W.P.Hwang
CS-CCS-NCTU

# *Lecture – Basic data types*

## Boolean type

- Traditionally, C has no Boolean type.
  Instead, it treats 0 as false, and any non-0 value as true, e.g.

```
if ("Snoopy likes C")
   printf("Snoopy is crazy!\n");
else
   printf("Snoopy is sane.\n");
```

- The representative value for true is 1. For example, a relational or logical expression yields a 1 (true) or a 0 (false), e.g.

```
printf("%d",2<3);     // 1
```

- In C++, **`bool`** is the built-in Boolean type with two values **`true`** and **`false`**.

- Usually, **`sizeof(bool)`**=1
  Comment
  **`sizeof(`***T***`)`** = the # of bytes occupied by variables of type *T*
  One bit is enough for a Boolean value.
  However, modern computers are mostly byte-addressable.

- **`bool`** may be used as a 1-bit integer type – a **`bool`** variable can only hold a 1 or a 0, e.g.

```
bool x=true;
x=x+1;
printf("%d",x);        // 1
```

  Observe that there are two implicit conversions:

  bool → int, integral promotion  $\boxed{1} \rightarrow \boxed{0}\boxed{0}\boxed{0}\boxed{1}$

  **`x = x + 1;`**  $\boxed{1} \leftarrow \boxed{0}\boxed{0}\boxed{0}\boxed{2}$

  int → bool, Boolean conversion

- Example – Find the number of divisors for $n \geq 1$

Algorithm A – $O(n)$ algorithm

```
int ndivs(int n)
{
   int r=0;
   for (int d=1;d<=n;d++)
      if (n%d==0) r++;        ①
   return r;
}
```

① or, `r += n%d==0;` // NOT recommended
or, `if (!(n%d)) r++;` // NOT recommended

Algorithm B – $O(n)$ algorithm

```
int ndivs(int n)
{
   int r=n==1? 1: 2;
   for (int d=2;d<=n/2;d++)
      if (n%d==0) r++;
   return r;
}
```

Algorithm C – $O(\sqrt{n})$ algorithm

Let $n = de$, then $d > \sqrt{n} \wedge e > \sqrt{n} \Rightarrow de > n$. A contradiction.
Assume that $d \leq \sqrt{n}$, then $d^2 \leq n$

```
int ndivs(int n)
{
   int r=n==1? 1: 2;
   for (int d=2;d*d<=n;d++)  ②
      if (n%d==0)             ①
         if (d*d==n) r++; else r+=2;
   return r;
}
```

① or,
```
if (n%d==0) {
        r++;
        if (d*d!=n) r++;
    }
```

- Example (Cont'd)

  ② In C, one may substitute **`d<=sqrt(n)`** directly for **`d*d<=n`**, albeit it is less efficient.

  Note that the signature of **`sqrt`**, declared in **`<math.h>`**, is

  ```
  double sqrt(double);
  ```

  Thus, there are two implicit conversions:

  ```
       int → double
  d <= sqrt(n)
              int → double
  ```

  As it is, the comparison works well. But, what we actually mean here is $d \le \lfloor\sqrt{n}\rfloor$, and so we should write

  ```
  d <= (int)sqrt(n)
              double → int
  ```

  where the cast operator **`(int)`** causes an explicit conversion

On linking with the math library in Unix

Let **`ndivs.c`** be the program that includes **`<math.h>`** and calls **`sqrt`**.

```
% gcc ndivs.c -lm
```

On C89/C99 math library

In C89, there are only **`double`** versions of the math functions in **`<math.h>`**. But, there are three floating-point types: **`float`**, **`double`**, **`long double`**.

This is problematic. For example, consider

```
float x=3.14f;
x=x+sqrt(x);
```

Since only the float type is involved, it is reasonable to expect single precision floating point arithmetic.

- Example (Cont'd)

  However, there are three implicit conversions involved:

$$\mathtt{x} = \overbrace{\underset{\downarrow\ \text{float} \to \text{double}}{\mathtt{x}} + \mathtt{sqrt(}\underset{\text{float} \to \text{double}}{\mathtt{x}}\mathtt{)}}^{\text{double} \to \text{float}}\mathtt{;}$$

  In addition to the **double** versions, C99 adds **float** and **long double** versions for the functions in **<math.h>**, e.g.

```
float sqrtf(float);
long double sqrtl(long double);
```

  Thus, in C99, we may write **x=x+sqrtf(x);**

  ### On monomorphic and polymorphic languages

  C is monomorphic, but C++ is polymorphic.
  In C++, the three versions for the math functions have the same name, e.g.

```
float sqrt(float);
double sqrt(double);
long double sqrt(long double);
```

  Thus, in C++, we may simply write **x=x+sqrt(x);**
  The compiler will select the **float** version for us, since it is better than the other two versions.

  However, in C++, we can't write **d<=sqrt(n)**.

  Why?

  Because they are all callable

```
float sqrt(float);               // int → float
double sqrt(double);             // int → double
long double sqrt(long double);   // int → long double
```

  However, none is the best – the call is ambiguous, meaning that the compile can't decide which one to call.

- Example (Cont'd)

Thus, in C++, we have to write

**`d<=sqrt((double)n)`**

assuming that the **`double`** version is preferred.

On orders of functions

Alg. A $n = 1n^1 = O(n)$

Alg. B $\frac{1}{2}n - 1 = O(n)$

Alg. C $n^{\frac{1}{2}} - 1 = O(\sqrt{n})$

The big-*O* notation emphasizes the following concepts:

1 The lower-order terms are immaterial.
2 The order of a function is much more important than the coefficient of the highest-order term.

Obviously, among the three algorithms, Alg. C is the best.
In the absence of Alg. C, Alg. B is clearly better than Alg. A – the coefficient is considered when the order is the same.

