Outline

• Why C/C++?
• Variables, types, and operators
• Control Structures
• Loops
Why C/C++?

• Ubiquitous
  • A huge fraction of the world’s most crucial code is written in C
    • Java’s JVM
    • Python’s interpreter
    • Web servers
    • ...

• Mature

• Efficient
Why C/C++?

- Ubiquitous
- Mature
  - C has been around since the 70s
  - C++ has been around since the 80s
    - Many systems are built on it, so many jobs require it
- Efficient
Why C/C++?

• Ubiquitous

• Mature

• Efficient
  • Gives you lower-level access to the way your program runs
  • Enables more opportunity for optimization
    • Memory layout
    • When/how memory is allocated and deallocated
    • Allows you to leverage specialty hardware
What happens when you compile?

1. Preprocessor
   - Prepares the code for compilation
     - Code inclusion
     - Macro expansion
     - Conditional compilation
     - Comment removal

2. Compiler

3. Linker
What happens when you compile?

1. Preprocessor
2. Compiler
   - Turns human-readable *source code* into *object code*
   - May yield compiler warnings/errors
3. Linker
What happens when you compile?

1. Preprocessor
2. Compiler
3. Linker
   - Bring together all the relevant *object code* into a single executable file
   - May yield linker warnings/errors
Hello World

#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
Hello World

```c
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
```

- `#include` is a preprocessor directive, similar to import
Hello World

```c
#include <stdio.h>

// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
```

- `#include` is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
Hello World

```c
#include <stdio.h>

// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
```

- `#include` is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- `main` is a function, every program has exactly one
  - `int` is its return value
  - `main( void )` says that `main` takes no parameters
Hello World

```c
#include <stdio.h>
// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf("Hello, world!\n");
    return 0;
}
```

- `#include` is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- `main` is a function, every program has exactly one
  - `int` is its return value
  - `main( void )` says that `main` takes no parameters
- Prints a string to the console followed by a newline
Hello World

```c
#include <stdio.h>

// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
```

- `#include` is a preprocessor directive, similar to import
- Explanatory comment before function is good practice
- `main` is a function, every program has exactly one
  - `int` is its return value
  - `main( void )` says that `main` takes no parameters
- Prints a string to the console followed by a newline
- Returns the state of the program when it terminated
  - A value of zero indicates no error
Q: What if we omit the line `#include <stdio.h>`?

```

#include <stdio.h>

// Print "Hello, world!" followed by newline and exit
int main( void )
{
    printf( "Hello, world!\n" );
    return 0;
}
```

A: The compiler doesn’t know what `printf` should mean.

```
helloWorldErr.c: In function main:
helloWorldErr.c:4:3: warning: implicit declaration of function printf [-Wimplicit-function-declaration]
    printf( "hello world\n" );
     ^~~~~~~

helloWorldErr.c:4:3: warning: incompatible implicit declaration of built-in function printf
helloWorldErr.c:4:3: note: include <stdio.h> or provide a declaration of printf
```
Variables

```c
int num_students;
```

- When declared, a variable gets a *type* (*int*) and a *name* (*num_students*)
  - C/C++ are *typed languages*: every variable must have a type
- A variable also has a *value* that may change throughout the program’s life
Assignment

```c
int num_students;
num_students = 32;
```

• When declared, a variable gets a type (int) and a name (num_students)
  • C/C++ are typed languages: every variable must have a type
• A variable also has a value that may change throughout the program’s life
• = is the assignment operator, which modifies a variable’s value
Assignment

```c
int num_students = 32;
```

- When declared, a variable gets a type (`int`) and a name (`num_students`)
  - C/C++ are typed languages: every variable must have a type
- A variable also has a value that may change throughout the program’s life
- `=` is the assignment operator, which modifies a variable’s value
- It is good practice to declare and assign at the same time
  - Otherwise you have variables with undefined (random) values
    ⇒ The way the code misbehaves from run to run will not be consistent
    ⇒ It may be very hard to debug the code
Types

```c
int num_students = 32;
```

- **Integer types:**
  - `[unsigned] char`: [un]signed character (typically 1 byte)
  - `[unsigned] int`: [un]signed integer (typically 4 bytes)

