Survival Guide
Announcements

- Homework 1 posted
  - Due 11am, August 31
  - Submit via svn

- Piazza access code: __________

- Discussion sections will be held this week
Good news: Writing C code is easy!

```c
void* myfunction() {
    char *p;
    *p = 0;
    return (void*) &p;
}
```
Bad news: Writing BAD C code is easy!

```c
void* myfunction() {
    char *p;
    *p = 0;
    return (void*) &p;
}
```

What is wrong with this code?
How do I write good C programs?

- Fluency in C syntax
- Stack (static) vs. Heap (dynamic) memory allocation
- Key skill: read code for bugs
  - Do not rely solely on compiler warnings, if any, and testing
- Key skill: debugging
  - Learn to use a debugger. Don’t only rely on `printfs`!
Why C instead of Java?

- C helps you learn how to write large-scale programs
  - C is lower-level
    - C provides more opportunities to create abstractions
  - C has some flaws
    - C’s flaws motivate discussions of software engineering principles

- C helps you get “under the hood”
  - C facilitates language levels tour
    - C is closely related to assembly language
  - C facilitates services tour
    - Many existing servers/systems written in C
C vs. Java: Design Goals

- **Java design goals**
  - Support **object-oriented** programming
  - Allow same program to run on **multiple operating systems**
  - Support using **computer networks**
  - Execute code from **remote sources securely**
  - Adopt the good parts of **other languages**

- **Implications for Java**
  - Good for **application-level** programming
  - **High-level** (insulates from assembly language, hardware)
  - Portability over efficiency
  - Security over efficiency

---

Copyright ©: University of Illinois CS 241 Staff
C vs. Java: Design Goals

- **C design goals**
  - Support **structured** programming
  - Support **development of the Unix OS** and Unix tools
    - As Unix became popular, so did C

- **Implications for C**
  - Good for **systems-level** programming
  - Low-level
  - Efficiency over portability
  - Efficiency over security

- Anything you can do in Java you can do in C – it just might look ugly in C!
C vs. C++

- C++ is “C with Classes”
  - C enhanced with objects

- C has some shortcomings compared to C++
  - C++ has objects, a bigger standard library (e.g., STL), parameterized types, etc.
  - C++ is a little bit more strongly typed

- Programming Challenge
  - All syntax you use in this class is valid for C++
  - Not all C++ syntax you’ve used, however, is valid for C
A Few Differences between C and C++

- **Input/Output**
  - C does not have "iostreams"
  - C++: `cout<<"hello world"<<endl;`
  - C: `printf("hello world\n");`

- **Heap memory allocation**
  - C++: `new/delete`
    - `int *x = new int[8]; delete(x);`
  - C: `malloc()/free()`
    - `int *x = malloc(8 * sizeof(int)); free(x);`
Compiler

- gcc
  - Preprocessor
  - Compiler
  - Linker
  - See manual “man” for options: man gcc

- "Ansi-C" standards C89 versus C99
  - C99: Mix variable declarations and code (for int i=…)
  - C++ inline comments //a comment

- make – a utility to build executables
Programming in C

- C = Variables + Instructions
Programming in C

- C = Variables + Instructions
  - char
  - int
  - float
  - pointer
  - array
  - string
  - ...

Programming in C

C = Variables + Instructions

- char
- int
- float
- pointer
- array
- string

- assignment
- printf/scanf
- if
- for
- while
- switch
What we’ll show you

- You already know a lot of C from C++:
  ```c
  int my_fav_function(int x) {
    return x+1;
  }
  ```

- Key concepts for this lecture:
  - Pointers
  - Memory allocation
  - Arrays
  - Strings

Theme: how memory really works
Instant C in 3 slides: Pointers

- Data type that “points to” a value in memory, using its address
- Reference operator: &
  - address-of
- Dereference operator: *
  - contents-of
- Automatic variables
  - Temporary and stored in the stack
- Character pointers: char* p;
  - *p =0;
  - contents-of p set to 0. (Kaboom!)
- Initialization
  - Initialize a pointer to something before using it. (Doh!)
Instant C in 3 slides: Pointers

