



Basic Types and Formatted I/O



C Variables Names (1)

Variable Names

- ◆ Names may contain letters, digits and underscores
- ◆ The first character must be a letter or an underscore.
 - the underscore can be used but **watch out!!**
- ◆ Case matters!
- ◆ C keywords cannot be used as variable names.

present, hello, y2x3, r2d3, ...

/* OK */

_1993_tar_return

/* OK but don't */

Hello#there

/* illegal */

double

/* shouldn't work */

2fartogo

/* illegal */

C Variables Names (2)

Suggestions regarding variable names

- ◆ DO: use variable names that are descriptive
- ◆ DO: adopt and stick to a standard naming convention
 - sometimes it is useful to do this consistently for the entire software development site
- ◆ AVOID: variable names starting with an underscore
 - often used by the operating system and easy to miss
- ◆ AVOID: using uppercase only variable names
 - generally these are pre-processor macros (later)

C Basic Types (1)

- ◆ There are only a few basic data types in C
 - char: a single byte, capable of holding one character
 - int: an integer of fixed length, typically reflecting the natural size of integers on the host machine (i.e., 32 or 64 bits)
 - float: single-precision floating point
 - double: double precision floating point

C Basic Types (2)

- ◆ There are a number of qualifiers which can be applied to the basic types
 - length of data
 - ❖ **short int**:
 - ❖ "shorter" int, \leq number of bits in an int
 - ❖ can also just write "**short**"
 - ❖ **long int**:
 - ❖ a "longer int", \geq number of bits in an int
 - ❖ often the same number of bits as an int
 - ❖ can also just write "**long**"
 - ❖ **long double**
 - ❖ generally extended precision floating point
 - signed and unsigned
 - ❖ **unsigned int**
 - ❖ an int type with no sign
 - ❖ if int has 32-bits, range from $0..2^{32}-1$
 - ❖ also works with **long** and **short**
 - ❖ **unsigned char**
 - ❖ a number from 0 to 255
 - ❖ **signed char**
 - ❖ a number from -128 to 127 (8-bit signed value)
 - ❖ very similar to byte in Java

C Basic Types (3)

- ◆ All types have a fixed size associated with them
 - this size can be determined at compile time
- ◆ Example storage requirements

<u>DATA</u>	<u>Bytes Required</u>
The letter x (char)	1
The number 100 (int)	4
The number 120.145 (double)	8

- ◆ These numbers are highly variable between C compilers and computer architectures.
- ◆ Programs that rely on these figures must be very careful to make their code portable (ie. Try not to avoid relying on the size of the predefined types)

C Basic Types (4)

Numeric Variable Types

◆ Integer Types:

- Generally 32-bits or 64-bits in length
- Suppose an int has b-bits
 - ❖ a signed int is in range $-2^{b-1}..2^{b-1}-1$
 - -32768 .. 32767 (32767+1=-32768)
 - ❖ an unsigned int is in range $0..2^b-1$
 - 0 .. 65535 (65535+1=0)
 - ❖ no error message is given on this "overflow"

◆ Floating-point Types:

- Generally IEEE 754 floating point numbers
 - ❖ float (IEEE single): 8 bits exponent, 1-bit sign, 23 bits mantissa
 - ❖ double (IEEE double): 10 bits exponent, 1-bit sign, 53 bits mantissa
 - ❖ long double (IEEE extended)
- Only use floating point types when really required
 - ❖ they do a lot of rounding which must be understood well
 - ❖ floating point operations tend to cost more than integer operations

C Basic Types (5)

A typical 32-bit machine

Type	Keyword	Bytes	Range
character	char	1	-128...127
integer	int	4	-2,147,483,648...2,147,438,647
short integer	short	2	-32768...32367
long integer	long	4	-2,147,483,648...2,147,438,647
long long integer	long long	8	-9223372036854775808 ... 9223372036854775807
unsigned character	unsigned char	1	0...255
unsigned integer	unsigned int	2	0...4,294,967,295
unsigned short integer	unsigned short	2	0...65535
unsigned long integer	unsigned long	4	0...4,294,967,295
single-precision	float	4	1.2E-38...3.4E38
double-precision	double	8	2.2E-308...1.8E308