- Example – Determine if an integer $n \geq 2$ is prime

Algorithm A – $O(\sqrt{n})$ algorithm

```
bool prime(int n)              ①
{
   return ndivs(n)==2;         ②
}
```

① In C, the return type is **`int`**.

② Never write
**`ndivs(n)==2? true: false`**
Because
**`ndivs(n)==2`** ⇔ **`ndivs(n)==2? true: false`**
That is, they are logically equivalent.

- Example (Con'td)

Algorithm B – $O(\sqrt{n})$ algorithm *in the worst case*

Version 1

```
bool prime(int n)
{
   for (int d=2;d*d<=n;d++)
      if (n%d==0) return false;
   return true;
}
```

Comment

Unlike Algorithm A that always takes $O(\sqrt{n})$ time, the running time of Algorithm B depends on input cases.
Best case: $n$ is even
In this case, Algorithm B takes $O(1)$ time.
Worst case: $n$ is prime
In this case, Algorithm B takes $O(\sqrt{n})$ time.

Version 2

```
bool prime(int n)
{
   bool r=true;
   for (int d=2;d*d<=n;d++)
      if (n%d==0) { r=false; break; }
   return r;
}
```

Version 3

```
bool prime(int n)
{
   bool r=true;
   for (int d=2;d*d<=n&&r;d++)  ①
      if (n%d==0) r=false;
   return r;
}
```

① Never write **`r==true`**
Note that **`r`** ⇔ **`r==true`** and **`!r`** ⇔ **`r==false`**

- Example (Cont'd)

Version 4 – Accumulation

```
// r=n%2!=0 && n%3!=0 && n%4!=0 && ...
bool prime(int n)
{
   bool r=true;        ①
   for (int d=2;d*d<=n&&r;d++)
      r=r&&n%d!=0;          // r &&= n%d!=0; ×
   return r;                // simplified to r=n%d!=0;
}
```

① true is the unit of logical and operation, i.e. true && *p* = *p*

## On analysis of algorithms

Best case – hardly useful
Average case – rely on probabilistic assumption
Worst case – the most important case

## On the prime problem

Algorithm B is indeed a poor algorithm for the prime problem.
Let $n$ be the largest prime that can be stored in $1000$ bits.
Then, $n \approx 2^{1000} = (2^{10})^{100} \approx (10^3)^{100} = 10^{300}$

On a 1 TIPS machine, Algorithm B takes a time in

| | |
|---|---|
| $\sqrt{n} \approx 10^{150}$ | iterations |
| $\geq 10^{150}/10^{12} = 10^{138}$ | seconds |
| $\geq 10^{138}/10^{5} = 10^{133}$ | days |
| $\geq 10^{133}/10^{3} = 10^{130}$ | years |

The AKS primality test algorithm (2002) runs in $O(\log^6 n)$ time.
On a 1 TIPS machine, it takes a time in

| | |
|---|---|
| $\log^6 n \approx \log^6 2^{1000} = 1000^6 = 10^{18}$ | iterations |
| $\geq 10^{18}/10^{12} = 10^{6}$ | seconds |
| $\geq 10^{6}/10^{5} = 10$ | days |

N.B.
TIPS (GIPS, MIPS) = Tera (Giga, Mega) Instructions Per Second
Kilo $2^{10} \approx 10^3$; Mega $2^{20} \approx 10^6$; Giga $2^{30} \approx 10^9$; Tera $2^{40} \approx 10^{12}$

## Arithmetic types

- Integral types **`bool`**† **`char`** **`int`**
  Floating types **`float`** **`double`**
  †C99 and C++ only

## Integral types (int)

- Six int types

  **`int = signed [int]`**
  **`[signed] short [int]`**
  **`[signed] long [int]`**
  **`unsigned [int]`**
  **`unsigned short [int]`**
  **`unsigned long [int]`**

- 2 $\leq$ sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long) $\geq$ 4
  size of signed version $=$ size of unsigned version

- C99 offers two more integral types:
  **`[signed] long long [int]`**
  **`unsigned long long [int]`**
  It is required that sizeof(long) $\leq$ sizeof(long long) $\geq$ 8

- Number system

  Base *b*
  Digits 0, 1, 2, …, *b*–1

  Example

  Binary 0, 1
  Octal 0, 1, 2, 3, 4, 5, 6, 7
  Decimal 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  Hexadecimal 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

  Positional number system

  $$7777_{10} = 7 \times 10^3 + 7 \times 10^2 + 7 \times 10^1 + 7 \times 10^0$$

  3 2 1 0

  $$1111_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 15_{10}$$

- Integers are usually represented by 2's complement.

| unsigned | | | signed | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 000 | 0 | | 011 | 3 | | 011 | 3 | | | |
| 001 | 1 | | 010 | 2 | | 010 | 2 | | | |
| 010 | 2 | | 001 | 1 | | 001 | 1 | | | |
| 011 | 3 | | 000 | 0 | ? | 000 | 0 | | | |
| 100 | 4 | | 100 | -1 | | 111 | -1 | ↔ | 001 | 1 |
| 101 | 5 | | 101 | -2 | | 110 | -2 | ↔ | 010 | 2 |
| 110 | 6 | | 110 | -3 | | 101 | -3 | ↔ | 011 | 3 |
| 111 | 7 | | 111 | -4 | | 100 | -4 | ↔ | 100 | 4 |

N.B.

| 1's complement | 2's complement |
|---|---|
| 101 | 101 |
| + 010 | + 011 |
| 111 | 1000 |

Cf.

| 9's complement | 10's complement |
|---|---|
| 234 | 234 |
| + 765 | + 766 |
| 999 | 1000 |

Characteristics

1 For signed integers, the leftmost bit is the sign bit – 0 for non-negative and 1 for negative.
For unsigned integers, there are no sign bits.