- **Floating-point types:**
  - `float`: single-precision floating point number (typically 4 bytes)
  - `double`: double-precision floating point number (typically 8 bytes)

https://en.wikipedia.org/wiki/C_data_types
Operators

Take one or two values (operands) and combine to get a new value

Unary:
- negation
  -num_students

Binary
+ addition
  3 + 4
- subtraction
  num_students - 4
* multiplication
  3 * num_students
/ division
  num_students / num_students
% modulus
  num_students % 4

What happens if you add an integer and a float?
What happens if you divide an odd number by two?
Mysterious program

```c
#include <stdio.h>
int main(void)
{
    int x = 75;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f\n", y);
    return 0;
}
```

This program compiles and runs, but the naming convention makes it hard to “read”.
Less mysterious program

```c
#include <stdio.h>
int main(void)
{
    int x = 75;
    float y = 5.0 / 9.0 * (x - 32);
    printf( "%.2f\n", y );
    return 0;
}
```

This program does the same thing, but is more “readable”.

```c
#include <stdio.h>
int main( void )
{
    int fahrenheit = 75;
    float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    // print up to 2 decimal places
    printf( "%0.2f\n", celsius );
    return 0;
}
```
Precedence

• Will this code compile?
  • Yes
• Is it correct?
  • No

```
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
    int fahrenheit = 75;
    float celsius = 5.0 / 9.0 * fahrenheit - 32;
    // print up to 2 decimal places
    printf( "%0.2f\n" , celsius );
    return 0;
}
```
Precedence

- C/C++ have rules about what order operations should be performed
- Know where to look up the rules and use parentheses when in doubt

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>++ -- () [] . -&gt; (type){list}</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>2</td>
<td>++ -- + - ! ~ (type) * &amp; sizeof _Alignof</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>3</td>
<td>* / %</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>4</td>
<td>+ -</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;&lt; &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>== !=</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>^</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Precedence

• C/C++ have rules about what order operations should be performed
• Know where to look up the rules and use parentheses when in doubt

float celsius = 5.0 / 9.0 * fahrenheit - 32;

float celsius = ( ( 5.0 / 9.0 ) * fahrenheit ) - 32;
Even less mysterious program

```c
#include <stdio.h>
// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
    int base = 32;
    float factor = 5.0 / 9.0;
    int fahrenheit = 75;
    float celsius = factor * (fahrenheit - base);
    // print up to 2 decimal places
    printf( "%.2f\n" , celsius );
    return 0;
}
```
And still less mysterious program

```c
#include <stdio.h>

// Convert 75 degrees Fahrenheit to Celsius, print result
int main( void )
{
    const int base = 32;
    const float factor = 5.0 / 9.0;
    int fahrenheit = 75;
    float celsius = factor * (fahrenheit - base);
    // print up to 2 decimal places
    printf( "%0.2f\n" , celsius );
    return 0;
}
```
The `const` keyword indicates that the variable cannot be modified after it’s been declared.
The `const` keyword indicates that the variable cannot be modified later.
```c
#include <stdio.h>

// Convert 75 degrees fahrenheit to celsius, print result
int main( void )
{
    const int base = 32;
    const float factor = 5.0 / 9.0;
    const int fahrenheit = 75;
    const float celsius = factor * ( fahrenheit - base );
    printf( "%0.2f\n", celsius ); // print up to 2 decimal places

    fahrenheit = 70;
    celsius = factor * ( fahrenheit - base ) - 32;
}
```

```
>> gcc convert_fc_var3.c -std=c99 -pedantic -Wall -Wextra
helloWorldErr.c: In function main:
helloWorldErr.c:8:14: error: assignment of read-only variable fahrenheit
    fahrenheit = 70;
      ^
helloWorldErr.c:9:11: error: assignment of read-only variable celsius
    celsius = 5.0 / 9.0 * fahrenheit - 32;
```
Types (more)

• Boolean type
  • `#include <stdbool.h>`
  • type is `bool`, value is either `true` or `false`
  • Integer types can also function as bools, where 0=false, non-0=true
    • This is quite common, since bool was only introduced in C99
    • Generally, C mindset is “Booleans are just integers”
Operators (more)

Takes Boolean value(s) and returns a Boolean value

• Unary:
  \( ! \) logical "not" \( !A \) is true iff. \( A \) is false

• Binary:
  \( \&\& \) logical "and" \( (A \&\& B) \) is true iff. both \( A \) and \( B \) are true
  \( || \) logical "or" \( (A || B) \) is true iff. either or both \( A \) and \( B \) are true
Operators (more)

