Programming in C

Based on the Original Slides from
Politehnica International- Computer Engineering
Lecture Slides
Variables

- Programs can use symbolic names for storing computation data
- Variable: a symbolic name for a memory location
  - programmer doesn't have to worry about specifying (or even knowing) the value of the location's address
- In C, variables have to be **declared** before they are used
  - Declarations specify the interpretation given to each identifier
  - Declarations that reserve storage are called *definitions*
- The declaration reserves memory space for the variable, but doesn’t put any value there
- Values get into the memory location of the variable by **initialization** or **assignment**

```c
int a;       // declaring a variable
int sum, a1,a2; // declaring 3 variables
int x=7;     // declaring and initializing a variable
```
Declaring variables

- Some older languages (FORTRAN, BASIC) allow you to use variables without declaring them.
- Other languages (C, Pascal) impose to declare variables
- Good reasons to declare variables:
  - Putting all the variables in one place makes it easier for a reader to understand the program
  - Thinking about which variables to declare encourages the programmer to do some planning before writing a program (What information does the program need? What must the program have to produce as output? What is the best way to represent the data?)
  - The obligation to declare all variables helps prevent bugs of misspelled variable names.
  - Compiler knows the amount of statically allocated memory needed
  - Compiler can verify that operations done on a variable are allowed by its type (strongly typed languages)
Variable names

Rules for valid variable names (*identifiers*) in C:

- Name must begin with a letter or underscore ( _ ) and can be followed by any combination of letters, underscores, or digits.
- Any name that has special significance to the C compiler (*reserved words*) cannot be used as a variable name.
- Examples of **valid** variable names: Sum, pieceFlag, I, J5x7, Number_of_moves, _sysflag
- Examples of **invalid** variable names: sum$value, 3Spencer, int.
- C is case-sensitive: sum, Sum, and SUM each refer to a different variable!
- Variable names can be as long as you want, although only the first 63 (or 31) characters might be significant. (Anyway, it’s not practical to use variable names that are too long)
- Choice of meaningful variable names can increase the readability of a program
Data types

- **Basic data types in C**: `int`, `float`, `double` and `char`.
- **Data type `int`**: can be used to store integer numbers (values with no decimal places)
- **Data type `float`**: can be used for storing floating-point numbers (values containing decimal places).
- **Data type `double`**: the same as type `float`, only with roughly twice the precision.
- **Data type `char`**: can be used to store a single character, such as the letter ‘a’, the digit character ‘6’, or a semicolon ‘;’. 
Example: Using data types

```c
#include <stdio.h>
int main (void)
{
    int integerVar = 100;
    float floatingVar = 331.79;
    double doubleVar = 8.44e+11;
    char charVar = 'W';
    printf ("integerVar = %d\n", integerVar);
    printf ("floatingVar = %f\n", floatingVar);
    printf ("doubleVar = %e\n", doubleVar);
    printf ("doubleVar = %g\n", doubleVar);
    printf ("charVar = %c\n", charVar);
    return 0;
}
```
The basic data type `int`

- Examples of integer constants: 158, –10, and 0
- No embedded spaces are permitted between the digits, and values larger than 999 cannot be expressed using commas. (The value 12,000 is not a valid integer constant and must be written as 12000.)
- Integer values can be displayed by using the format characters `%i` in the format string of a `printf` call.
- Also the `%d` format characters can be used to display an integer.
- Integers can also be expressed in a base other than decimal (base 10): octal (base 8) or hexa (base 16).
Octal notation for integers

- **Octal notation** (base 8): If the first digit of the integer value is 0, the integer is taken as expressed in *octal* notation. In that case, the remaining digits of the value must be valid base-8 digits and, therefore, must be 0–7.
- Example: Octal value 0177 represents the decimal value 127 \((1 \times 8^2 + 7 \times 8 + 7)\).
- An integer value can be displayed in octal notation by using the format characters \%o or \%#o in the format string of a printf statement.
Hexadecimal notation for integers

- **Hexadecimal notation** (base 16): If an integer constant is preceded by a zero and the letter x (either lowercase or uppercase), the value is taken as being expressed in hexadecimal. Immediately following the letter x are the digits of the hexadecimal value, which can be composed of the digits 0–9 and the letters a–f (or A–F). The letters represent the values 10–15, respectively.

- Example: hexadecimal value 0xA3F represents the decimal value 2623 \((10 \times 16^2 + 3 \times 16 + 15)\).