- Data type that “points to” a value in memory, using its address
- Reference operator: &
  - address-of
- Dereference operator: *
  - contents-of
- Automatic variables
  - Temporary and stored in the stack
- Character pointers: char* p;
  - *p = 0;
  - contents-of p set to 0. (Kaboom!)
- Initialization
  - Initialize a pointer to something before using it. (Doh!)

```
int x=4;
int *y = &x;
```

Question: What is the value of y?
Instant C in 3 slides: Pointers

- Data type that “points to” a value in memory, using its address
- Reference operator: &
  - address-of
- Dereference operator: *
  - contents-of
- Automatic variables
  - Temporary and stored in the stack
- Character pointers: char* p;
  - *p = 0;
  - contents-of p set to 0. (Kaboom!)
- Initialization
  - Initialize a pointer to something before using it. (Doh!)

```c
int x=4;
int *y = &x;
int a = *y;
int b = y;
```

Question: What are the values of a and b?
Instant C in 3 slides: Pointers

- Data type that “points to” a value in memory, using its address
- Reference operator: &
  - address-of
- Dereference operator: *
  - contents-of
- Automatic variables
  - Temporary and stored in the stack
- Character pointers: char* p;
  - *p = 0;
  - contents-of p set to 0. (Kaboom!)
- Initialization
  - Initialize a pointer to something before

```c
void main() {
    func();
}

void* func() {
    int x = 3;
}
```

Question: What happens to x after func() returns?
Instant C in 3 slides: Pointers

- Data type that “points to” a value in memory, using its address
- Reference operator: &
  - address-of
- Dereference operator: *
  - contents-of
- Automatic variables
  - Temporary and stored in the stack
- Character pointers: char* p;
  - *p = 0;
  - contents-of p set to 0. (Kaboom!)
- Initialization
  - Initialize a pointer to something before using it. (Doh!)

```c
void* func() {
    int* w;
    *w = 0
}

Question: What does this code output?
Instant C #2: Strings

- Unlike C++ and Java, C does not have a native string type
  - Instead, use arrays of characters terminated with a null byte

- Functions
  - `strcpy("hello", "world")` will crash
  - `strcmp(s1,s2)` returns 0 if `s1==s2`

- Arguments
  - `argv[0]` is the name of the executable
  - `argv[argc]` is a null pointer
Instant C #3: Dynamic Memory Allocation

- Allocation
  - `malloc(bytes)` to reserve memory

- Clean up
  - `free(ptr)` to free up memory

- Dynamically allocated memory is stored on the “heap”
  - Static variables are stored on the “stack”
  - You often use static variables (pointers) to refer to and manipulate heap memory
  - e.g., `char* c = malloc(sizeof(char))`
Common Causes of 'Death'

1. Uninitialized pointers
   ```
   char *dest;
   strcpy(dest,"hello");
   ```
2. C Strings need a null byte at the end
3. Buffer overflow
4. Un-initialized memory
5. Too confident: not checking return values
6. Misuse of static vs. stack variables.
Pointers
**Variables**

Type of each variable (also determines size)

- `int` $x$
- `double` $y$
- `float` $z$
- `double*` $p$
- `int` $d$

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>$x$</td>
<td>Value1</td>
</tr>
<tr>
<td>10,002</td>
<td>$y$</td>
<td>Value2</td>
</tr>
<tr>
<td>10,008</td>
<td>$z$</td>
<td>Value3</td>
</tr>
<tr>
<td>10,010</td>
<td>$p$</td>
<td>Value4</td>
</tr>
<tr>
<td>10,012</td>
<td>$d$</td>
<td>Value5</td>
</tr>
</tbody>
</table>

...
The “&” Operator: Reads “Address of”
A pointer is a variable whose value is the address of another.
The "*" Operator
Reads "Variable pointed to by"

A pointer is a variable whose value is the address of another

Copyright ©: University of Illinois CS 241 Staff
main() {
    int *p, q, x;
    x=10;
    p=&x;
    *p=x+1;
    q=x;
    printf ("Q = %d\n", q);
}
main() {
    int *p, q, x;
    x=10;
    p=&x;
    *p=x+1;
    q=x;
    printf ("Q = %d\n", q);
}

What is the Output?
main() {
    int *p, q, x;
    x = 10;
    p = &x;
    *p = x + 1;
    q = x;
    printf("Q = %d\n", q);
}

What is the Output?

main() {
    int *p, q, x;
    x=10;
p=&x;  
    *p=x+1;
    q=x;
    printf ("Q = %d\n", q);
}

Q = 11
What is the Output?

main() {
    int *p, q, x;
    x = 10;
    p = &x;
    *p = x + 1;
    q = x;
    printf ("Q = %d\n", q);
}

Copyright ©: University of Illinois CS 241 Staff
main() {
    int *p, q, x;
    x = 10;
    p = &x;
    *p = x + 1;
    q = x;
    printf("Q = %d\n", q);
}

What is the Output?
Cardinal Rule: Must Initialize Pointers before Using them