Takes integer/floating-point value and returns a Boolean value

• Equality operators:
  \[ A == B \] is true iff A equals B
  \[ A != B \] is true iff A does not equal B

• Relational operators
  \[ A > B \] is true iff A is greater than B
  \[ A < B \] is true iff A is less than B
  \[ A >= B \] is true iff A is greater than or equal to B
  \[ A <= B \] is true iff A is less than or equal to B

*You should avoid using these to compare floating point values!
Operators (more)

Short-circuiting:

- When C evaluates the composition of logical expression. ...

\[
\begin{align*}
\text{if( (statement \_1) || (statement\_2) )} \\
\text{if( (statement \_1) && (statement\_2) )}
\end{align*}
\]

... it \textit{short circuits} as soon as answer is definitely true or definitely false.

- \text{if( a == 7 || b == 7 ):}
  
  When (a==7) is true, the entire expression is true so we don’t need to test if (b == 7) is true.

- \text{if( a == 7 && b == 7 ):}
  
  When (a==7) is false, the entire expression is false so we don’t need to test if (b == 7) is true.
Types (more)

• Character type
  • a char variable holds a single character:
    • char digit = '4';
    • char bang = '!';
  • These must be single quotes; double quotes are for strings, not chars
  • Behind the scenes, char is just like int:
    \[
    \text{char digit} = '4' - 1;
    \]
    digit now contains the character '3'
• The ASCII standard governs the mapping between characters and integers.
Q: What does this print?

```c
#include <stdio.h>
int main( void )
{
    char char_0 = '0';
    int int_0 = char_0 - '0';
    printf( "Character printed as character: %c\n" , char_0 );
    printf( "Character printed as integer: %d\n" , char_0 );
    printf( "Integer printed as integer: %d\n" , int_0 );
}
```

```
Character printed as character: 0
Character printed as integer: 48
Integer printed as integer: 0
```

>
Control structures

• The `if` statement evaluates a Boolean predicate and executes the code in braces if the predicate is true.

```c
#include <stdio.h>
int main( void )
{
    int n = 12;
    if( n % 2 == 0 )
    {
        printf( "E\n" );
    }
    return 0;
}

>> ./a.out
E
>>
Control structures

• The if statement evaluates a Boolean predicate and executes the code in braces if the predicate is true.

• If no braces are provided, the if only affects the next command (i.e. up to the next “;”).

```c
#include <stdio.h>
int main( void )
{
    int n = 12;
    if( n % 2 == 0 )
        printf( “E\n” );
    return 0;
}
```

Note: White-space / indentation has no effect on what the if applies to.
Control structures

• The `if` statement evaluates a Boolean predicate and executes the code in braces if the predicate is true.

• It no braces are provided, the `if` only affects the next command (i.e. up to the next `;`).

• Can even put on one line (if it’s readable).

```c
#include <stdio.h>
int main( void )
{
    int n = 12;
    if( n % 2 == 0 ) printf( "E\n" );
    return 0;
}
```

Note: White-space / indentation has no effect on what the `if` applies to.
Control structures

• The if / else statement evaluates a Boolean predicate and follows the if branch if the predicate is true and the else branch otherwise.

```c
#include <stdio.h>
int main( void )
{
    int n = 13;
    if( n % 2 == 0 )
    {
        printf( "E\n" );
    }
    else
    {
        printf( "O\n" );
    }
    return 0;
}
```
Control structures

• The `if / else` statement evaluates a Boolean predicate and follows the if branch if the predicate is true and the else branch otherwise.

• It no braces are provided, the `if / else` only effect the next command (i.e. up to the next “;”).

```c
#include <stdio.h>
int main( void )
{
    int n = 13;
    if( n % 2 == 0 ) printf( "E\n" );
    else printf( "O\n" );
    return 0;
}
```
Control structures

- The *if* / *else* statement evaluates a Boolean predicate and follows the *if* branch if the predicate is true and the *else* branch otherwise.

- It no braces are provided, the *if* / *else* only effect the next command (i.e. up to the next `;`).

- The *else* is always associated to the last (unmatched) *if*.

```c
#include <stdio.h>
int main( void )
{
    int n = 13;
    if( n % 2 == 0 )
        if( n==11 ) printf( "11\n" );
        else printf( "E\n" );
    return 0;
}
```

```
>> ./a.out
>>
```
Control structures

• The if / else statement evaluates a Boolean predicate and follows the if branch if the predicate is true and the else branch otherwise.

• It no braces are provided, the if / else only effect the next command (i.e. up to the next “;”).

• The else is always associated to the last (unmatched) if.