2 The same bit pattern may have different interpretations, e.g
100 = 4 as an unsigned integer
100 = -4 as a signed integer

3 An integer overflow causes a wrap-around.

| unsigned | | signed | |
|---|---|---|---|
| 111 | ~~1~~000 | 100 | 011 |
| + 1 | − 1 | − 1 | + 1 |
| ~~1~~000 | 111 | 011 | 100 |

- Ranges of $n$-bit signed and unsigned integers

  Signed $-2^{n-1} \sim 2^{n-1} - 1$
  Unsigned $0 \sim 2^n - 1$

- Typical sizes on 32-bit machines

| | size | signed | unsigned |
|---|---|---|---|
| char | 1 | -128~127 | 0~255 |
| short | 2 | -32768~32767 | 0~65535 |
| int | 4 | -2147483648~2147483647 | 0~4294967295 |
| long | 4 | 〃 | 〃 |
| long long | 8 | | 0~18,446,744,073,709,551,615 |

- Header file **<limits.h>**

| | | | |
|---|---|---|---|
| short | **SHRT_MAX** | **SHRT_MIN** | **USHRT_MAX** |
| int | **INT_MAX** | **INT_MIN** | **UINT_MAX** |
| long | **LONG_MAX** | **LONG_MIN** | **ULONG_MAX** |
| char | **CHAR_MAX** | **CHAR_MIN** | |
| | **SCHAR_MAX** | **SCHAR_MIN** | **UCHAR_MAX** |
| long long | **LLONG_MAX**† | **LLONG_MIN**† | **ULLONG_MAX**† |

†C99 only

- Integer overflow

  In C/C++, the results of signed integer overflows are undefined.

  **printf("%d",INT_MAX+1);** // undefined; usually **INT_MIN**

  But, the results of unsigned integer overflows are well-defined. (See later).

- Differences between math and programming language

  Certain mathematical properties do not hold in programming languages, e.g. let $x$, $y$, and $z$ be three integers and consider

  $(x + y) - z = x + (y - z)$
  $xy \leq z = x \leq z/y, \qquad y > 0$

  Both are true in math, but aren't in programming languages, for $x + y$ and $xy$ may overflow.
  For example, assume that they are all **int** variables and let $x =$ **INT_MAX**, $y = z = 2$

- Example – Generate $k!$, $k = 0, 1, 2, 3, \cdots$ until overflow

  Version 1 – $O(n^2)$ for $k = 0, 1, 2, \cdots, n$, since $\sum_{k=0}^{n}(k-1) = O(n^2)$

```
#include <limits.h>
void factgen(void)
{
   for (unsigned k=0;;k++) {      // default is true
      unsigned r=1;
      for (unsigned i=2;i<=k;i++)
         if (r<=UINT_MAX/i)       // ☹ r*i<=UINT_MAX
            r*=i;
         else {
            printf("%u!=Overflow!\n",k);
            return;
         }
      printf("%u!=%u\n",k,r);
   }
}
```

  Version 2 – $O(n)$ for $k = 0, 1, 2, \cdots, n$

```
void factgen(void)
{
   unsigned k=0,r=1;
   while (true) {
      printf("%u!=%u\n",k,r);
      k++;
      if (r<=UINT_MAX/k) r*=k;
      else {
         printf("%u!=Overflow!\n",k);
         break;                   // or, return;
      }
   }
}
```

- Integer constants

| | | | | |
|---|---|---|---|---|
| Prefix | `0` | octal, e.g. `010`≠`10` | | |
| | `0x` | hexadecimal | | |
| Suffix | `u` | unsigned | `ll`† | long long |
| | `l` | long | `ull`† | unsigned long long |
| | `ul` | unsigned long | †C99 only | |

N.B. Both upper- and lower-case letters are allowed. Also, the order of **`u`** and **`l`** (or **`ll`**) doesn't matter.

- Integer constants are nonnegative.
  e.g. - 3 is an expression, where - is 2's complement operator

- There are no constants for short integers.

- The type of an integer constant is the 1st type in the sequence

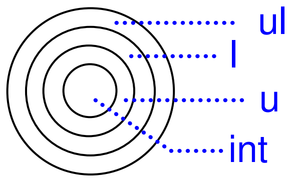
  **`int`**, **`unsigned`**, **`long`**, **`unsigned long`**, **`long long`**†, **`unsigned long long`**†

  in which the constant can fit, subject to the suffix (if any).

  Exception: For unsuffixed decimals, the sequence is

  C89/C++ **`int`**, **`long`**, **`unsigned long`**

  C99 **`int`**, **`long`**, **`long long`**

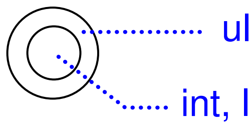



- Example (C89/C++)

  Assume that sizeof(int) = sizeof(long) = 4

  **`-2147483648 = 2147483648 ≠ -2147483647-1`**

  $1000...000_2$




```
#define INT_MIN (-2147483647-1)
#define LONG_MIN (-2147483647L-1L)   // one L suffices
```

- Example

  Unsuffixed decimals can never be of unsigned int.
  Otherwise, suppose sizeof(int)=2 and sizeof(long)=4, we have

  -30000 < 0 since 30000 is int
  -50000 > 0 since 50000 is unsigned
  -70000 < 0 since 70000 is long

# Integral types (char)

- Three char types

  **`char`** // as characters
  **`signed char`** // as tiny signed integers
  **`unsigned char`** // as tiny unsigned integers

- **`char`** = **`signed char`** or **`unsigned char`** is undefined.