- The format characters %x, %X, %#x, or %#X display a value in hexadecimal format
Data display vs data storage

- The option to use decimal, octal or hexadecimal notation doesn't affect how the number is actually stored internally!
- When/where to use octal and hexa: to express computer-related values in a more convenient way

```c
int x =16;
printf("%d %#X %#o\n", x,x,x);
```

```
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
```

Program       Display
```
"%d"  "%#X"  "%#o"
```

```
16     0X10   020
```
The floating number type float

- A variable declared to be of type float can be used for storing values containing decimal places.
- Examples of floating-point constants: 3., 125.8, −.0001
- To display a floating-point value at the terminal, the printf conversion characters %f are used.
- Floating-point constants can also be expressed in scientific notation. The value 1.7e4 represents the value $1.7 \times 10^4$.
- The value before the letter e is known as the mantissa, whereas the value that follows is called the exponent. This exponent, which can be preceded by an optional plus or minus sign, represents the power of 10 by which the mantissa is to be multiplied.
- To display a value in scientific notation, the format characters %e should be specified in the printf format string.
- The printf format characters %g can be used to let printf decide whether to display the floating-point value in normal floating-point notation or in scientific notation. This decision is based on the value of the exponent: If it’s less than −4 or greater than 5, %e (scientific notation) format is used; otherwise, %f format is used.
The extended precision type double

- Type `double` is very similar to type `float`, but it is used whenever the range provided by a `float` variable is not sufficient. Variables declared to be of type `double` can store roughly twice as many significant digits as can a variable of type `float`.
- Most computers represent double values using 64 bits.
- Unless told otherwise, all floating-point constants are taken as double values by the C compiler!
- To explicitly express a `float` constant, append either an f or F to the end of the number: 12.5f
- To display a double value, the format characters %f, %e, or %g, which are the same format characters used to display a float value, can be used.
The character type `char`

- A `char` variable can be used to store a single character.
- A character constant is formed by enclosing the character within a pair of single quotation marks. Valid examples: 'a', ';', and '0'.
- Character zero (‘0’) is not the same as the number (integer constant) 0.
- Do not confuse a character constant with a character string: character ‘0’ and string “0”.
- The character constant ‘\n’ — the newline character — is a valid character constant: the backslash character is a special character in the C system and does not actually count as a character.
- There are other special characters (escape sequences) that are initiated with the backslash character: \, "", \t
- The format characters `%c` can be used in a `printf` call to display the value of a `char` variable
- To handle characters internally, the computer uses a numerical code in which certain integers represent certain characters. The most commonly used code is the ASCII code
Assigning values to char

```c
char letter;    /* declare variable letter of type char */

letter = 'A';   /* OK */
letter = A;     /* NO! Compiler thinks A is a variable */
letter = "A";   /* NO! Compiler thinks "A" is a string */
letter = 65;    /* ok because characters are really
                 stored as numeric values (ASCII code),
                 but poor style */
```
Data display vs data storage

/* displays ASCII code for a character */

#include <stdio.h>
int main(void)
{
    char ch;
    ch='A';
    printf("The code for \%c is \%d.\n", ch, ch);
    return 0;
}
Storage sizes and ranges

• Every type has a *range* of values associated with it.
• This range is determined by the amount of storage that is allocated to store a value belonging to that type of data.
• In general, that amount of storage is not defined in the language. It typically depends on the computer you’re running, and is, therefore, called *implementation*- or *machine*-dependent.
  – For example, an integer might take up 32 bits on your computer, or it might be stored in 64. You should never write programs that make any assumptions about the size of your data types!
• The language standards only guarantees that a **minimum** amount of storage will be set aside for each basic data type.
  – For example, it’s guaranteed that an integer value will be stored in a minimum of 32 bits of storage, which is the size of a “word” on many computers.
Integer overflow

• What happens if an integer tries to get a value too big for its type (out of range)?

```c
#include <stdio.h>
int main(void) {
    int i = 2147483647;
    printf("%i %i %i\n", i, i+1, i+2);
    return 0;
}
```

Program output:
2147483647 -2147483648 -2147483647

Explanation:
On this computer, int is stored on 32 bits: the first bit represents the sign, the rest of 31 bits represent the value.
Biggest positive int value here: $2^{31}-1 = 2147483647$
Floating point round-off error

```c
#include <stdio.h>
int main(void)
{
    float a, b;
    b = 2.0e20 + 1.0;
    a = b - 2.0e20;
    printf("%f \n", a);
    return 0;
}
```