Formatted Printing with printf (1)

- ◆ The `printf` function is used to output information (both data from variables and text) to standard output.
 - A C library function in the `<stdio.h>` library.
 - Takes a format string and parameters for output.
- ◆ `printf(format string, arg1, arg2, ...);`
 - e.g. `printf("The result is %d and %d\n", a, b);`
- ◆ The format string contains:
 - Literal text: is printed as is without variation
 - Escaped sequences: special characters preceded by `\`
 - Conversion specifiers: `%` followed by a single character
 - ❖ Indicates (usually) that a variable is to be printed at this location in the output stream.
 - ❖ The variables to be printed must appear in the parameters to `printf` following the format string, in the order that they appear in the format string.

Formatted Printing with printf (2)

◆ Conversion Specifiers

Specifier	Meaning
<code>%c</code>	Single character
<code>%d</code>	Signed decimal integer
<code>%x</code>	Hexadecimal number
<code>%f</code>	Decimal floating point number
<code>%e</code>	Floating point in “scientific notation”
<code>%s</code>	Character string (more on this later)
<code>%u</code>	Unsigned decimal integer
<code>%%</code>	Just print a % sign
<code>%ld, %lld</code>	long, and long long

- ◆ There must be one conversion specifier for each argument being printed out.
- ◆ Ensure you use the correct specifier for the type of data you are printing.

Formatted Printing with printf (3)

◆ Escape Sequences:

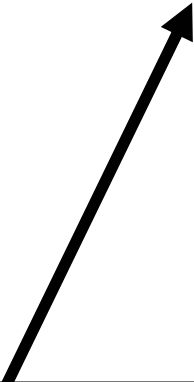
Sequence	Meaning
<code>\a</code>	Bell (alert)
<code>\b</code>	Backspace
<code>\n</code>	Newline
<code>\t</code>	Horizontal tab
<code>\\</code>	Backslash
<code>\'</code>	Single quote
<code>\"</code>	Double quotation
<code>\xhh</code>	ASCII char specified by hex digits <i>hh</i>
<code>\ooo</code>	ASCII char specified by octal digits <i>ooo</i>

Formatted Printing with printf (4)

- ◆ An example use of `printf`

```
#include <stdio.h>
int main() {
    int ten=10,x=42;
    char ch1='o', ch2='f';
    printf("%d%% %c%c %d is %f\n",
           ten,ch1,ch2,x, 1.0*x / ten );
    return 0;
}
```

- ◆ What is the output?



Why do we need to do this?
We will talk about a better way later.
(Type conversion)

Reading Numeric Data with scanf (1)

- ◆ The `scanf` function is the input equivalent of `printf`
 - A C library function in the `<stdio.h>` library
 - Takes a format string and parameters, much like `printf`
 - The format string specifiers are nearly the same as those used in `printf`
- ◆ Examples:

```
scanf ("%d", &x);    /* reads a decimal integer */
scanf ("%f", &rate); /* reads a floating point value */
```
- ◆ The ampersand (&) is used to get the “address” of the variable
 - All the C function parameters are “passed by value”.
 - If we used `scanf("%d",x)` instead, the value of `x` is passed. As a result, `scanf` will not know where to put the number it reads.
 - More about this in Section 15 (“Functions”)

Reading Numeric Data with scanf (2)

- ◆ Reading more than one variable at a time:

- For example:

```
int n1, n2; float f;  
scanf("%d%d%f",&n1,&n2,&f);
```

- Use white spaces to separate numbers when input.

```
5 10 20.3
```

- ◆ In the format string:

- You can use other characters to separate the numbers

```
scanf("value=%d,ratio=%f", &value,&ratio);
```

- ❖ You must provide input like:

```
value=27,ratio=0.8
```

- `scanf` returns an int

- ❖ If end-of-file was reached, it returns EOF, a constant defined in `<stdio.h>`

- ❖ Otherwise, it returns the number of input values correctly read from standard input.