- ASCII code
  **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange

  0~31 Control characters, e.g. `'\n'` = 10
  32~127 Printable characters, e.g. `'0'` = 48, `'a'` = 97
  128~255 Extended ASCII (Latin-1)

- Character constants

| | | | | |
|---|---|---|---|---|
| Printable | `'a'` | `'b'` | `'c'` | etc |
| Nonprintable | `'\b'` | `'\10'` | `'\x8'` | backspace |
| (escape sequence) | `'\n'` | `'\12'` | `'\xa'` | newline |
| | `'\r'` | `'\15'` | `'\xd'` | carriage return |

  e.g.
  ```
  printf("Pluto\rSnooo\bpy\n");    // Snoopy
  ```

- Usually, **`sizeof(char)`** = 1

- Example
  ```
  void charset(void)
  {
     unsigned char c;      // as a tiny unsigned integer
     for (c=0;c<255;c++) printf("%c",c);
     printf("%c",c);
  }
  ```
  Comments

  1 The type of **`c`** can't be **`signed char`** (since the loop won't terminate) or **`char`** (since it may be **`signed char`**).

  2 **`c<=255`** can't be the termination condition of the loop, since the loop won't terminate.

- Example – Find 2's complement representation of an integer

  Decimal-2-Binary conversion

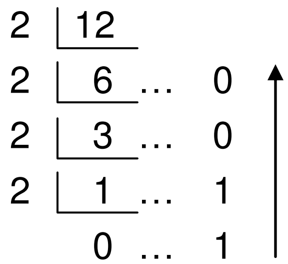
  Given a decimal integer $x$, find its binary equivalence $d_k \cdots d_1 d_0$

  $x = d_k \cdots d_1 d_0 = (d_k \cdots d_1) \times 2 + d_0$ Cf. 9876 = 987*10+6

  $x/2 = d_k \cdots d_1$

  $x\%2 = d_0$

  For example,



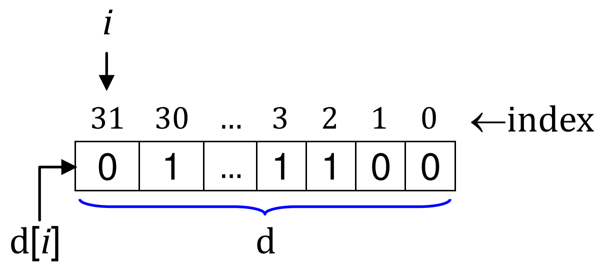

$12_{10} = 1100_2$

Q: How to store the binary digits?
A: A bad view – Treat them as distinct elements
This makes code tedious , e.g.

```
char d31,d30,…,d0;
printf("%d%d…%d",d31,d30,…,d0);
```

A better view
Treat them as components of a single compound element.

`char d[32];` // **d** is an array of 32 characters




Now, a simple loop outputs the binary digits:

```
for (int i=31;i>=0;i--) printf("%d",d[i]);
```

Comments

`char[32]` // array-type declarator
`d[i]` // indexing operator

- Example (Cont'd)

  Primitive and structured data types

  Primitive data type
  1 bool, char, int, etc.
  2 can't be decomposed

  Compound (structured) data type
  1 decomposable
  2 homogeneous: array
  3 heterogeneous: struct, union

  Static, semidynamic, and dynamic arrays

  Static array
  1 Array size is determined at compile time.
  That is, array bounds are *constant expressions* that can be evaluated at compile time, e.g. **`char d[5*6+2];`**
  2 C89, C99, C++

  Semidynamic array (Variable-length array)
  1 Array size is determined and fixed at run time, e.g.
  **`void p(int n) { char d[n]; }`**
  2 C99

  Dynamic array
  1 Array size may vary at run time.
  2 C++ library

  Q: How would you make the code suitable for 64-bit integers?

```
char d[32];
for (int i=31;i>=0;i--) printf("%d",d[i]);
```

  A: Change 32 to 64, and 31 to 63.
  But, this is cumbersome and error-prone. It would be better if the code is written as

```
char d[bits];
for (int i=bits-1;i>=0;i--) printf("%d",d[i]);
```

  Then, we need only change the value of **`bits`** to 64.

  Q: How to define **`bits`**, then?

- Example (Cont'd)

  Approach 1 – Semidynamic array (C99)

```
int bits=32;
char d[bits];
```

This isn't a typical use of semidynamic arrays. After all, 32 is a constant.

Approach 2 – Static array with macro (C89, C99, C++)

```
#define bits 32
char d[bits];
```

Drawback: Macros aren't subject to the scope rules.

```
void p(void)
{
   #define bits 32
   char d[bits];
}
void q(void) { int bits; }   ☹ int 32;
```

Approach 3 – Static array with **`const`** variable ( C++)

```
const int bits=32;
char d[bits];                ①
```

Comment

**`const`** variables occupy storage and are subject to the scope rules. (Macros have no storage.)

Comment

**`const`** means "unmodifiable". It doesn't mean that the values of **`const`** variables must be known by the compiler, e.g.

```
void p(int n)
{
   const int bits=n;
   char d[bits];             ②
}
```

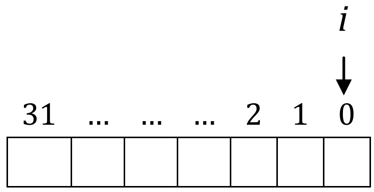
- Example (Cont'd)

  In C, all **`const`** variables can't be used in constant expressions.
  Thus, in C89, both ① and ② are illegal.
  However, both are legal in C99, as they as treated as semi-dynamic arrays, i.e. **`bits`** is a variable, not a constant.

  In C++, **`const`** variables initialized with constant expressions can be used in constant expressions.
  Thus, in C++, ① is legal, but ② is illegal.