```c
#include <stdio.h>
int main( void )
{
    int n = 13;
    if( n % 2 == 0 )
    {
        if( n==11 ) printf( "11\n" );
    }
    else printf( "O\n" );
    return 0;
}
```

```
> .a.out
0
> 
```
Control structures

- The `if / else if / else` statement evaluates a sequence of Boolean predicates, and executes the code when the predicate is true.

```c
#include <stdio.h>
int main( void )
{
    int x = 79;
    if ( x >= 90 ) printf( "A\n" );
    else if( x >= 80 ) printf( "B\n" );
    else if(x >= 70) printf( "C\n" );
    else if(x >= 60) printf( "D\n" );
    else printf( "F\n" );
    return 0;
}
```
Control structures

• The `switch` statement tests if a value matches one of a set of prescribed cases and executes all the code after if it does.
  • `switch`: Specifies the value to be tested
  • `case`: specifies the case to execute
  • `break`: do not continue to the next case
  • `default`: if nothing else matched...

```c
#include <stdio.h>
int main( void )
{
    char grade = 'C';
    int points = 0;
    switch( grade )
    {
        case 'A':
            points = 4;
            break;
        case 'B':
            points = 3;
            break;
        case 'C':
            points = 2;
            break;
        case 'D':
            points = 1;
            break;
        default:
            points = 0;
            break;
    }
    printf( "Grade %c -> %d GPA points\n", grade, points);
}
```

```
>> ./a.out
Grade C -> 2 points>
```
Compound assignment

Combine binary operators with assignment operators:

\[
\begin{align*}
A &+= B; \quad \Rightarrow \quad A = A + B; \\
A &-= B; \quad \Rightarrow \quad A = A - B; \\
A &= B; \quad \Rightarrow \quad A = A * B; \\
A &/= B; \quad \Rightarrow \quad A = A / B; \\
A &%= B; \quad \Rightarrow \quad A = A % B;
\end{align*}
\]

The right hand side can be either a variable or a constant.
Increment and decrement

Increase / decrease the value by one:

\[ A++; \Rightarrow A = A+1; \]
\[ A--; \Rightarrow A = A-1; \]
\[ ++A; \Rightarrow A = A+1; \]
\[ --A; \Rightarrow A = A-1; \]

The difference between \( A++ \) and \( ++A \) (or \( A-- \) and \( --A \)) is precedence.
Increment and decrement

Increase / decrease the value by one:

\[ B = A++; \quad \Rightarrow \quad \{ \ B = A; \quad A = A+1; \ \} \]
\[ B = A--; \quad \Rightarrow \quad \{ \ B = A; \quad A = A-1; \ \} \]
\[ B = ++A; \quad \Rightarrow \quad \{ \quad A = A+1; \quad B = A; \ \} \]
\[ B = --A; \quad \Rightarrow \quad \{ \quad A = A-1; \quad B = A; \ \} \]
Increment and decrement

Increase / decrease the value by one:

```c
#include <stdio.h>
int main( void )
{
    int i = 0;
    if( ++i ) printf( "++i was non-zero\n" );
    printf( "i=%d\n" , i );
    i = 0;
    if( i++ ) printf( "i++ was non-zero\n" );
    printf( "i=%d\n" , i );
}
```

```
>> ./a.out
++i was non-zero
i=1
>>
```
Loops

The for loop

#include <stdio.h>
int main( void )
{
    for( int i=0 ; i<10 ; i++ )
    {
        printf( "%d\n", i );
    }
}
Loops

The `for` loop:

- Initializes a loop variable

```c
#include <stdio.h>
int main( void )
{
    for( int i=0 ; i<10 ; i++ )
    {
        printf( "%d\n" , i );
    }
}
```
Loops

The *for* loop:

- Initializes a loop variable
- Iterates while the looping condition is met

```c
#include <stdio.h>
int main( void )
{
    for( int i=0 ; i<10 ; i++ )
    {
        printf( "%d\n" , i );
    }
}
```
Loops

The **for** loop:

- Initializes a loop variable
- Iterates while the looping condition is met
- Adjusts the loop value **after** each iteration

