"There is no reason anyone would want a computer in their home."
Ken Olson, president, chairman and founder of Digital Equipment Corp., 1977

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg."
Bjarne Stroustrup

"Prediction is very hard. Especially about the future."
Yogi Berra

**History of C++**

- Bjarne Stroustrup, ATT Bell Labs, 1980
- A "mid-level" language, C plus OOP plus lots of new syntax
  - statically typed; compiled into native executables (like C)
  - designed to be forward-compatible (old C programs work as C++)
  - supports many programming styles; but difficult to master
- Current usage
  - most operating system software (Windows, Linux) is in C/C++
  - most applications, games, device drivers, embedded software

**Design goals of C++**

- Provide object-oriented features in C-based language, without compromising efficiency
  - backwards compatibility with C
  - better static type checking
  - data abstraction
  - objects and classes
  - prefer efficiency of compiled code where possible
- Important principle
  - if you do not use a feature, your compiled code should be as efficient as if the language did not include the feature

**Difficulties in using C++**

- Casts
  - sometimes no-op, sometimes not (e.g., multiple inheritance)
- Lack of garbage collection
- Objects can be allocated on stack or heap
  - can be more efficient, but assignment works badly; dangling pointers
- Too many ways to do the same thing
- Multiple inheritance
  - efforts at efficiency lead to complicated behavior
- Lack of standardization between C++ compilers (improving)

**Hello, world!**

```cpp
// hello.cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}
```
Compiling a C++ program

- `g++ -g -Wall -o executable source.cpp`
- `g++ -g -Wall -c source.cpp` (make a .o file)
- program files named with .cpp, not .c
- sometimes also named .cc
- g++ compiler, not gcc
- same command-line arguments and concepts

Basic language syntax

- same as C:
  - all control statements (if/else, for, while, do), expressions, precedence, variables, braces, functions, parameters, returns, types (can use bool without including stdbool), comments (;/ officially allowed), preprocessor
- new/different:
  - classes and objects
  - inheritance (single and multiple!)
  - data structures (STL)
  - operator overloading
  - templates (generics)
  - exceptions
  - namespaces
  - reference parameters

I/O streams

- `#include <iostream>`
  - I/O library; replaces some features of stdio.h
  - in C++ you can include system libraries without writing the .h
- stream: a source/target for reading/writing bytes in sequence.
- other iostreams: fstream, stringstream, etc.

<table>
<thead>
<tr>
<th>variable</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cin</td>
<td>standard input stream</td>
</tr>
<tr>
<td>cout</td>
<td>standard output stream</td>
</tr>
<tr>
<td>cerr</td>
<td>standard error stream</td>
</tr>
</tbody>
</table>

Using I/O streams

- sends data "in the direction of the arrow"
- `endl` sends '\n' and flushes stream:
  ```
  cout << "Student #" << i << endl;
  ```
- input with `cin`:
  ```
  int age;
  cout << "Type your age: ";
  cin >> age;
  ```

Formatting: iomanip

- `#include <iomanip>`
- formatted output (a la printf)
  - `setw(n)` - set width of next field to be printed
  - `setprecision(p)` - set precision (decimal places) of next field
  - `setfill`, `setbase`, ...
  - (you can still use printf if you want; often easier)
- `cout << "You have " << setw(4) << x << " credits." << endl;`

Namespaces

- `using namespace name;`
- namespace: An abstract container for holding a logical grouping of unique identifiers (names) in a program:
  - allows grouping of names, functions, classes
  - doesn't exist in C (all functions are global)
  - a bit like packages in Java; can be nested
- `cin`, `cout`, `endl`, strings, etc. are all found in namespace std
  - can 'use' that namespace to access those identifiers
  - or the :: scope resolution operator (also seen in OOP code):
    ```
    std::cout << "Hello, world!" << std::endl;
    ```
Namespaces, cont’d.

```cpp
namespace name {
    <your code>
}
namespace integermath {
    int squared(int x) {
        return x * x;
    }
}
...

int main(void) {
    cout << integermath::squared(7);   // 49
}
```

Functions and parameters

- functions can be overloaded in C++
  - two functions with the same name, different parameters
  - compares how to polymorphism?
- parameters can have default values (must be the last
  param(s))

```cpp
void printLetter(char letter, int times = 1) {
    for (int i = 1; i <= times; i++) {
        cout << letter;
    }
    cout << endl;
}
...
printLetter('*');      // prints 1 star
printLetter('!', 10);  // prints 10 !s
```

References

- `type& name = variable;`
- reference: A variable that is a direct alias for another variable.
  - any changes made to the reference will affect the original
  - like pointers, but more constrained and simpler syntax
  - an effort to "fix" many problems with C's implementation of
    pointers
- Example:
  ```cpp
  int x = 3;
  int& r = x;     // now use r just like any int
  r++;
  // r == 4, x == 4
  ```
- value on right side of = must be a variable, not an
  expression/cast

References vs. pointers

- don't use * and & to reference / dereference (just & at
  assignment)
- cannot refer directly to a reference; just refers to what it refers to
- a reference must be initialized at declaration
- `int& r; // error`
- a reference cannot be reassigned to refer to something else
  ```cpp
  int x = 3, y = 5;
  int& r = x;
  r = y;         // sets x == 5, r == 5
  ```
- a reference cannot be null, and can only be "invalid" if it refers to
  an object/memory that has gone out of scope or was freed