  Decimal-2-Binary conversion (implementation)

  Version A – Chars as tiny integers




```
#include <stdlib.h>
void twoscomp(int n)
{
    const int bits=8*sizeof(int);
    char a[bits];          // either signed or unsigned is ok
    unsigned m=abs(n);                                      ①
    int i=0;                                                ②
    while (m>0) {
        a[i]=m%2; m=m/2; i++;
    }
    for (;i<bits;i++) a[i]=0;
    if (n<0) {                                              ③
        for (int i=0;i<bits;i++) a[i]=1-a[i];               ④
        int i=0;
        while (a[i]==1) { a[i]=0; i++; }                    ⑤
        a[i]=1;
    }
    for (int i=bits-1;i>=0;i--) printf("%d",a[i]);
    printf("\n");
}
```

② As written, there are 4 distinct variables, all named **`i`**.
Alternatively, we may simply retain the first one.
Q: Which is better?
A: 4 is better – each controls its own loop with limited scope
Also, compilers usually optimize the space by letting the last 3 variables share the same storage.

- Example (Cont'd)

③ or,
```
if (n<0) {                        //   xxxx1000
      int i=0;                    //   x̄x̄x̄x̄0111
      while (a[i]==0) i++;        // +         1
      for (i++;i<bits;i++)        //   x̄x̄x̄x̄1000
         a[i]=1-a[i];
}
```

④ char → int, integral promotion

`a[i] = 1 - a[i];`

int → char, integral conversion

or

`a[i] = a[i] == 0? 1: 0`

⑤ Q: How can one be sure that there is a 0?
A: If all are 1's, then **n** = 0

① In **<stdlib.h>**, the function **abs** is defined as

```
int abs(int n) { return n<0? -n: n; }
```

But, the return type ought to be **unsigned**.

So, the calls to **abs** should be used in **unsigned** context.

```
printf("%u\n",abs(INT_MIN));    // 2147483648
printf("%d\n",abs(INT_MIN));    // -2147483648
```

As another example, suppose we write

```
int m=abs(n);
```

If **n** = **INT_MIN**, **m** = **INT_MIN** < 0 and the conversion fails.

Final remarks

**<stdlib.h>** also includes

```
long int labs(long int);
```

In **<math.h>**, there are

```
double fabs(double);
float fabsf(float);                // C99 only
long double fabsl(long double);    // C99 only
```

- Example (Cont'd)

Version B – Chars as characters

```
void twoscomp(int n)
{
   const int bits=8*sizeof(int);
   char a[bits];
   unsigned m=abs(n);
   int i=0;
   while (m>0) {
      a[i]='0'+m%2; m=m/2; i++;
   }
   for(;i<bits;i++) a[i]='0';
   if (n<0) {
      for(int i=0;i<bits;i++) a[i]='0'+'1'-a[i]; ①
      int i=0;
      while (a[i]=='1') { a[i]='0'; i++; }
      a[i]='1';
   }
   for(int i=bits-1;i>=0;i--) printf("%c",a[i]);
   printf("\n");
}
```

① or, `a[i]=a[i]=='0'?'1':'0';`

Another algorithm

1 If $n \geq 0$, signbit = 0.
  If $n < 0$, signbit = 1 and $n = n - \mathrm{INT\_MIN}$

2 Convert $n$ (now $\geq 0$) to binary as usual

Reason:

```
    1 ? ?  ...  ?      n < 0
+   0 d d  ...  d     -n > 0
  -----------------
   1 0 0 0 ...  0
```

Thus,

```
??...? + dd...d = 100...0
??...? + -n     = -INT_MIN
??...? = n-INT_MIN
```

# Floating types

- Floating-point vs fixed-point numbers
  Suppose real numbers have 4 decimal digits
  1. floating-point representation
     0.1234, 1.234, 12.34, 123.4, 1234.0 are representable.
  2. fixed-point representation
     With 2 decimal places, only 12.34 is representable.
     With 3 decimal places, only 1.234 is representable.

- Three floating types
  **`float`**
  **`double`**
  **`long double`**

- sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

- Typical sizes on 32-bit machines
  sizeof(float) = 4,
  sizeof(double) = 8,
  sizeof(long double) = 8/10

- Header file <float.h> contains symbolic constants for the sizes of floating types.
  **`FLT_MAX  FLT_MIN`**
  **`DBL_MAX  DBL_MIN`** etc

- Floating constants

| | | |
|---|---|---|
| Suffix | **`f`** | float |
| | **`l`** | long double |

  N.B. Both upper- and lower-case letters are allowed, e.g.

| | | |
|---|---|---|
| float | 3.14F | 0.314e1F |
| double | 3.14 | 0.314e1 |
| long double | 3.14L | 0.314e1L |

- IEEE 754 single precision format

| *R* | 1-bit sign *s* | 8-bit exponent *e* | 23-bit fraction *f* |
|---|---|---|---|

*R* = $(-1)^s 1.f \times 2^{e-127}$, where $1 \le e \le 254$ is in excess-127.

*R* = 0, if $e = 0$ and $f = 0$

*R* = $\pm\infty$, if $e = 255$, $f = 0$, and $s = 0,1$

*R* = NaN (Not a Number), if $e = 255$ and $f \neq 0$

- IEEE 754 double precision format

| *R* | 1-bit sign *s* | 11-bit exponent *e* | 52-bit fraction *f* |
|---|---|---|---|

*R* = $(-1)^s 1.f \times 2^{e-1023}$, where $1 \le e \le 2046$ is in excess-1023

*R* = 0, if $e = 0$ and $f = 0$

*R* = $\pm\infty$, if $e = 2047$, $f = 0$, and $s = 0,1$

*R* = NaN, if $e = 2047$ and $f \neq 0$

- Range and precision

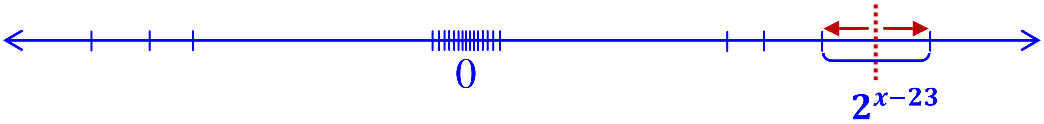
| | Single | Double |
|---|---|---|
| Positive max | $2^{128} \approx 3.4 \cdot 10^{38}$ | $2^{1024} \approx 1.8 \cdot 10^{308}$ |
| Positive min | $2^{-126} \approx 1.2 \cdot 10^{-38}$ | $2^{-1022} \approx 2.2 \cdot 10^{-308}$ |
| Precision | $24/\log_2 10 \approx 7.22$ | $53/\log_2 10 \approx 15.95$ |

$\because 2^k = 10 \Rightarrow k = \log_2 10$

- Estimation of real numbers

For single precision, the difference between two consecutive floating numbers is, e.g.