Reading Numeric Data with scanf (3)

◆ One tricky point:

- If you are reading into a **long** or a **double**, you must precede the conversion specifier with an **l** (a lower case L)
- Example:

```
int main() {  
    int x;  
    long y;  
    float a;  
    double b;  
    scanf("%d %ld %f %lf", &x, &y, &a, &b);  
    return 0;  
}
```

Type Conversion

- ◆ C allows for conversions between the basic types, implicitly or explicitly.
- ◆ Explicit conversion uses the cast operator.
- ◆ Example 1:

```
int x=10;
float y,z=3.14;
y=(float) x;    /* y=10.0 */
x=(int) z;      /* x=3    */
x=(int) (-z);   /* x=-3   -- rounded approaching zero */
```

- ◆ Example 2:

```
int i;
short int j=1000;
i=j*j;          /* wrong!!! */
i=(int)j * (int)j; /* correct */
```


Implicit Conversion

- ◆ If the compiler expects one type at a position, but another type is provided, then implicit conversion occurs.
- ◆ Conversion during assignments:

```
char c='a';
```

```
int i;
```

```
i=c; /* i is assigned the ASCII code of 'a' */
```

- ◆ Arithmetic conversion – if two operands of a binary operator are not the same type, implicit conversion occurs:

```
int i=5 , j=1;
```

```
float x=1.0 , y;
```

```
y = x / i; /* y = 1.0 / 5.0 */
```

```
y = j / i; /* y = 1 / 5 so y = 0 */
```

```
y = (float) j / i; /* y = 1.0 / 5 */
```

```
/* The cast operator has a higher precedence */
```

Example

The `sizeof()` function returns the number of bytes in a data type.

```
int main() {  
    printf("Size of char ..... = %2d byte(s)\n", sizeof(char));  
    printf("Size of short ..... = %2d byte(s)\n", sizeof(short));  
    printf("Size of int ..... = %2d byte(s)\n", sizeof(int));  
    printf("Size of long long ..... = %2d byte(s)\n", sizeof(long long));  
    printf("Size of long ..... = %2d byte(s)\n", sizeof(long));  
    printf("Size of unsigned char. = %2d byte(s)\n", sizeof (unsigned char));  
    printf("Size of unsigned int.. = %2d byte(s)\n", sizeof (unsigned int));  
    printf("Size of unsigned short = %2d byte(s)\n", sizeof (unsigned short));  
    printf("Size of unsigned long. = %2d byte(s)\n", sizeof (unsigned long));  
    printf("Size of float ..... = %2d byte(s)\n", sizeof(float));  
    printf("Size of double ..... = %2d byte(s)\n", sizeof(double));  
    printf("Size of long double .. = %2d byte(s)\n", sizeof(long double));  
    return 0;  
}
```

Example

Results of a previous run of this code on obelix ...

Size of char	= 1 byte(s)
Size of short	= 2 byte(s)
Size of int	= 4 byte(s)
Size of long	= 4 byte(s)
Size of long long	= 8 byte(s)
Size of unsigned char	= 1 byte(s)
Size of unsigned int	= 4 byte(s)
Size of unsigned short	= 2 byte(s)
Size of unsigned long	= 4 byte(s)
Size of float	= 4 byte(s)
Size of double	= 8 byte(s)
Size of long double	=16 byte(s)

Creating Simple Types

- ◆ `typedef` creates a new name for an existing type
 - Allows you to create a new name for a complex old name
- ◆ Generic syntax

```
typedef oldtype newtype;
```
- ◆ Examples:

```
typedef long int32; /* suppose we know an int has 32-bits */
typedef unsigned char byte; /* create a byte type */
typedef long double extended;
```
- ◆ These are often used with complex data types
 - Simplifies syntax!

