Review

Variables and data types

Operators

Epilogue
• C is a fast, small, general-purpose, platform independent programming language.
• C is used for systems programming (e.g., compilers and interpreters, operating systems, database systems, microcontrollers etc.)
• C is static (compiled), typed, structured and imperative.
• "C is quirky, flawed, and an enormous success."–Ritchie
Review: Basics

- Variable declarations: \texttt{int i; float f;}
- Initialization: \texttt{char c=’A’; int x=y=10;}
- Operators: +,−,∗,/,%
- Expressions: \texttt{int x,y,z; x=y*2+z*3;}
- Function: \texttt{int factorial (int n); /*function takes int, returns int*/}
- Review

- Variables and data types

- Operators

- Epilogue
Definitions

Datatypes:

- The **datatype** of an object in memory determines the set of values it can have and what operations that can be performed on it.
- C is a *weakly* typed language. It allows implicit conversions as well as forced (potentially dangerous) casting.

Operators:

- **Operators** specify how an object can be manipulated (*e.g.*, numeric vs. string operations).
- operators can be unary (*e.g.*, -,++), binary (*e.g.*, +,-,*,/), ternary (?:)
Expressions:

- An expression in a programming language is a combination of values, variables, operators, and functions.

Variables:

- A variable is a named link/reference to a value stored in the system’s memory or an expression that can be evaluated.

Consider: `int x=0,y=0; y=x+2;`.

- `x, y` are variables
- `y = x + 2` is an expression
- `+` is an operator.
Variable names

Naming rules:

- Variable names can contain letters, digits and _.
- Variable names should start with letters.
- Keywords (e.g., for, while etc.) cannot be used as variable names.
- Variable names are case sensitive. `int x;` and `int X` declares two different variables.

Pop quiz (correct/incorrect):

- `int money$owed;` (incorrect: cannot contain $)
- `int total_count` (correct)
- `int score2` (correct)
- `int 2ndscore` (incorrect: must start with a letter)
- `int long` (incorrect: cannot use keyword)
Data types and sizes

C has a small family of datatypes.

- Numeric (int, float, double)
- Character (char)
- User defined (struct, union)
Numeric data types

Depending on the precision and range required, you can use one of the following datatypes.

<table>
<thead>
<tr>
<th></th>
<th>signed</th>
<th>unsigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>short int x; short y;</td>
<td>unsigned short x; unsigned short int y;</td>
</tr>
<tr>
<td>default</td>
<td>int x;</td>
<td>unsigned int x;</td>
</tr>
<tr>
<td>long</td>
<td>long x;</td>
<td>unsigned long x;</td>
</tr>
<tr>
<td>float</td>
<td>float x;</td>
<td>N/A</td>
</tr>
<tr>
<td>double</td>
<td>double x;</td>
<td>N/A</td>
</tr>
<tr>
<td>char</td>
<td>char x; signed char x;</td>
<td>unsigned char x;</td>
</tr>
</tbody>
</table>

- The unsigned version has roughly double the range of its signed counterparts.
- Signed and unsigned characters differ only when used in arithmetic expressions.
- Titbit: Flickr changed from unsigned long \(2^{32} - 1\) to string two years ago.
Big endian vs. little endian

The individual sizes are machine/compiler dependent. However, the following is guaranteed:

\[
\text{sizeof(char)} < \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \quad \text{and} \\
\text{sizeof(char)} < \text{sizeof(short)} \leq \text{sizeof(float)} \leq \text{sizeof(double)}
\]

"NUXI" problem: For numeric data types that span multiple bytes, the order of arrangement of the individual bytes is important. Depending on the device architecture, we have "big endian" and "little endian" formats.
Big endian vs. little endian (cont.)

- Big endian: the **most** significant bits (MSBs) occupy the lower address. This representation is used in the powerpc processor. Networks generally use big-endian order, and thus it is called **network order**.

- Little endian: the **least** significant bits (LSBs) occupy the lower address. This representation is used on all x86 compatible processors.

Figure: (from [http://en.wikipedia.org/wiki/Little_endian](http://en.wikipedia.org/wiki/Little_endian))
## Constants

Constants are literal/fixed values assigned to variables or used directly in expressions.