Program output:
4008175468544.000000

Explanation: the computer doesn't keep track of enough decimal places! The number 2.0e20 is 2 followed by 20 zeros and by adding 1 you are trying to change the 21st digit. To do this correctly, the program would need to be able to store a 21-digit number. A float number is typically just six or seven digits scaled to bigger or smaller numbers with an exponent.
Type Specifiers: long, long long, short, unsigned, signed

- Type specifiers: extend or limit the range of certain basic types on certain computer systems.
- If the specifier `long` is placed directly before the `int` declaration, the declared integer variable is of extended range on some computer systems.
- Example of a `long int` declaration: `long int factorial;`
- On many systems, an `int` and a `long int` both have the same range and either can be used to store integer values up to 32-bits wide ($2^{31} - 1$, or 2,147,483,647).
- A constant value of type `long int` is formed by optionally appending the letter `L` (upper- or lowercase) at the end of an integer constant.
- Example: `long int numberOfPoints = 131071100L;`
- To display the value of a `long int` using `printf`, the letter `l` is used as a modifier before the integer format characters `d`, `o`, and `x`. 
# Basic Data Types - Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Constants Ex.</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer value; guaranteed to contain <strong>at least 16 bits</strong></td>
<td>12, -7, 0xFFF0, 0177</td>
<td>%i, %d, %x, %o</td>
</tr>
<tr>
<td>short int</td>
<td>Integer value of reduced precision; guaranteed to contain <strong>at least 16 bits</strong></td>
<td>-</td>
<td>%hi, %hx, %ho</td>
</tr>
<tr>
<td>long int</td>
<td>Integer value of extended precision; guaranteed to contain <strong>at least 32 bits</strong></td>
<td>12L, 23L, 0xFFFFL</td>
<td>%li, %lx, %lo</td>
</tr>
<tr>
<td>long long int</td>
<td>Integer value of extraextended precision; guaranteed to contain <strong>at least 64 bits</strong></td>
<td>12LL, 231LL, 0xFFFFLL</td>
<td>%lli, %llx, %llo</td>
</tr>
<tr>
<td>unsigned int</td>
<td>Positive integer value; can store positive values up to twice as large as an int; guaranteed to contain at least 16 bits (all bits represent the value, no sign bit)</td>
<td>12u, 0xFFU</td>
<td>%u, %x, %o</td>
</tr>
<tr>
<td>unsigned short int</td>
<td></td>
<td>-</td>
<td>%hu, %hx, %ho</td>
</tr>
<tr>
<td>unsigned long int</td>
<td></td>
<td>12UL, 100ul, 0xffffUL</td>
<td>%lu, %lx, %lo</td>
</tr>
<tr>
<td>unsigned long long int</td>
<td></td>
<td>12ull, 0xffffeULL</td>
<td>%llu, %llx, %llo</td>
</tr>
</tbody>
</table>
## Basic Data Types - Summary (contd.)

<table>
<thead>
<tr>
<th>Type</th>
<th>Meaning</th>
<th>Constants</th>
<th>printf</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>Floating-point value; a value that can contain decimal places; guaranteed to contain at <strong>least six digits of precision</strong>.</td>
<td>12.34f, 3.1e-5f</td>
<td>%f, %e, %g</td>
</tr>
<tr>
<td>double</td>
<td>Extended accuracy floating-point value; guaranteed to contain <strong>at least 10 digits of precision</strong>.</td>
<td>12.34, 3.1e-5, 3.1e-5l</td>
<td>%f, %e, %g</td>
</tr>
<tr>
<td>long double</td>
<td>Extraextended accuracy floating-point value; guaranteed to contain <strong>at least 10 digits of precision</strong>.</td>
<td>12.341, 3.1e-5, 3.1e-51</td>
<td>%Lf, %le, %Lg</td>
</tr>
</tbody>
</table>
| char       | Single character value; on some systems, sign extension might occur when used in an expression. | 'a', '
' | %c |
| unsigned char | Same as char, except ensures that sign extension does not occur as a result of integral promotion. | -               |        |
| signed char | Same as char, except ensures that sign extension does occur as a result of integral promotion. | -               |        |
Knowing actual ranges for types

- Defined in the system include files `<limits.h>` and `<float.h>`
- `<limits.h>` contains system-dependent values that specify the sizes of various character and integer data types:
  - the maximum size of an `int` is defined by the name `INT_MAX`
  - the maximum size of an `unsigned long int` is defined by `ULONG_MAX`
- `<float.h>` gives information about floating-point data types.
  - `FLT_MAX` specifies the maximum floating-point number,
  - `FLT_DIG` specifies the number of decimal digits of precision for a float type.
Working with arithmetic expressions