Reference parameters

```cpp
returntype name(type& name, ...) {
    ...
}
```
- client passes parameter using normal syntax
- if function changes parameter's value, client variable will change
- you almost never want to return a reference
  - except in certain cases in OOP

const and references

- `const`: Constant, cannot be changed.
  - used much, much more in C++ than in C
  - can have many meanings (const pointer to a const int?)

```cpp
void printSquare(const int& i){
    i = i * i;            // error
    cout << i << endl;
}
int main() {
    int i = 5;
    printSquare(i);
}
```
Strings

• `#include <string>`
  • C++ actually has a class for strings
    – much like Java strings, but mutable (can be changed)
    – not the same as a "literal" or a char*, but can be implicitly converted

```
string str1 = "Hello"; // impl. conv.
```

• Concatenating and operators

```
string str3 = str1 + str2;
if (str1 == str2) { // compares characters
  if (str1 < str3) { // compares by ABC order
    char c = str3[0]; // first character
  }
}
```

String methods

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>append(str)</code></td>
<td>append another string to end of this one</td>
</tr>
<tr>
<td><code>c_str()</code></td>
<td>return a const char* for a C++ string</td>
</tr>
<tr>
<td><code>clear()</code></td>
<td>removes all characters</td>
</tr>
<tr>
<td><code>compare()</code></td>
<td>like Java's compareTo</td>
</tr>
<tr>
<td><code>find(str [, index])</code></td>
<td>search for index of a substring</td>
</tr>
<tr>
<td><code>find(str [, index])</code></td>
<td>search for index of a substring</td>
</tr>
<tr>
<td><code>insert(index, str)</code></td>
<td>add characters to this string at given index</td>
</tr>
<tr>
<td><code>length()</code></td>
<td>number of characters in string</td>
</tr>
<tr>
<td><code>push_back(ch)</code></td>
<td>adds a character to end of this string</td>
</tr>
<tr>
<td><code>replace(index, len, str)</code></td>
<td>replace given range with new text</td>
</tr>
<tr>
<td><code>substr(start [, len])</code></td>
<td>substring from given start index</td>
</tr>
</tbody>
</table>

```
string s = "Goodbye world!";
s.insert(7, " cruel"); // "Goodbye cruel world!"
```

String concatenation

• a string can do + concatenation with a string or char*, but not with an int or other type:

```
string s1 = "hello"
string s2 = "there"
s1 = s1 + " " + s2; // ok
s1 = s1 + 42; // error
```

• to build a string out of many values, use a stringstream
  – works like an ostream (cout) but outputs data into a string
  – call .str() on stringstream once done to extract it as a string

```
#include <sstream>
stringstream stream;
stream << s1 << " " << s2 << 42;
s1 = stream.str(); // ok
```

Libraries

#include `<cmath>`

<table>
<thead>
<tr>
<th>library</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cassert</td>
<td>assertion functions for testing (assert)</td>
</tr>
<tr>
<td>cctype</td>
<td>char type functions (isalpha, tolower)</td>
</tr>
<tr>
<td>cmath</td>
<td>math functions (sqrt, abs, log, cos)</td>
</tr>
<tr>
<td>cstdio</td>
<td>standard I/O library (fopen, rename, printf)</td>
</tr>
<tr>
<td>cstdlib</td>
<td>standard functions (rand, exit, malloc)</td>
</tr>
<tr>
<td>cstring</td>
<td>char* functions (strcpy, strlen) (not the same as &lt;string&gt;, the string class)</td>
</tr>
<tr>
<td>ctime</td>
<td>time functions (clock, time)</td>
</tr>
</tbody>
</table>

Arrays

• stack-allocated (same as C):
  
```
type name[size];
```

• heap-allocated:
  
```
type* name = new type[size];
  
  C++ uses new and delete keywords to allocate/free memory
  
  - arrays are still very dumb (don't know size, etc.)
  
  int* nums = new int[10];
  for (int i = 0; i < 10; i++) {
    nums[i] = i * i;
  }

  ... delete[] nums;
```

malloc vs. new

<table>
<thead>
<tr>
<th>malloc</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>place in language</td>
<td>a function</td>
</tr>
<tr>
<td>how often used in C</td>
<td>often</td>
</tr>
<tr>
<td>how often used in C++</td>
<td>rarely</td>
</tr>
<tr>
<td>allocates memory for</td>
<td>anything</td>
</tr>
<tr>
<td>returns what</td>
<td>void* (requires cast)</td>
</tr>
<tr>
<td>when out of memory</td>
<td>returns NULL</td>
</tr>
<tr>
<td>deallocating</td>
<td>free</td>
</tr>
</tbody>
</table>
Exceptions

- exception: An error represented as an object or variable.
  - C handles errors by returning error codes
  - C++ can also represent errors as exceptions that are thrown / caught
- throwing an exception with throw:
  
  ```cpp
  double sqrt(double n) {
    if (n < 0) {
      throw n; // kaboom
    }
  }
  ```

- can throw anything (a string, int, etc.)
- can make an exception class if you want to throw lots of info:
  
  ```cpp
  #include <exception>
  ```

More about exceptions

- catching an exception with try/catch:
  
  ```cpp
  try {
    double root = sqrt(x);
  } catch (double d) {
    cout << d << " can't be squirted!" << endl;
  }
  ```

- throw keyword indicates what exception(s) a method may throw
  - void f() throw(); // none
  - void f() throw(int); // may throw ints

- predefined exceptions (from std::exception):
  - bad_alloc, bad_cast, ios_base::failure,

Questions?