$1.0 \ldots 01 \times 2^x - 1.0 \ldots 00 \times 2^x = 2^{x-23}, \; -126 \le x \le 127$




Observe that large floating numbers are sparse, whereas small floating numbers are dense.

Also, the error of estimation $\le \frac{1}{2} \times \mathbf{2^{x-23}}$

- Decimal-2-Binary conversion (fraction)

Given a real number $x < 1$, find its binary equivalence $0.d_{-1}d_{-2}\ldots$

$x = 0.d_{-1}d_{-2} \cdots < 1$ Cf. $0.234 \times 10 = 2.34$

$2x = d_{-1}.d_{-2} \cdots < 2$

- Example

$0.1 \times 2 = 0.2$
$0.2 \times 2 = 0.4$ 3 bits = 1 octal digit
$0.4 \times 2 = 0.8$ 4 bits = 1 hexadecimal digit
$0.8 \times 2 = 1.6$
$0.6 \times 2 = 1.2$ normalization
$0.2 \times 2 =$

$$0.1_{10} = 0.0\overline{0011} = 0.0001100110011\cdots = 1.1\overline{0011} \times 2^{-4}$$

Single precision representation

s = $\underbrace{0}_{\text{sign}} \; \underbrace{01111011}_{\text{exponent}} \; \underbrace{10011001100110011001101}_{\text{fraction}}$ = 3dcccccd

Observe that s is an approximation of 0.1, and that s > 0.1

Double precision representation

d = $\underbrace{0}_{\text{sign}} \; \underbrace{01111111011}_{\text{exponent}} \; \underbrace{1001100110011001100\cdots0011010}_{\text{fraction}}$

= 3fb999999999999a

Observe that d is an approximation of 0.1, and that d > 0.1

- Example (Cont'd)

## On printf

1. printf is a variable-argument function
2. For variable-argument function calls, the compiler performs *default argument promotions* on the arguments.
   a) **`float`** arguments are converted to **`double`**
   b) integral promotions are performed on the arguments
3. Assume that sizeof(double)=8, sizeof(int)=4

| | |
|---|---|
| %f %e %g | expect 8-byte double |
| %d | expect 4-byte int |
| %o %x | expect 4-byte unsigned int |

```
printf("%f",0.1);        // 0.100000
printf("%x%x",0.1);
```

Machine byte order

| | | |
|---|---|---|
| 3fb99999 | 9999999a | Byte 0 → byte 7 |
| 9a999999 | 9999b93f | Byte 7 ← byte 0 |

```
printf("%f",0.1f);       // 0.100000
printf("%x%x",0.1f);     // float → double
```

| | |
|---|---|
| 3fb99999 | a0000000 |
| 000000a0 | 9999b93f |

Q: How to display the internal representation of 0.1f ?

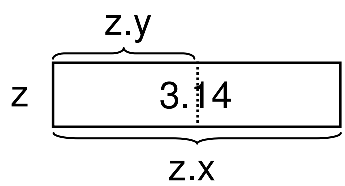
## Heterogeneous data types

```
struct { double x; int y; } z={3.14,7};
printf("%d",z.y);        // member access operator
```

$$\underbrace{\begin{array}{|c:c|}\multicolumn{1}{c}{x} & \multicolumn{1}{c}{y} \\ \hline 3.14 & 7 \\ \hline\end{array}}_{z}$$

- Example (Cont'd)

```
union { double x; int y; } z = {3.14};
printf("%d",z.y);   // meaningless
```




Comment
Only the 1st member of a union can be initialized, e.g.

```
union { double x; int y; } z = {3.14,7};  // no
```

The internal representation of 0.1f can be displayed as follows:

```
void show01(void)
{
   union { float x; int y; } a={0.1f};
   printf("%x\n",a.y);
   union { double x; int y[2];} b={0.1};
   printf("%x %x\n",b.y[0],b.y[1]);
}
```

- Error propagation

$$\begin{array}{rl} & 1.xxx\ldots xxx \times 2^3 \\ + & 1.yyy\ldots yyy \times 2^5 \end{array} \quad \Rightarrow \quad \begin{array}{rl} & 1.xxx\ldots xxx \times 2^3 \\ + & 1yy.y\ldots yyy00 \times 2^3 \end{array}$$

$\Downarrow$ No! The MSDs can't be discarded.

$$\begin{array}{rl} & 0.01xxx\ldots.xxx \times 2^5 \\ + & \underline{1.yyy\ldots yyy \quad \times 2^5} \\ & 10.0zz\ldots zzz\text{, say} \end{array}$$

The blue-colored LSDs are rounded.

Due to error propagation, the following ways of summing up a series are likely to yield different results.

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \frac{1}{8} + \cdots$$

$$\left(1 - \frac{1}{2}\right) + \left(\frac{1}{3} - \frac{1}{4}\right) + \left(\frac{1}{5} - \frac{1}{6}\right) + \left(\frac{1}{7} - \frac{1}{8}\right) + \cdots$$

$$1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \cdots - \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right)$$

- Example – Calculate 0.1 + 0.2 + . . . + 1.0

Incorrect method

```
float sum=0.0f;
for (float d=0.1f;d<=1.0f;d+=0.1f)       ①
   sum+=d;
printf("%f",sum);        // 4.500000

double sum=0.0;
for (double d=0.1;d<=1.0;d+=0.1)         ②
   sum+=d;
printf("%f",sum);        // 5.500000
```

① How about using `d!=1.1f` ?
② How about using `d!=1.1` ?