```
int *p;
*p = 10;
```
Cardinal Rule: Must Initialize Pointers before Using them

int *p;
*p = 10;

BAD!

Pointing somewhere random
Cardinal Rule: Must Initialize Pointers before Using them

```c
int *p;
*p = 10;
```

write to address: `#@*%!`
Memory allocation
Memory allocation

- Two ways to dynamically allocate memory
- Stack
  - Named variables in functions
    - Allocated for you when you call a function
    - Deallocated for you when function returns
- Heap
  - Memory on demand
    - You are responsible for all allocation and deallocation
Allocating and deallocating heap memory

- **Dynamically allocating memory**
  - Programmer explicitly requests space in memory
  - Space is allocated dynamically on the heap
  - E.g., using “malloc” in C, “new” in Java

- **Dynamically deallocating memory**
  - Must reclaim or recycle memory that is never used again
  - To avoid (eventually) running out of memory

- **“Garbage”**
  - Allocated blocks in heap that will not be used again
  - Can be reclaimed for later use by the program
Option #1: Garbage Collection

- Run-time system does garbage collection (Java)
  - Automatically determines which objects can’t be accessed
  - And then reclaims the resources used by these objects

```java
Object x = new Foo();
Object y = new Bar();
x = new Quux();
if (x.check_something()) {
  x.do_something(y);
}
System.exit(0);
```

Object Foo() is never used again!
Challenges of Garbage Collection

- Detecting the garbage is not always easy
  - `long char z = x;`
  - `x = new Quux();`
  - Run-time system cannot collect all the garbage

- Detecting the garbage introduces overhead
  - Keeping track of references to object (e.g., counters)
  - Scanning through accessible objects to identify garbage
  - Sometimes walking through a large amount of memory

- Cleaning the garbage leads to bursty delays
  - E.g., periodic scans of the objects to hunt for garbage
  - Leads to unpredictable “freezes” of the running program
  - Very problematic for real-time applications
    - ... though good run-time systems avoid long freezes
**Option #2: Manual Deallocation**

- **Programmer** deallocates the memory (C and C++)
  - Manually determines which objects can’t be accessed
  - And then explicitly returns those resources to the heap
  - E.g., using “free” in C or “delete” in C++

**Advantages**
- Lower overhead
- No unexpected “pauses”
- More efficient use of memory

**Disadvantages**
- More complex for the programmer
- Subtle memory-related bugs
- Can lead to security vulnerabilities in code
Dangling pointers

- Programmer frees a region of memory
- ... but still has a pointer to it
- Dereferencing pointer reads or writes nonsense values

```c
int main(void) {
    char *p;
    p = malloc(10);
    ...
    free(p);
    ...
    printf("%c\n", *p);
}
```

May print nonsense character
Manual deallocation can lead to bugs

- Memory leak
  - Programmer neglects to free unused region of memory
  - So, the space can never be allocated again
  - Eventually may consume all of the available memory

```c
void f(void) {
    char *s;
    s = malloc(50);
}

int main(void) {
    while (1) f();
}
```

Eventually, `malloc()` returns NULL
Manual deallocation can lead to bugs

- Double free
  - Programmer mistakenly frees a region more than once
  - Leading to corruption of the heap data structure
  - ... or premature destruction of a different object

```c
int main(void) {
    char *p, *q;
    p = malloc(10);
    ...
    free(p)
    q = malloc(10);
    free(p)
}
```

Might free space allocated by `q`!
Heap memory allocation

- **C++:**
  - `new` and `delete` allocate memory for a whole object

- **C:**
  - `malloc` and `free` deal with unstructured blocks of bytes

```c
void* malloc(size_t size);
void free(void* ptr);
```
Example

```c
int* p;
p = (int*) malloc(sizeof(int));
*p = 5;
free(p);
```

Cast to the right type

How many bytes do you want?
I’m hungry. More bytes plz.