Variable Declaration (1)

- ◆ Generic Form

```
typename varname1, varname2, ...;
```

- ◆ Examples:

```
int count;
```

```
float a;
```

```
double percent, total;
```

```
unsigned char x,y,z;
```

```
long int aLongInt;
```

```
long AnotherLongInt
```

```
unsigned long a_1, a_2, a_3;
```

```
unsigned long int b_1, b_2, b_3;
```

```
typedef long int32;
```

```
int32 n;
```

- ◆ Where declarations appear affects their scope and visibility

- Rules are similar to those in Java

- Declaration outside of any function are for global variables

- ❖ e.g., just before the main routine

Variable Declaration (2)

Initialization

- ◆ ALWAYS initialize a variable before using it
 - Failure to do so in C is asking for trouble
 - The value of an uninitialized variables is undefined in the C standards

- ◆ Examples:

```
int count;           /* Set aside storage space for count */  
count = 0;          /* Store 0 in count */
```

- ◆ This can be done at definition:

```
int count = 0;  
double percent = 10.0, rate = 0.56;
```

- ◆ Warning: be careful about “out of range errors”

```
unsigned int value = -2500;
```

- The C compiler does not detect this as an error
 - ❖ What do you suspect it does?

Constants (1)

Constants

◆ You can also declare variables as being constants

- Use the `const` qualifier:

```
const double pi=3.1415926;
```

```
const int maxlength=2356;
```

```
const int val=(3*7+6)*5;
```

Note: simple computed values are allowed

- must be able to evaluate at *compile time*

- Constants are useful for a number of reasons

- ❖ Tells the reader of the code that a value does not change
 - Makes reading large pieces of code easier
- ❖ Tells the compiler that a value does not change
 - The compiler can potentially compile faster code

◆ Use constants whenever appropriate

◆ NOTE: You will get errors with the `cc` compiler --- use the `gcc` compiler (newer)

Constants (2)

Preprocessor Constants

- ◆ These are an older form of constant which you still see
 - There is a potential for problems, so be careful using them!
- ◆ Generic Form:
 - `#define CONSTNAME literal`
 - ❖ Generally make pre-processor constants all upper case (convention).
- ◆ Example:
 - `#define PI 3.14159`
- ◆ What really happens
 - The C preprocessor runs before the compiler.
 - Every time it sees the token PI, it substitutes the value 3.14159.
 - The compiler is then run with this “pre-processed” C code.
- ◆ Why this is dangerous?
 - Hard to determine the value of a multiply-defined constant (which you are allowed to create)

Constants (3)

- ◆ An example of constants in use:

```
#include <stdio.h>
#define GRAMS_PER_POUND 454
const int FARFARAWAY = 3000;
int main ()
{
    int weight_in_grams, weight_in_pounds;
    int year_of_birth, age_in_3000;
    printf ("Enter your weight in pounds: ");
    scanf ("%d", &weight_in_pounds);
    printf ("Enter your year of birth: ");
    scanf ("%d", &year_of_birth);
    weight_in_grams = weight_in_pounds * GRAMS_PER_POUND;
    age_in_3000 = FARFARAWAY - year_of_birth;
    printf ("Your weight in grams = %d\n", weight_in_grams);
    printf ("In 3000 you will be %d years old\n", age_in_3000);
    return 0;
}
```

Literals in C

- ◆ Literals are representations of values of some types
 - C allows literal values for integer, character, and floating point types
- ◆ Integer literals
 - Just write the integer in base 10
 - `int y=-46;`
 - We will discuss base 8 and base 16 literals later
- ◆ Character literals
 - Character literals are specified with a single character in single quotes
 - `char ch='a';`
 - Special characters are specified with escape characters
 - ❖ Recall the discussion of ‘escaped’ characters with the shell
 - ❖ We will discuss these later
- ◆ Floating point literals
 - Just write the floating point number
 - `float PI=3.14159;`
 - Can also use mantissa/exponent (scientific) notation
 - `double minusPItimes100 = -3.14159e2`