Lesson: Never test floating numbers for (in)equality.

Correct methods

```
int sum=0;
for (int d=1;d<=10;d++) sum+=d;
printf("%f",sum/10.f);
```

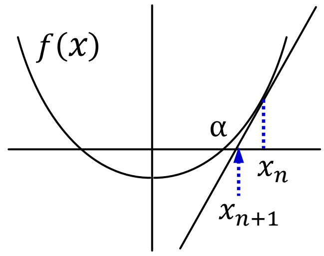
or

```
float sum=0.f;
for (float d=1.f;d<=10.f;d++) sum+=d;
printf("%f",sum/10.f);
```

or

```
float sum=0.f,epsilon=.05f;
for (float d=.1f;d<=1.f+epsilon;d+=.1f)
   sum+=d;
printf("%f",sum);
```

- Example – Square root




*Newton's method for rootfinding*

Formula $\quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \quad n \geq 0$

Error test 1 $\quad |f(x_n)| \leq \varepsilon$, poor if $f'(\alpha) \to 0, \infty$

Error test 2 $\quad |x_n - x_{n+1}| \leq \varepsilon \Rightarrow |\alpha - x_n| \leq \varepsilon$

By the mean value theorem, $\exists z_n \in [\alpha, x_n]$ for which

$$f(x_n) - f(\alpha) = f'(z_n)(x_n - \alpha)$$

Thus,

$$x_{n+1} - x_n = -\frac{f(x_n)}{f'(x_n)} \approx -\frac{f(x_n)}{f'(z_n)} = x_n - \alpha \quad \text{as} \quad x_n \to \alpha$$

*Newton's method for finding square root*

To compute $\sqrt{a}$, let $f(x) = x^2 - a$, then

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \quad n \geq 0$$

Observe that

1) $x_{n+1}^2 - a = \left(\frac{x_n^2 - a}{2x_n}\right)^2 \quad n \geq 0$ and thus $x_n \geq \sqrt{a},\ n \geq 1$

2) $x_{n+1} - x_n = -\frac{x_n^2 - a}{2x_n} \leq 0 \quad n \geq 1$ and thus $x_n \geq x_{n+1},\ n \geq 1$

3) Error test $\quad x_n - x_{n+1} \leq \varepsilon,\ n \geq 1$
or,
let $x_{-1} = 0, x_0 = (1 + a)/2$, then $x_n - x_{n+1} \leq \varepsilon,\ n \geq 0$

- Example (Cont'd)

Version 1 – Pre-test loop

```
double sqrt(double a)
{
   const double epsilon=1e-15;
   double x=(1+a)/2,x1=(x+a/x)/2;
   while (x-x1>epsilon) {
      x=x1;
      x1=(x+a/x)/2;
   }
   return x1;
}
```

Version 2 – Middle-test loop

```
double sqrt(double a)
{
   const double epsilon=1e-15;
   double x=(1+a)/2,x1;
   while (true) {
      x1=(x+a/x)/2;
      if (x-x1<=epsilon) break;
      x=x1;
   }
   return x1;
}
```

Version 3 – Post-test loop

```
double sqrt(double a)
{
   const double epsilon=1e-15;
   double x,x1=(1+a)/2;
   do {
      x=x1;
      x1=(x+a/x)/2;
   } while (x-x1>epsilon);
   return x1;
}
```

- Example (Cont'd)

  Newton's method converges very rapidly.
  For example, `sqrt(2000)` produces the following sequence.

```
501.249500249875
252.619764582810
130.268400745618
 72.810659086329
 50.139582364779
 45.014113668459
 44.722311528907
 44.721359560128
 44.721359549996
 44.721359549996
```

  The output is formatted by `printf("%16.12f\n",x1);`

  N.B. `printf("%f\n",x1);` ≡ `printf("%.6f\n",x1);`

## Type conversions

- Conversion to unsigned type

  As mentioned before, the results of unsigned integer overflows are well-defined in C/C++.

```
unsigned short x=-1;
unsigned short x=65536;
```

  How can these be possible? Well …

  8 hours before 5 p.m. = 5 p.m. – 8 = −3 = 9 a.m.
  8 hours after 5 p.m. = 5 p.m. + 8 = 13 = 1 a.m.

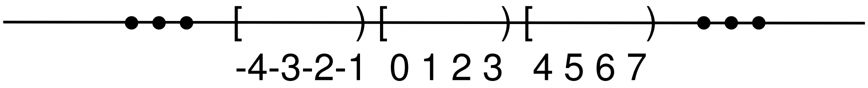
  −3 mod 12 = 9 mod 12
  i.e. $-3 \equiv 9 \pmod{12}$ −3 and 9 are congruent modulo 12

  13 mod 12 = 1 mod 12
  i.e. $13 \equiv 1 \pmod{12}$ 13 and 1 are congruent modulo 12

- Congruence modulo $n$

For $n = 4$, there are 4 equivalence classes.