```c
int* p = (int*) malloc(10 * sizeof(int));
```

Now I have space for 10 integers, laid out contiguously in memory. What would be a good name for that...?
Arrays

- Contiguous block of memory
  - Fits one or more elements of some type

- Two ways to allocate
  - named variable
    ```c
    int x[10];
    ```
  - dynamic
    ```c
    int* x = (int*) malloc(10*sizeof(int));
    ```

Is there a difference?

One is on the stack, one is on the heap
Arrays

```c
int p[5];
```

Name of array (is a pointer)

Shorthand:
*(p+1) is called p[1]
*(p+2) is called p[2]
etc..
Example

```c
int y[4];
y[1]=6;
y[2]=2;
```

![Array Assignment Diagram]
Array Name as Pointer

- What’s the difference between the examples?

- Example 1:
  ```c
  int z[8];
  int *q;
  q=z;
  ```

- Example 2:
  ```c
  int z[8];
  int *q;
  q=&z[0];
  ```
Array Name as Pointer

- What’s the difference between the examples?

- Example 1:
  ```c
  int z[8];
  int *q;
  q=z;
  ```

- Example 2:
  ```c
  int z[8];
  int *q;
  q=&z[0];
  ```

  `z` (the array name) is a pointer to the beginning of the array, which is `&z[0]`
Questions

- What’s the difference between
  ```c
  int* q;
  int q[5];
  ```

- What’s wrong with
  ```c
  int ptr[2];
  ptr[1] = 1;
  ptr[2] = 2;
  ```
What is the value of \( b[2] \) at the end?

```csharp
int b[3];
int* q;

b[0] = 48; b[1] = 113; b[2] = 1;

q = b;
*(q+1) = 2;
b[2] = *b;
```
Questions

- What is the value of \( b[2] \) at the end?

```c
int b[3];
int* q;
b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2; b[2]=*b;
```
What is the value of b[2] at the end?

```c
int b[3];
int* q;
b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2;
b[2]=*b;
```
Questions

What is the value of \( b[2] \) at the end?

```
int b[3];
int* q;
b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2; b[2]=*b;
```

\[
\begin{array}{c|c|c|c}
\hline
\hline
48 & 113 & 1 \\
48 & 2 & 1 \\
48 & 2 & 48 \\
48 & 2 & 50 \\
\hline
\end{array}
\]
Questions

What is the value of \( b[2] \) at the end?

```c
int b[3];
int* q;

b[0]=48; b[1]=113; b[2]=1;
q=b;
*(q+1)=2; b[2]=*b;
```

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>113</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>48</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>50</td>
</tr>
</tbody>
</table>
Strings
Strings (Null-terminated Arrays of Char)

- Strings are arrays that contain the string characters followed by a “Null” character '\0' to indicate end of string.
  - Do not forget to leave room for the null character
- Example
  - char s[5];
Conventions

- **Strings**
  - “string”
  - “c”

- **Characters**
  - ‘c’
  - ‘X’
String Operations

- `strcpy`
- `strlen`
- `strcat`
- `strcmp`
strcpy, strlen

- strcpy(ptr1, ptr2);
  - ptr1 and ptr2 are pointers to char

- value = strlen(ptr);
  - value is an integer
  - ptr is a pointer to char

int len;
char str[15];
strcpy (str, "Hello, world!");
len = strlen(str);
What’s wrong with

```c
char str[5];
strcpy (str, "Hello");
```
strncpy

- `strncpy(ptr1, ptr2, num);`
  - `ptr1` and `ptr2` are pointers to char
  - `num` is the number of characters to be copied

```c
int len;
char str1[15], str2[15];
strcpy (str1, "Hello, world!");
strncpy (str2, str1, 5);
```
strncpy

- `strncpy(ptr1, ptr2, num);`
  - `ptr1` and `ptr2` are pointers to char
  - `num` is the number of characters to be copied

int len;
char str1[15], str2[15];
strcpy (str1, "Hello, world!");
strncpy (str2, str1, 5);

Caution: `strncpy` blindly copies the characters. It does not voluntarily append the string-terminating null character.
strcat

- strcat(ptr1, ptr2);
  - ptr1 and ptr2 are pointers to char

- Concatenates the two null terminated strings yielding one string (pointed to by ptr1).