```c
#include <stdio.h>
int main( void )
{
    for( int i=0 ; i<10 ; i++ )
    {
        printf( "%d\n" , i );
    }
}
```
Loops

The **for** loop:

- Initializes a loop variable
- Iterates while the looping condition is met
- Adjusts the loop value **after** each iteration
- Performs the calculation in braces at each iteration

```c
#include <stdio.h>
int main( void )
{
    for( int i=0 ; i<10 ; i++ )
    {
        printf("%d
", i);
    }
}
```
Loops

The **for** loop:

- Initializes a loop value
- Iterates while the looping condition is met
- Adjusts the loop value **after** each iteration
- Performs the calculation in braces at each iteration
  - If no braces are provided, it performs the next command

```c
#include <stdio.h>
int main( void )
{
    for( int i=0 ; i<10 ; i++ )
        printf( "%d\n" , i );
}
```
Loops

The **while** loop:
- Iterates until the **while** condition fails.
- Performs the calculation in braces at each iteration.
Loops

The **while** loop:
- Iterates until the **while** condition fails.
- Performs the calculation in braces at each iteration

How about this?

```c
#include <stdio.h>
int main( void )
{
    int i = 0;
    while( (i%7) != 0 )
    {
        printf( "%d\n" , i );
        i++;
    }
}
```

```bash
> ./a.out
```
Loops

The **while** loop:
- Iterates until the **while** condition fails.
- Performs the calculation in braces at each iteration
  - If no braces are provided, it performs the next command

```c
#include <stdio.h>
int main( void )
{
    int i = 1;
    while( (i%7) != 0 )
    {
        printf( "%d
", i++ );
    }
```


Loops

The `while` loop:

- Iterates until the `while` condition fails.

Note that a `for` loop can always be implemented as a `while` loop.

```c
#include <stdio.h>
int main( void )
{
    int i = 1;
    while( (i%7) != 0 )
        printf( "%d\n" , i++ );
}
```

```c
#include <stdio.h>
int main( void )
{
    for( int i=1 ; (i%7) != 0 ; i++ )
        printf( "%d\n" , i);
}
```
Loops

The **do / while** loop:

- Like a while loop, but is always guaranteed to perform at least one iteration (i.e. tests the condition after the loop, not before)
- Performs the calculation in braces at each iteration

```c
#include <stdio.h>
int main( void )
{
    int i = 0;
    do 
    {
        printf( "%d\n", i );
        i++;
    }
    while( (i%7) != 0 );
}
```
Loops

The **do / while** loop:

- Like a while loop, but is always guaranteed to perform at least one iteration (i.e. tests the condition after the loop, not before)
- Performs the calculation in braces at each iteration
  - If no braces are provided, it performs the next command

```c
#include <stdio.h>
int main( void )
{
    int i = 0;
    do printf( "%d\n", i++ );
    while( (i%7) != 0 );
}
```
Loops (summary)

• **while**(boolean expression){ statements }
  • Iterates 0 times, as long as boolean expression is true
  • Execute statements at each iteration

• **do** { statements } **while**(boolean expression)
  • Iterates 1 times, as long as boolean expression is true
  • Execute statements at each iteration

• **for**(init ; boolean expression ; update){ statements }
  • init happens first; usually declares & assigns “index variable”
  • Iterates 0 times, as long as boolean expression is true
  • Execute statements at each iteration
  • update is run after statements; often it increments the loop variable (i++)
Loops (summary)

- **while**(boolean expression){
  - Iterates 0 times, as long as boolean expression is true.
  - Execute statements at each iteration.
  - Iterate 0 times, as long as boolean expression is true.
  - Execute statements at each iteration.

- **do**{ statements }**while**(boolean expression){
  - Iterates 1 times, as long as boolean expression is true.
  - Execute statements at each iteration.
  - Iterate 1 times, as long as boolean expression is true.
  - Execute statements at each iteration.

- **for**(init; boolean expression; update){
  - init happens first; usually declares & assigns "index variable".
  - Iterate 0 times, as long as boolean expression is true.
  - Execute statements at each iteration.
  - Iterate 0 times, as long as boolean expression is true.
  - Execute statements at each iteration.

  - update is run after statements; often it increments the loop variable (i++).

If statements has the command break, the code terminates the loop regardless of whether or not boolean expression is true.

```c
#include <stdio.h>
int main( void )
{
    int i = 0;
    do
    {
        printf( "%d\n" , i++ );
        if( (i%7) != 0 )
            break;
    } while( true );
}```