[0] = { ..., −4, 0, 4, ... }    // 0 is the representative.
[1] = { ..., −3, 1, 5, ... }    // 1 "
[2] = { ..., −2, 2, 6, ... }    // 2 "
[3] = { ..., −1, 3, 7, ... }    // 3 "




- Conversion to unsigned type (Cont'd)

In general, the conversion

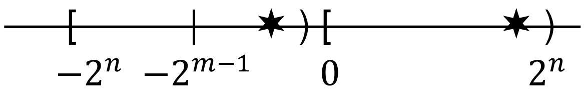
$m$-bit (un)signed $\rightarrow$ $n$-bit unsigned

uses congruence modulo $2^n$ arithmetic:

$$x \rightarrow x \bmod 2^n$$

Case 1: $m \leq n$

$x \geq 0$    $x \rightarrow x$
$x < 0$    $x \rightarrow x + 2^n$




```
short a=-1;
unsigned short b=a;       // b = -1 + 2^16 = 65535
unsigned c=a;             // c = -1 + 2^32 = UINT_MAX
```

Machine code for $m = n$

$x \geq 0$    $x \rightarrow x$    copy
$x < 0$    $x \rightarrow x + 2^n$    copy

| | | signed | unsigned |
|---|---|---|---|
| e.g. $x = -3$ | 11…1101 | $-3$ | $2^n - 3$ |
| | + 00…0011 | $3$ | $3$ |
| | 100…0000 | $0$ | $2^n$ |

$-3$ and $2^n - 3$ have the same bit pattern.

- Conversion to unsigned type (Cont'd)

Machine code for $m < n$

$m$-bit unsigned $\xrightarrow{\text{zero-filled}}$ $n$-bit unsigned

$m$-bit int $\xrightarrow{\text{sign extension}}$ $n$-bit int $\xrightarrow{\text{copy}}$ $n$-bit unsigned

sign extension

```
short a=-1;
unsigned short b=a;
unsigned c=a;
printf("%hX\n",a)†;        // FFFF
printf("%hX\n",b);         // FFFF
printf("%X\n",c);          // FFFFFFFF
```

```
†printf("%hX\n",a);        // short → int
```

| FFFF | FFFF |
|---|---|

Case 2: $m > n$

$x \to x \bmod 2^n$

```
signed a=-456789;
unsigned short b=a;        // b = −456789 mod 2^16 = 1963
unsigned c=456789
unsigned short d=c;        // d = 456789 mod 2^16 = 63573
```

Machine code for $m > n$

$x$

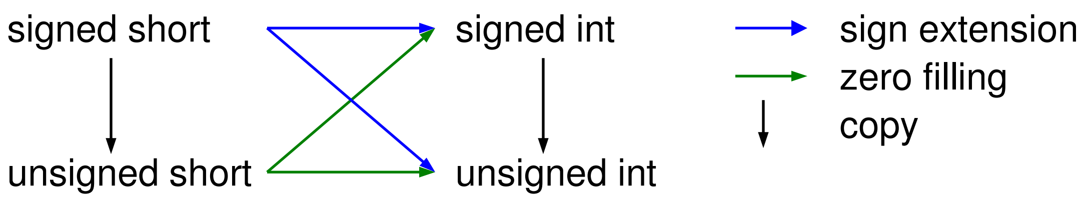
| $(m-n)$-bit $y$ | $n$-bit $z$ |
|---|---|

$x = y \times 2^n + z$

$x \bmod 2^n = z$ so, left truncation

- Widening – Safe†

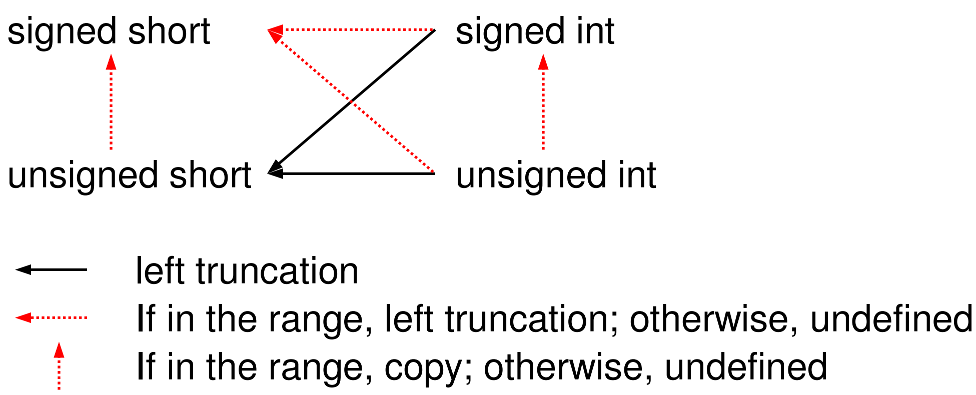
Assume that sizeof(short) = 2 and sizeof(int) = 4




integral → floating        possible loss of significant digits

float → double → long double        value unchanged

- Narrowing – Possibly unsafe‡




integral ← floating
If in the range, discard fractional part; otherwise, undefined

float ← double ← long double
If in the range, round fractional part; otherwise, undefined

---

† If the destination type is unsigned, the resulting value is the least unsigned integer congruent to the source integer (modulo $2^n$ where n is the number of bits used to represent the unsigned type).
‡ If the destination type is signed, the value is unchanged if it can be represented in the destination type; otherwise, the value is implementation-defined.

## Assignment conversions

- Take place in three situations
  1 Assignment
  2 Parameter passing
  3 Function value returning
- Involve both widening and narrowing

## Arithmetic conversions

- Take place in arithmetic expressions
- Involve only widening
- Integral promotion
  If an integral expression① has an operand of type② smaller than integer type, the operand is first promoted to integer type③.

  ① For `'a'+3.14`, char is converted to double immediately
  ② bool, char, short, etc
  ③ Usually, promoted to int; but if sizeof(short) = sizeof(int), unsigned short will be promoted to unsigned int.

- Example

| | |
|---|---|
| `float f=3.14;` | ① double → float |
| `int x=f;` | ② float → int |
| `int y=x+f;` | ③ First, int → float, and then, float → int |

For narrowing, the compiler may warn you of 'possible loss of data'. To suppress the warning messages, do these:

```
float f=3.14f;
int x=(int)f;
int y=x+(int)f;
```

The last case also saves one conversion.