• Basic arithmetic operators: +, -, *, /
• Precedence: one operator can have a higher priority, or precedence, over another operator.
  – Example: * has a higher precedence than +
  – a + b * c
  – if necessary, you can always use parentheses in an expression to force the terms to be evaluated in any desired order.
• Associativity: Expressions containing operators of the same precedence are evaluated either from left to right or from right to left, depending on the operator. This is known as the associative property of an operator
  – Example: + has a left to right associativity
Working with arithmetic expressions

```c
#include <stdio.h>
int main (void)
{
    int a = 100;
    int b = 2;
    int c = 25;
    int d = 4;
    int result;
    result = a - b; // subtraction
    printf ("a - b = %d\n", result);
    result = b * c; // multiplication
    printf ("b * c = %d\n", result);
    result = a / c; // division
    printf ("a / c = %d\n", result);
    result = a + b * c; // precedence
    printf ("a + b * c = %d\n", result);
    printf ("a * b + c * d = %d\n", a * b + c * d);
    return 0;
}
```
Integer arithmetic and the unary minus operator

// More arithmetic expressions
#include <stdio.h>
int main (void)
{
    int a = 25;
    int b = 2;
    float c = 25.0;
    float d = 2.0;
    printf ("6 + a / 5 * b = %d\n", 6 + a / 5 * b);
    printf ("a / b * b = %d\n", a / b * b);
    printf ("c / d * d = %f\n", c / d * d);
    printf ("-a = %d\n", -a);
    return 0;
}
The modulus operator

// The modulus operator
#include <stdio.h>
int main (void)
{
    int a = 25, b = 5, c = 10, d = 7;
    printf ("a %% b = %d\n", a % b);
    printf ("a %% c = %d\n", a % c);
    printf ("a %% d = %d\n", a % d);
    printf ("a / d * d + a %% d = %i\n", a / d * d + a % d);
    return 0;
}

Integer and Floating-Point Conversions

• Assign an integer value to a floating variable: does not cause any change in the value of the number; the value is simply converted by the system and stored in the floating

• Assign a floating-point value to an integer variable: the decimal portion of the number gets truncated.

• Integer arithmetic (division):
  – int divided to int => result is integer division
  – int divided to float or float divided to int => result is real division (floating-point)
Integer and Floating-Point Conversions

// Basic conversions in C
#include <stdio.h>
int main (void)
{
float f1 = 123.125, f2;
int i1, i2 = -150;
char c = 'a';
i1 = f1; // floating to integer conversion
printf ("%f assigned to an int produces %d\n", f1, i1);
f1 = i2; // integer to floating conversion
printf ("%d assigned to a float produces %f\n", i2, f1);
f1 = i2 / 100; // integer divided by integer
printf ("%d divided by 100 produces %f\n", i2, f1);
f2 = i2 / 100.0; // integer divided by a float
printf ("%d divided by 100.0 produces %f\n", i2, f2);
f2 = (float) i2 / 100; // type cast operator
printf ("(float) %d divided by 100 produces %f\n", i2, f2);
return 0;
}
The Type Cast Operator

- \( f2 = (\text{float}) i2 / 100; // \text{ type cast operator} \)
- The type cast operator has the effect of converting the value of the variable \( i2 \) to type float for purposes of evaluation of the expression.
- This operator does NOT permanently affect the value of the variable \( i2 \);
- The type cast operator has a higher precedence than all the arithmetic operators except the unary minus and unary plus.

- Examples of the use of the type cast operator:
- \((\text{int}) 29.55 + (\text{int}) 21.99 \) results in \( 29 + 21 \)
- \((\text{float}) 6 / (\text{float}) 4 \) results in \( 1.5 \)
- \((\text{float}) 6 / 4 \) results in \( 1.5 \)
The assignment operators

- The C language permits you to join the arithmetic operators with the assignment operator using the following general format: \( \text{op=} \), where \( \text{op} \) is an arithmetic operator, including +, −, ×, /, and %.
- \( \text{op} \) can also be a logical or bit operator => later in this course
- Example:
  
  \[
  \text{count} += 10;
  \]
  
  – Equivalent with:
  
  \[
  \text{count} = \text{count} + 10;
  \]
- Example: precedence of \( \text{op=} \):
  
  \[
  a /= b + c
  \]
  
  – Equivalent with:
  
  \[
  a = a / (b + c)
  \]
  
  – addition is performed first because the addition operator has higher precedence than the assignment operator