problem.

seq.c

```c
#include <iostream.h>
// computes number of 0-1 sequences
// without two consecutive 0's
int main()
{
    int N, i, a[50], b[50];
    a[0]=1; b[0]=1;
    for (i=1; i<50; i=i+1)
    {
        a[i]=b[i-1];
        b[i]=a[i-1]+b[i-1];
    }
    cout << "N? 
";
    cin >> N;
    cout << a[N-1]+b[N-1] << "\n";
}
```
Arrays

What is happening?

- The declaration `int a[50]` declares a sequence of variables `a[0], a[1], ..., a[49]`.
- Let the contents of the variable `i` be, say `r`. Then the variable `a[i]` accesses the `r`-th location from this sequence.
- Thus, an array allows us to access any particular element of the collection.
- Such a collection is called an array.

seq.c

```c
#include <iostream.h>
// computes number of 0-1 sequences
// without two consecutive 0’s
int main()
{
    int N, i, a[50], b[50];
    a[0]=1; b[0]=1;
    for (i=1; i<50; i=i+1)
    {
        a[i]=b[i-1];
        b[i]=a[i-1]+b[i-1];
    }
    cout << "N? \n";
    cin >> N;
    cout<< a[N-1]+b[N-1]<< "\n";
}
```
More Arrays

- What we saw was a **1-dimensional array** of integers.
- `int a[10][10]` is a $10 \times 10$ two-dimensional array of integers. An element of this array is `a[4][3]`. 
More Arrays

- What we saw was a 1-dimensional array of integers.
- `int a[10][10]` is a $10 \times 10$ two-dimensional array of integers. An element of this array is `a[4][3]`.

Naturally...

Arrays occur naturally.
- Your computer screen is a $700 \times 1100$ array of `pixels`. Each pixel holds a color.
- Space is a 3-dimensional array with each element having attributes such as `mass`, `charge`, `spin`, `refractive index` and so on.
- **Space-Time** is a 4-dimensional array...
Matrix Multiplication

A matrix, after all, is a 2-dimensional array. Given an $a \times b$-matrix $A$, and a $b \times c$-matrix $B$, $AB$ is a $a \times c$-matrix.

If $C = AB$, then

$$C[i][j] = \sum_k A[i][k] \times B[k][j]$$

We first read in the matrices $A$ and $B$. Next, $C$ is computed as above. $C[i][j]$ is outputted as soon as it is ready.

Watch for indices and the input/output.

File name matmult.c
Matrix Multiplication

A matrix, after all, is a 2-dimensional array. Given an \(a \times b\)-matrix \(A\), and a \(b \times c\)-matrix \(B\), \(AB\) is a \(a \times c\)-matrix.

If \(C = AB\), then

\[
C[i][j] = \sum_k A[i][k] \times B[k][j]
\]

We first read in the matrices \(A\) and \(B\). Next, \(C\) is computed as above. \(C[i][j]\) is outputted as soon as it is ready.

Watch for indices and the input/output.

File name matmult.c

```c
#include <iostream.h>
// performs matrix mult
int main()
{
    int a,b,c,i,j,k;
    int A[10][10], B[10][10], C[10][10];
    cin >> a >> b;
    for (i=0;i<a;i=i+1)
    {
        for (j=0;j<b; j=j+1)
        {
            cin >> A[i][j];
        };
    }
    // read in B here skipped)
    compute C=A*B
    
} Milind Sohoni ()
CS101 Computer Programming and Utilization
May 12, 2006 14 / 22
```
The Multiplication

```
for (i=0; i<a; i=i+1)
{
    for (j=0; j<c; j=j+1)
    {
        C[i][j]=0;
        for (k=0; k<b; k=k+1)
        {
            C[i][j]=C[i][j]+A[i][k]*B[k][j];
        }
        cout << C[i][j] << " ";
    }
    cout << "\n";
}
```

- Note the nested for loops.
- Note the order in which the elements are read, computed and printed:

```
1 2 3
4 5 6
```

- Note the location of the cout C[i][j].
- Note all the bounds in the for loops.
Character

C++ also defines a primitive type called `char`. Thus

```cpp
char pm;
char name[20];
```

defines `pm` as a single character and `name` as an array of length 20 of characters.

Reverse

Write a program to input a word and output its reverse.
C++ also defines a primitive type called `char`. Thus

```cpp
char pm;
char name[20];
```

defines `pm` as a single character and `name` as an array of length 20 of characters.

### Reverse

Write a program to input a word and output its reverse.

```cpp
#include <iostream.h>
int main()
{
    int i,N;
    char name[10];
    cout << "N?\n";
    cin >> N;
    cout << "word?\n";
    for (i=0;i<N;i=i+1)
    {
        cin >> name[i];
    }
    for(i=N;i>0;i=i-1)
    {
        cout << name[i-1];
    }
    cout<< "\n";
}
```
Pretty Printing

cout output frequently looks bad. For example an output of matmult.c may well look like this:

1 2
345 678

We would ideally like:

1 2
345 678

Help is around in the form of printf. The general command structure is as follows:

printf("%x1 %x2",var1,var2)
Pretty Printing

cout output frequently looks bad. For example an output of matmult.c may well look like this:

\begin{verbatim}
1 2
345 678
\end{verbatim}

We would ideally like:

\begin{verbatim}
  1  2
 345 678
\end{verbatim}

Help is around in the form of printf. The general command structure is as follows:

\begin{verbatim}
printf("%x1 %x2",var1,var2)
\end{verbatim}

\begin{verbatim}
#include <iostream.h>
int main()
{
  int a,b,c;
  float p,q,r;
  a=-1; b=10; c=100;
  p=123.456; q=0.1234; r=-12.34;
  printf("%5d \n",a);
  printf("%5d \n",b);
  printf("%5d \n",c);
  printf("%2d \n",a);
  printf("%2d \n",b);
  printf("%2d \n",c);
  printf("%8.4f \n",p);
  printf("%8.4f \n",q);
  printf("%8.4f \n",r);
  printf("%4.2f \n",p);
  printf("%4.2f \n",q);
  printf("%4.2f \n",r);
}
\end{verbatim}
**Pretty Printing**

`cout` output frequently looks bad.
For example an output of `matmult.c` may well look like this:

```
-1
10
100

-1
10
100

123.4560
  0.1234
-12.3400

123.46
  0.12
-12.34
```

```cpp
#include <iostream.h>
int main()
{
    int a,b,c;
    float p,q,r;
    a=-1; b=10; c=100;
    p=123.456; q=0.1234; r=-12.34;
    printf("%5d \n",a);
    printf("%5d \n",b);
    printf("%5d \n",c);
    printf("%2d \n",a);
    printf("%2d \n",b);
    printf("%2d \n",c);
    printf("%8.4f \n",p);
    printf("%8.4f \n",q);
    printf("%8.4f \n",r);
    printf("%4.2f \n",p);
    printf("%4.2f \n",q);
    printf("%4.2f \n",r);
}
```