

Intermediate Programming

Day 22

Outline

- C++
- I/O
- Namespaces
- *strings*
- Review questions

C++

- C++ is designed to enrich C by providing additional functionality:
 - Classes and inheritance
 - Overloading
 - Templates
 - Standard template library (STL)
 - Cleaner I/O
- It is not quite a superset of C. Many C programs won't work "out of the box" as C++.

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++

- Stages of compilation are the same as for C:
preprocess → compile → link → execute
- Use g++ instead of gcc
- Use -std=c++11 instead of -std=c99
- Files end with .cpp instead of .c

```
main.cpp  
#include <iostream>  
  
int main( void )  
{  
    std::cout << "Hello World!" << std::endl;  
}
```

```
>> g++ hello_world.cpp -std=c++11 -pedantic -Wall -Wextra  
>> ./a.out  
Hello World!  
>>
```

C++

- Stages of compilation are the same as for C:

preprocess → compile → link → execute

- Can still compile and link separately:

- -c: compile object files only
- -o: specify the output file-name

- Can debug:

- -g: include debugging symbols

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
}
```

```
>> g++ -c -g hello_world.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o hello_world hello_world.o
>> ./hello_world
Hello World!
>>
```

C++

- Our favorite tools work just as well with C++ as with C:
 - make
 - gdb
 - valgrind

```
main.cpp  
#include <iostream>  
  
int main( void )  
{  
    std::cout << "Hello World!" << std::endl;  
    return 0;  
}
```

C++

`#include <iostream>`

- As in C, headers provided by C++ are included with angle brackets
- For C++ headers, do not use a trailing .h:
`<iostream.h> → <iostream>`
- User-defined headers still go in quotes
and have .h extensions:
`#include "linkedList.h"`

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

C++

- Can use familiar C headers: `assert.h`, `math.h`, `ctype.h`, `stdlib.h`, ...

main.cpp

```
#include <iostream>
#include <cassert>
int main( int argc , char *argv[] )
{
    assert( argc>1 );
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

```
>> g++ -c main.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o main main.o
>> ./main misha
Hello misha!
>>
```


C++

- Can use familiar C headers: `assert.h`, `math.h`, `ctype.h`, `stdlib.h`, ...
 - When `#include`'ing, drop `.h` and add `c` at the beginning

main.cpp

```
#include <iostream>
#include <cassert>
int main( int argc , char *argv[] )
{
    assert( argc>1 );
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

```
>> g++ -c main.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o main main.o
>> ./main misha
Hello misha!
>>
```

C++

- Can use familiar C headers: `assert.h`, `math.h`, `ctype.h`, `stdlib.h`, ...
 - When `#include`'ing, drop `.h` and add `c` at the beginning
 - `argc` and `argv` work as before

```
main.cpp
#include <iostream>
#include <cassert>
int main( int argc , char *argv[] )
{
    assert( argc>1 );
    std::cout << "Hello " << argv[1] << "!" << std::endl;
    return 0;
}
```

```
>> g++ -c main.cpp -std=c++11 -pedantic -Wall -Wextra
>> g++ -o main main.o
>> ./main misha
Hello misha!
>>
```

Outline

- C++
- I/O
- Namespaces
- *strings*
- Review questions

C++ Input/Output

```
#include <iostream>
```

- This is the main C++ library for (streaming) input and output

```
main.cpp  
#include <iostream>  
  
int main( void )  
{  
    std::cout << "Hello World!" << std::endl;  
    return 0;  
}
```

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

- `std::cout` is the standard output stream
 - Like `stdout` in C

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

- `std::endl` is the end-of-line character
 - In C, we called it '`\n`'
 - In C++ it's better to use `std::endl` (this flushes the buffer)

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

- << is the output-to-stream operator
 - It takes a reference* to an output stream and a string
 - It returns a reference* to the output stream
 - It is processed left to right
(std::cout << "Hello World!") << std::endl;
⇒ We can chain outputs

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
>> ./a.out
Hello World!
>>
```

*More on this later.

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

main.cpp

```
#include <iostream>
int main(void)
{
    int inventory = 44;
    double price = 0.70;
    const char *item = "chainsaw";
    std::cout << "We have " << inventory << " " << item << "s left,"
        << " costing $" << price << " per unit" << std::endl;
    return 0;
}
```

```
>> ./a.out
We have 44 chainsaws left, costing $0.7 per unit
>>
```

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

main.cpp

```
#include <iostream>
int main(void)
{
    int inventory = 44;
    double price = 0.70;
    const char *item = "chainsaw";
    std::cout << "We have " << inventory << " " << item << "s left,"
              << " costing $" << price << " per unit" << std::endl;
    return 0;
}
```

No formatting required (%d, %f, %s, etc.)

- [Overloading] The compiler uses the type of the argument to determine how to print the argument

C++ Input/Output

```
std::cout << "Hello World!" << std::endl;
```

- An example of C++ I/O but also an example of (operator) *overloading*^{*}
 - << usually does bitwise left-shift
 - If the left operand is a C++ stream (`std::cout`), then << is the output operator
 - If the argument is a **string** then print the value as a string
 - If the argument is an **int** then print the value as an **int**
 - If the argument is a **float** then print the value as a **float**
 - etc.

