History of C++

- made by Bjarne Stroustrup, AT&T / Bell Labs in 1980
  - original name: "C with Classes"
  - Stroustrup's book: *The C++ Programming Language*

- 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
Things that suck about C++

• Casts
  ▪ sometimes no-op, sometimes not (e.g., multiple inheritance)

• Lack of garbage collection
  ▪ memory management is error prone

• Objects can be allocated on stack or heap
  ▪ can be more efficient, but assignment works badly; dangling ptrs

• (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!

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

int main() {
    cout << "Hello, world!" << endl;
    return 0;
}

#include <stdio.h>   /* hello.c */
int main(void) {
    printf("Hello, world!\n");
    return 0;
}
Compiling a C++ program

```bash
g++ -g -Wall -o executable source.cpp

['.cpp', not '.c
```
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.

<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>

- other iostreams: `fstream`, `stringstream`, etc.
Using I/O streams

<table>
<thead>
<tr>
<th>command</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cout &lt;&lt; \textit{expression}</td>
<td>output extraction operator; write the value of \textit{expression} to standard out</td>
</tr>
<tr>
<td>cin &gt;&gt; \textit{variable}</td>
<td>input extraction operator; read from standard input and store it in \textit{variable}</td>
</tr>
</tbody>
</table>

- sends data "in the direction of the arrow"

- \texttt{endl} sends '\n' and flushes stream:
  - cout << "Student #" << i << endl;

- input with cin: (can also use getline to read entire line)
  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)

```cpp
cout << "You have " << setw(4) << x << " credits." << endl;
```
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.

• placing your own code inside a namespace:

```cpp
namespace name {
  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

• 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
```
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:

```c
int x = 3;
int& r = x; // now I 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

- references differ from 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
    ```
    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, seen later

- Exercise: Write a swap method for two ints.
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);
}
```
#include <string>

- C++ actually has a class for strings (yay!)
  - much like Java strings, but *mutable* (can be changed)
  - not the same as a "literal" or a char*, but can be implicitly converted

```cpp
string str1 = "Hello";  // implicit conversion
```

- 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(str)</code></td>
<td>like Java's <code>compareTo</code></td>
</tr>
<tr>
<td><code>find(str [, index])</code></td>
<td>search for index of a substring</td>
</tr>
<tr>
<td><code>rfind(str [, index])</code></td>
<td></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:

```cpp
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

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

```c
#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)</td>
</tr>
<tr>
<td></td>
<td>(not the same as <code>&lt;string&gt;</code>, the string class)</td>
</tr>
<tr>
<td>ctime</td>
<td>time functions (clock, time)</td>
</tr>
</tbody>
</table>
Arrays

- stack-allocated (same as C):
  
  \[ \text{type name}[\text{size}]; \]

- heap-allocated:
  
  \[ \text{type}^* \ \text{name} = \text{new} \ \text{type}[\text{size}]; \]

  - C++ uses `new` and `delete` keywords to allocate/free memory
  - arrays are still very dumb (don't know size, etc.)

  ```
  int^* \ \text{nums} = \text{new} \ \text{int}[10];
  \text{for (int i = 0; i < 10; i++)}{
    \text{nums}[i] = i * i;
  }
  ...
  \text{delete[]} \ \text{nums};
  ```
# malloc vs. new

<table>
<thead>
<tr>
<th></th>
<th>malloc</th>
<th>new</th>
</tr>
</thead>
<tbody>
<tr>
<td>place in language</td>
<td>a function</td>
<td>an operator (and a keyword)</td>
</tr>
<tr>
<td>how often used in C</td>
<td>often</td>
<td>never (not in language)</td>
</tr>
<tr>
<td>how often used in C++</td>
<td>rarely</td>
<td>frequently</td>
</tr>
<tr>
<td>allocates memory for</td>
<td>anything</td>
<td>arrays, structs, and objects</td>
</tr>
<tr>
<td>returns what</td>
<td>void* (requires cast)</td>
<td>appropriate type (no cast)</td>
</tr>
<tr>
<td>when out of memory</td>
<td>returns NULL</td>
<td>throws an exception</td>
</tr>
<tr>
<td>deallocating</td>
<td>free</td>
<td>delete (or delete[])</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

```cpp
void f() throw();    // none
void f() throw(int); // may throw ints
```

• predefined exceptions: bad_alloc, bad_cast, ios_base::failure, ...
  ▪ all derive from std::exception