The int data type

- **As stated earlier, the declaration**

```
int a;
```

  - **is an instruction to the compiler to reserve a certain amount of memory to hold the values of the variable a.**

- **How much memory?**

  - **Two bytes (usually, but not always, 16 bits)**
  - **A bit is either 0 or 1**
  - **So the computer sets aside 16 places, each of which can store either 0 or 1**

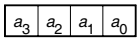
- **So the computer sets aside 16 places each of which looks like the following.**

0	1	0	1	1	0	0	1	0	1	1	1	0	1	0	1

- **This bit pattern corresponds to the binary representation which the computer uses to represent numbers internally.**
- **The number 40000 is too large to be represented in this way.**
- **But by what logic did the computer arrive at the figure 20000 + 20000 = -25536?**

The binary representation

- **Consider, first, the representation of a number with only 4 bits available.**
- **The bit pattern**



- **where each a_i is either 0 or 1 can be interpreted as the number**

$a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0$

- **Thus, a_0=0, a_1=1, a_1=1, a_0=1, or the bit pattern 0111, corresponds to the number 0+4+2+1 = 7**

Limitations of the binary representation with fixed width

- **(1) This method can represent only non-negative integers.**
  - **In C-language, non-negative integers are represented by the data type unsigned int.**
- **(2) The maximum integer that can be represented with 4 bits is**

$2^3 + 2^2 + 2^1 + 2^0 = 15$

- **(3) The maximum integer that can be represented with *n* bits is**

  $2^{n-1} + 2^{n-2} + 2^{n-3} + \ldots + 2^0 = 2^n - 1$

  (The sum is obtained by summing the geometric series or geometric progression.)

  - **Hence, the maximum integer that can be represented with 16 bits is 65535.**
  - **This is the maximum value for the unsigned data type.**

Sign-magnitude representation

- **How are negative numbers represented?**
- **One way is to reserve one bit for the sign of the number.**
  - **This is called the sign bit.**
  - **Then all numbers beginning with a 1 are regarded as negative.**
  - **positive numbers are expressed as before.**
- **Thus, 1000 0001 is the representation of -1**

Difficulties with the sign-nagnitude representation

- **What is 1000 0000 ?**
  - **This is "negative zero"!**
  - **Positive zero is still 0000 0000**
- **Consider the sum**

```
   0000 0001   (+1)
+  1000 0001   (-1)
-------------------
   1000 0010   (-2)
-------------------
```

- **Thus, we now have the conclusion: 1+ (-1) = -2**

One's complement

- **A second method is to use the one's complement for** *negative numbers*.
- **The one's complement of a bit pattern is obtained by replacing all 0's by 1's and 1's by 0's. E.g.**

Number	0	1	0	1
1^c	1	0	1	0

- **Then all numbers beginning with a 1 are regarded as negative.**
- $-a$ **is represented by** 1^c **of** a **(for positive** a**).**

Difficulties with one's complement

- With 8 bits, the one's complement representation of -38 is obtained as follows.
  - First obtain the binary representation of 38
    38 = 0010 0110 (= 32+4+2)
  - Take the 1^C of this bit pattern to obtain
    -38 = 1101 1001.
- If we add 38 + (-38) we must get
  38 + (-38) = 1111 1111
  - This is the 1^C of 0000 0000, so it should be something like negative zero.

- But we also have the sum

```
    0000 0000   (+0)
-   0000 0001  -(+1)
    ---------
    1111 1111
    ---------
```

which is to be understood like the sum

```
      1 0000 0000
-       0000 0001
      ----------
        9999 9999
      ----------
```

in the decimal representation, since 1 plays the role of both 1 and 9 in the binary representation.

- **Thus, we have the difficulty that**
  0-1 = 38 - 38

Two's complement

- **To correct this problem, we use the two's complement for negative numbers.**
- **To get the 2^c of a number**
  - **Take the 1^c**
  - **Add 1 to it.**
- **This makes the above sum come out correct.**

- **1111 1110 (1^c representation of -1)**
  becomes
  1111 1111 (2^c representation of -1)

  so that $0 - 1 = -1$

Difficulties with the two's complement

- **But we now have a new problem.**
- **Consider the sum 127 + 1 (using 8 bits)**

```
  0111 1111    (127)
+ 0000 0001    (1)
-----------------
  1000 0000    (2c representation of -128)
-----------------
```

- **Thus,** $127 + 1 = -128$

Solving the puzzle

- **Extending the above logic to 16 bits, we get**

  32767 + 1 = –32768

- **A graphical way to describe this would be that the computer uses a number circle instead of a number line!**

Solving the puzzle (contd)

- **Thus, the computer adds correctly up to 32767**
- **Thereafter, it “wraps around”.**
- **Since, 40000 - 32767 = 7533**

  we get

  20000 + 20000 = –32768 +7532 = –25536

- **We can easily verify the above considerations, by means of a simple program.**

Checking the calculations

```
/*Program Maxint.c*/
/*Function: To show the maximum value of
int, and the wrap around property*/

#include <values.h>
#include <stdio.h>
#include <conio.h>
main()
 {
  printf ("\n Maximum int = %d", MAXINT);
  printf ("\n Max int + 1 = %d", MAXINT+1);
  getch();
  return 0;
 }
```


- **Output:**
  Maximum int: 32767
  Max int + 1 = -32768

Checking that 2 bytes is NOT always 16 bits

- In the above program, we #included values.h, since this is the header file where the value MAXINT is defined.
- Why is it necessary to have a special constant for this?
- An int is 2 bytes, but not necessarily 16 bits.
- On a Win 32 platform, an int is 32 bits.
- We can check this out by running the Addint.c program in Visual C++.

- We now get
  20000 + 20000 = 40000
- But the basic principles remain the same, as we can check by adding enough zeros after 2.

Note: Using the Visual C++ IDE

- **This assumes that you have access to Visual C++.**
  - **This refers to Visual C++ 6.0**
  - **But the same thing will work with any version of Visual C++.**
- **Step 1: Start the Visual C++ IDE.**
- **Step 2: Click File → New**
  - **Select Win 32 Console Application**
  - **Select "Create an Empty Project"**
- **Step 3: Click Project → Add to Project → Files (or New if you want to key in the code directly).**

- **To compile and run, select Build → Run, and click OK.**
- **(Or use other choices under the Build option, if there are mistakes in the code.)**

The long modifier

- **The same sort of effect can be achieved in Turbo C, by using the modifier long.**
- **This allows 32 bits of storage (in Turbo C) and changes the above values.**
- **We now have**
  2147483647 + 1 = - 2147483648
- **But there is no change in the principle of representing or adding numbers.**

- **The modifiers long and unsigned can be used together.**

```
unsigned a;
long b;
unsigned long int c;
```

  - **are all valid declarations.**
- **The order in which the modifiers are used does not matter.**
- **There is no difference between**

```
unsigned long c;
```

  - **and**

```
long unsigned c;
```

- **For unsigned long we have the equation**

  4294967295 + 1 = 0

- **Obviously,**

```
long short int d;
```

  - **will generate the compiler error 'too many types in declaration'.**

- **IMPORTANT NOTE: C99 permits the data type**

```
long long
```


Fundamental data types II: floats

- **Is this the best we can do in computer arithmetic?**
- **We can improve matters by using the floating point data type.**
- **The integer data type cannot also be used where fractions are involved.**
- **For fractions, one uses the float data type, declared as follows:**

```
float a, b, c;
```