<table>
<thead>
<tr>
<th>Datatype</th>
<th>example</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>i=3;</code></td>
<td>integer</td>
</tr>
<tr>
<td><code>long</code></td>
<td><code>l=3;</code></td>
<td>long integer</td>
</tr>
<tr>
<td>integer</td>
<td><code>unsigned long ul= 3UL;</code></td>
<td>unsigned long</td>
</tr>
<tr>
<td></td>
<td><code>int i=0xA;</code></td>
<td>hexadecimal</td>
</tr>
<tr>
<td></td>
<td><code>int i=012;</code></td>
<td>octal number</td>
</tr>
<tr>
<td><code>float</code></td>
<td><code>pi=3.14159</code></td>
<td>float</td>
</tr>
<tr>
<td>floating point</td>
<td><code>float pi=3.141F</code></td>
<td>float</td>
</tr>
<tr>
<td><code>double</code></td>
<td><code>pi=3.1415926535897932384L</code></td>
<td>double</td>
</tr>
<tr>
<td>Datatype</td>
<td>example</td>
<td>meaning</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>character</td>
<td>'A'</td>
<td>character specified in hex</td>
</tr>
<tr>
<td></td>
<td>'\x41'</td>
<td>specified in octal</td>
</tr>
<tr>
<td></td>
<td>'\0101'</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>&quot;hello world&quot;</td>
<td>string literal</td>
</tr>
<tr>
<td></td>
<td>&quot;hello&quot; &quot;world&quot;</td>
<td>same as &quot;hello world&quot;</td>
</tr>
<tr>
<td>enumeration</td>
<td><code>enum</code> BOOL {NO,YES}</td>
<td>NO=0, YES=1</td>
</tr>
<tr>
<td></td>
<td><code>enum</code> COLOR {R=1,G,B,Y=10}</td>
<td>G=2, B=3</td>
</tr>
</tbody>
</table>
The general format for a declaration is

\textit{type variable-name [=}value\].}

Examples:

- \textbf{char} \texttt{x}; /* uninitialized */
- \textbf{char} \texttt{x=’A’}; /* initialized to ’A’ */
- \textbf{char} \texttt{x=’A’,y=’B’}; /* multiple variables initialized */
- \textbf{char} \texttt{x=y=’Z’}; /* multiple initializations */
Pop quiz II

• `int x=017; int y=12; /* is x>y? */`
• `short int s=0xFFFF12; /* correct? */`
• `char c=−1; unsigned char uc=−1; /* correct? */`
• `puts("hel"+"lo"); puts("hel " "lo"); /* which is correct? */`
• `enum sz{S=0,L=3,XL}; /* what is the value of XL? */`
• `enum sz{S=0,L=−3,XL}; /* what is the value of XL? */`
Review

Variables and data types

Operators

Epilogue
### Arithmetic operators

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>x=3+2; /<em>constants</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y+z; /<em>variables</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x+y+2; /<em>both</em>/</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>3−2; /<em>constants</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int x=y−z; /<em>variables</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>y−2−z; /<em>both</em>/</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>int x=3*2; /<em>constants</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int x=y*z; /<em>variables</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x<em>y</em>2; /<em>both</em>/</td>
</tr>
</tbody>
</table>
**Arithmetic operators (contd.)**

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>division</td>
<td>float  x=3/2; /*produces x=1 (int /) */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>float  x=3.0/2 /*produces x=1.5 (float /) */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int  x=3.0/2; /<em>produces x=1 (int conversion)</em>/</td>
</tr>
<tr>
<td>%</td>
<td>modulus (remainder)</td>
<td>int  x=3%2; /<em>produces x=1</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int  y=7;int  x=y%4; /<em>produces 3</em>/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int  y=7;int  x=y%10; /<em>produces 7</em>/</td>
</tr>
</tbody>
</table>
Relational Operators

Relational operators compare two operands to produce a 'boolean' result. In C any non-zero value (1 by convention) is considered to be 'true' and 0 is considered to be false.

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>3&gt;2; /*evaluates to 1 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.99&gt;3 /*evaluates to 0 */</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>3&gt;=3; /*evaluates to 1 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.99&gt;=3 /*evaluates to 0 */</td>
</tr>
<tr>
<td>&lt;</td>
<td>lesser than</td>
<td>3&lt;3; /*evaluates to 0 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>’A’ &lt; ’B’ /<em>evaluates to 1</em>/</td>
</tr>
<tr>
<td>&lt;=</td>
<td>lesser than or equal to</td>
<td>3&lt;=3; /*evaluates to 1 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.99&lt;3 /*evaluates to 0 */</td>
</tr>
</tbody>
</table>
Relational Operators

Testing equality is one of the most commonly used relational operator.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
<td>3==3; /*evaluates to 1 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>’A’ ==’a’ /*evaluates to 0 */</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>3!=3; /*evaluates to 0 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.99!=3 /*evaluates to 1 */</td>
</tr>
</tbody>
</table>

Gotchas:

- Note that the "==" equality operator is different from the "+=" assignment operator.
- Note that the "==" operator on float variables is tricky because of finite precision.
# Logical operators

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
</table>
| && | AND | `((9/3)==3) && (2*3==6); /*evaluates to 1 */`  
`('A'=='a') && (3==3) /*evaluates to 0 */` |
| || | OR | `2==3 || 'A'=='A'; /*evaluates to 1 */`  
`2.99>=3 || 0 /*evaluates to 0 */` |
| ! | NOT | `!(3==3); /*evaluates to 0 */`  
`!(2.99>=3) /*evaluates to 1 */` |

Short circuit: The evaluation of an expression is discontinued if the value of a conditional expression can be determined early. Be careful of any side effects in the code.