```c
char S[25] = "world!";
char D[25] = "Hello, ";
strcat(D, S);
```
strcat

- `strcat(ptr1, ptr2);`
  - `ptr1` and `ptr2` are pointers to `char`

- Concatenates the two null terminated strings yielding one string (pointed to by `ptr1`).
  - Find the end of the destination string
  - Append the source string to the end of the destination string
  - Add a NULL to new destination string
**strcat Example**

- What’s wrong with

```c
char S[25] = "world!";
strcat("Hello, ", S);
```
Example

What’s wrong with

```c
char *s = malloc(11 * sizeof(char));
    /* Allocate enough memory for an array of 11 characters, enough to store a 10-char long string. */
strcat(s, "Hello");
strcat(s, "World");
```
- **strcat**

  - `strcat(ptr1, ptr2);`
    - `ptr1` and `ptr2` are pointers to char

- Compare to Java and C++
  - `string s = s + " World!";`

- What would you get in C?
  - If you did `char* ptr0 = ptr1+ptr2;`
  - You would get the sum of two memory locations!
**strcmp**

- \( \text{diff} = \text{strcmp}(\text{ptr1}, \text{ptr2}); \)
  - \( \text{diff} \) is an integer
  - \( \text{ptr1} \) and \( \text{ptr2} \) are pointers to char

- Returns
  - zero if strings are identical
  - \(< 0\) if \( \text{ptr1} \) is less than \( \text{ptr2} \) (earlier in a dictionary)
  - \( > 0\) if \( \text{ptr1} \) is greater than \( \text{ptr2} \) (later in a dictionary)

```c
int diff;
char s1[25] = "pat";
char s2[25] = "pet";
diff = strcmp(s1, s2);
```
Can we make this work?!

int x;

printf("This class is %s.\n", &x);
Can we make this work?!

```c
int x;

printf("This class is %s.\n", );
```
Can we make this work?!

```c
int x;
(char*)&x

printf("This class is %s.\n", &x);
```
int x;
((char*)&x)[0] = 'f';

printf("This class is %s.\n", &x);
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';

printf("This class is %s.\n", &x);
Can we make this work?!

```c
int x;

((char*)&x)[0] = 'f';
((char*)&x)[1] = 'u';
((char*)&x)[2] = 'n';
((char*)&x)[3] = '\0';

printf("This class is %s.\n", &x);
```

Perfectly legal and perfectly horrible!
Can we make this work?!

int x;
char* s = &x;
strcpy(s, "fun");

printf("This class is %s\n", &x);

Perfectly legal and perfectly horrible!
Other operations
Increment & decrement

- \texttt{x++}: yield old value, add one
- \texttt{++x}: add one, yield new value

\begin{verbatim}
int x = 10;
x++;  \hspace{1cm} 11
int y = x++; \hspace{1cm} 11
int z = ++x; \hspace{1cm} 13
\end{verbatim}

- \texttt{--x} and \texttt{x--} are similar (subtract one)
Math: Increment and Decrement Operators

Example 1:
```c
int x, y, z, w;
y=10; w=2;
x=++y;
z=--w;
```

Example 2:
```c
int x, y, z, w;
y=10; w=2;
x=y++;
z=w--;
```

What are \( x \) and \( y \) at the end of each example?
Math: Increment and Decrement Operators

- **Example 1:**
  ```c
  int x, y, z, w;
y=10; w=2;
x=++y;
z=--w;
  ```
  - First increment/decrement, then assign result
  - x is 11, z is 1

- **Example 2:**
  ```c
  int x, y, z, w;
y=10; w=2;
x=y++;  
z=w--;  
  ```
  - First assign result, then increment/decrement
  - x is 10, z is 2

Copyright ©: University of Illinois CS 241 Staff
Math: Increment and Decrement Operators on Pointers

- Example 1:

```c
int a[2];
int number1, number2, *p;
a[0]=1; a[1]=10;
p=a;
number1 = *p++;
number2 = *p;
```

- What will `number1` and `number2` be at the end?
Math: Increment and Decrement Operators on Pointers

- Example

```c
int a[2];
int number1, number2, *p;
a[0]=1; a[1]=10;
p=a;
number1 = *p++;  // Hint: ++ increments pointer p not variable *p
number2 = *p;
```

- What will `number1` and `number2` be at the end?
Logic: Relational (Condition) Operators

==  equal to
!=  not equal to
>   greater than
<   less than
>=  greater than or equal to
<=  less than or equal to
Logic Example

```c
if (a == b)
    printf ("Equal.");
else
    printf ("Not Equal.");
```

Question: what will happen if I replaced the above with:

```c
if (a = b)
    printf ("Equal.");
else
    printf ("Not Equal.");
```

Perfectly LEGAL C statement! (syntactically speaking)
It copies the value in b into a. The statement will be interpreted as
TRUE if b is non-zero.
Review
int p1;
What does &p1 mean?
Review

- How much is $y$ at the end?

```c
int y, x, *p;

x = 20;
*p = 10;
y = x + *p;
```
Review

- How much is `y` at the end?

```c
int y, x, *p;

x = 20;
*p = 10;
y = x + *p;
```

BAD!!
Dereferencing an uninitialized pointer will likely segfault or overwrite something!

Segfault = unauthorized memory access
What are the differences between x and y?

```c
char* f() {
    char *x;
    static char*y;
    return y;
}
```
Review: Debugging

```c
if(strcmp("a","a"))
    printf("same!"ัญ);
```
Review: Debugging

```c
int i = 4;
int *iptr;
iptr = &i;
*iptr = 5; //now i=5
```
char *p;
p=(char*)malloc(99);
strcpy("Hello",p);
printf("%s World",p);
free(p);
char msg[5];
strcpy (msg,"Hello");
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Parentheses (function call)</td>
<td>left-to-right</td>
</tr>
<tr>
<td>[]</td>
<td>Brackets (array subscript)</td>
<td>left-to-right</td>
</tr>
<tr>
<td>.</td>
<td>Member selection via object name</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&gt;</td>
<td>Member selection via pointer</td>
<td>left-to-right</td>
</tr>
<tr>
<td>++ --</td>
<td>Postfix increment/decrement</td>
<td>right-to-left</td>
</tr>
<tr>
<td>++ --</td>
<td>Prefix increment/decrement</td>
<td>right-to-left</td>
</tr>
<tr>
<td>+ -</td>
<td>Unary plus/minus</td>
<td>left-to-right</td>
</tr>
<tr>
<td>! ~</td>
<td>Logical negation/bitwise complement</td>
<td>left-to-right</td>
</tr>
<tr>
<td>(type)</td>
<td>Cast (change type)</td>
<td>left-to-right</td>
</tr>
<tr>
<td>*</td>
<td>Derefence</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Address</td>
<td>left-to-right</td>
</tr>
<tr>
<td>sizeof</td>
<td>Determine size in bytes</td>
<td>left-to-right</td>
</tr>
<tr>
<td>*/ %</td>
<td>Multiplication/division/modulus</td>
<td>left-to-right</td>
</tr>
<tr>
<td>+ -</td>
<td>Addition/subtraction</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>Bitwise shift left, Bitwise shift right</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>Relational less than/less than or equal to/ greater than/greater than or equal to</td>
<td>left-to-right</td>
</tr>
<tr>
<td>== !=</td>
<td>Relational is equal to/is not equal to</td>
<td>left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>left-to-right</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive OR</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise inclusive OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Ternary conditional</td>
<td>right-to-left</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>+= -=</td>
<td>Addition/subtraction assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>*= /=</td>
<td>Multiplication/division assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>%= &amp;%=</td>
<td>Modulus/bitwise AND assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>^=</td>
<td>=</td>
<td>Bitwise exclusive/inclusive OR assignment</td>
</tr>
<tr>
<td>&lt;&lt;= &gt;&gt;=</td>
<td>Bitwise shift left/right assignment</td>
<td>right-to-left</td>
</tr>
<tr>
<td>,</td>
<td>Comma (separate expressions)</td>
<td>left-to-right</td>
</tr>
</tbody>
</table>