Examples:

- `(3==3) || ((c=getchar())=='y')`. The second expression is not evaluated.
- `(0) && ((x=x+1)>0) . The second expression is not evaluated.
Increment and decrement are common arithmetic operation. C provides two short cuts for the same.

**Postfix**

- `x++` is a short cut for `x=x+1`
- `x--` is a short cut for `x=x-1`
- `y=x++` is a short cut for `y=x;x=x+1`. `x` is evaluated **before** it is incremented.
- `y=x--` is a short cut for `y=x;x=x-1`. `x` is evaluated **before** it is decremented.
Increment and decrement operators

Prefix:

- $++x$ is a short cut for $x=x+1$
- $--x$ is a short cut for $x=x-1$
- $y=++x$ is a short cut for $x=x+1;y=x;$. $x$ is evaluate after it is incremented.
- $y=--x$ is a short cut for $x=x-1;y=x;$. $x$ is evaluate after it is decremented.
# Bitwise Operators

<table>
<thead>
<tr>
<th>operator</th>
<th>meaning</th>
<th>examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>AND</td>
<td>0x77 &amp; 0x3; /*evaluates to 0x3 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x77 &amp; 0x0; /*evaluates to 0 */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x700</td>
</tr>
<tr>
<td></td>
<td>OR</td>
<td>0x070</td>
</tr>
<tr>
<td>^</td>
<td>XOR</td>
<td>0x770</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x33</td>
</tr>
<tr>
<td>«</td>
<td>left shift</td>
<td>0x01</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01 &lt;&lt;4; /*evaluates to 0x10 */</td>
</tr>
<tr>
<td>»</td>
<td>right shift</td>
<td>0x010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x010 &gt;&gt;4; /*evaluates to 0x01 */</td>
</tr>
</tbody>
</table>

Notes:

- AND is true only if both operands are true.
- OR is true if any operand is true.
- XOR is true if only one of the operand is true.
Another common expression type found while programming in C is of the type \( \text{var} = \text{var} \ (\text{op}) \ \text{expr} \)

- \( \text{x}=\text{x}+1 \)
- \( \text{x}=\text{x}\times10 \)
- \( \text{x}=\text{x}/2 \)

C provides compact assignment operators that can be used instead.

- \( \text{x}+=1 \) /*is the same as\( \text{x}=\text{x}+1 \)
- \( \text{x}-=1 \) /*is the same as\( \text{x}=\text{x}-1 \)
- \( \text{x}*=10 \) /*is the same as\( \text{x}=\text{x}\times10 \)
- \( \text{x}/=2 \) /*is the same as\( \text{x}=\text{x}/2 \)
- \( \text{x}%=2 \) /*is the same as\( \text{x}=\text{x}\%2 \)
Conditional Expression

A common pattern in C (and in most programming) languages is the following:

```c
if (cond)
    x=<expra>;
else
    x=<exprb>;
```

C provides *syntactic sugar* to express the same using the ternary operator ’?:’

```c
sign=x>0?1:-1;
if (x>0)
    sign=1
else
    sign=-1

isodd=x%2==1?1:0;
if (x%2==1)
    isodd=1
else
    isodd=0
```

Notice how the ternary operator makes the code shorter and easier to understand (syntactic sugar).
- Review
- Variables and data types
- Operators
- Epilogue
Type Conversions

When variables are promoted to higher precision, data is preserved. This is automatically done by the compiler for mixed data type expressions.

```c
int i;
float f;
f = i + 3.14159; /* i is promoted to float, f = (float) i + 3.14159 */
```

Another conversion done automatically by the compiler is 'char' → 'int'. This allows comparisons as well as manipulations of character variables.

```c
isupper = (c >= 'A' && c <= 'Z') ? 1 : 0; /* c and literal constants are converted to int */
if (!isupper)
    c = c - 'a' + 'A'; /* subtraction is possible because of integer conversion */
```

As a rule (with exceptions), the compiler promotes each term in an binary expression to the highest precision operand.
Precedence and Order of Evaluation

- `++`, `-`, `(cast)`, `sizeof` have the highest priority
- `*`, `/`, `%` have higher priority than `+`, `-`
- `==`, `!=` have higher priority than `&&`, `||`
- assignment operators have very low priority

Use `()` generously to avoid ambiguities or side effects associated with precedence of operators.

- `y=x*3+2` /*same as `y=(x*3)+2`*/
- `x!=0 && y==0` /*same as `(x!=0) && (y==0)`*/
- `d= c>=’0’ && c<=’9’` /*same as `d=(c>=’0’) && (c<=’9’)`*/