^{*}More on this later.

C++ Input/Output

Q: How much of C can we use in C++?

A: Nearly everything

main.cpp

```
#include <stdio>
int main( void )
{
    int inventory = 44;
    double price = 0.70;
    const char *item = "chainsaw";
    printf( "We have %d %ss left costing $%f per unit\n", inventory , item , price );
    return 0;
}
```

```
>> ./a.out
We have 44 chainsaws left costing $0.700000 per unit
>>
```

C++ Input/Output

- Read in from a stream using the >> operator
 - `std::cin` is the standard input stream (like `stdin` in C)

- Reads a whitespace-delimited token from the stream and places result in the right-hand-side
- Takes a reference* to a stream and a reference* to a `string/int/etc.` as input
- Returns a reference* to the input stream

⇒ We can chain inputs

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string first , last;
    int age;
    std::cout << "Please enter your name and age: ";
    std::cin >> first >> last >> age;
    std::cout << "Hi: ";
    std::cout << last << ", " << first << ": " << age << std::endl;
    ret >> echo misha Kazhdan 25 | ./a.out
    Please enter your name and age: Hi: kazhdan, misha: 25
    >>
```

C++ Input/Output

- Read in from a stream using the >> operator
 - `std::cin` is the standard input stream (like `stdin` in C)

- Reads a whitespace-delimited token from the stream and places result in the right-hand-side
- Takes a reference* to a stream and a reference* to a `string/int/etc.` as input
- Returns a reference* to the input stream
- Input stream “evaluates” to true if it is in a "good" state (no error, no EOF)

```
main.cpp
#include <iostream>
#include <string>
int main( void )
{
    std::string word , earliest;
    while( std::cin >> word )
        if( earliest.empty() || word < earliest )
            earliest = word;
    std::cout << earliest << std::endl;
    return 0;
}
```

```
>> echo "the quick brown fox" | ./a.out
brown
>>
```

C++ Input/Output

- Read in from a stream using the *get* method
 - Extracts a single character from a stream
 - Takes a reference* to a char
 - Returns (a reference* to) the stream
 - Set the argument to the read in character

```
main.cpp
#include <iostream>
#include <locale>
int main( void )
{
    char ch;
    while( std::cin.get(ch) )
    {
        ch = std::toupper(ch);
        std::cout << ch;
    }
    return 0;
}
```

```
>> echo "the quick brown fox" | ./a.out
THE QUICK BROWN FOX
>>
```

Outline

- C++
- I/O
- Namespaces
- *strings*
- Review questions

Namespaces

- C++ has *namespaces*
 - In C, when two things have the same name, problems can result
 - In C++, items with same name can safely be placed in distinct “namespaces”
 - By default, classes / objects are defined within the *global* namespace
 - However, classes / objects provided by C++ are defined within the **std** namespace

Namespaces

- When doing I/O, we prefix names with `std::` to indicate who / where the name was defined
 - This is necessary because they are defined within the `std` namespace and we need to tell the compiler where to import them from

```
main.cpp
#include <iostream>

int main( void )
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Namespaces

- When doing I/O, we prefix names with `std::` to indicate who / where the name was defined
 - This is necessary because they are defined within the `std` namespace and we need to tell the compiler where to import them from
 - We can avoid specifying the namespace each time by telling the compiler which namespace we are **using**
 - `using std::cout`
⇒ when you see `cout`, assume it is `std::cout`

```
main.cpp  
#include <iostream>  
using std::cout;  
using std::endl;  
int main( void )  
{  
    cout << "Hello World!" << endl;  
    return 0;  
}
```

Namespaces

- When doing I/O, we prefix names with `std::` to indicate who / where the name was defined
 - This is necessary because they are defined within the `std` namespace and we need to tell the compiler where to import them from
 - We can avoid specifying the namespace each time by telling the compiler which namespace we are **using**
 - If we really want, we can import the entirety of the `std` namespace
 - ⇒ when you see something you don't recognize check if it's from the `std` namespace
 - Don't!
This is too broad and can cause confusion due to accidental name conflicts

```
main.cpp  
#include <iostream>  
using namespace std;  
  
int main( void )  
{  
    cout << "Hello World!" << endl;  
    return 0;  
}
```

Namespaces

- When doing I/O, we prefix names with `std::` to indicate who / where the name was defined
 - This is necessary because they are defined within the `std` namespace and we need to tell the compiler where to import them from
 - We can avoid specifying the namespace each time by telling the compiler which namespace we are **using**
 - If we really want, we can import the entirety of the `std` namespace
 - ⇒ when you see something you don't recognize check if it's from the `std` namespace

```
main.cpp  
#include <iostream>  
using namespace std;  
  
int main( void )
```

Do not use **using** in header files!!!

- Any code **#include**ing your header files will be forced to import the objects
- This may introduce new naming conflicts

Outline

- C++
- I/O
- Namespaces
- **strings**
- Review questions

C++ *strings*

`#include <string>`

- C++ strings provide user-friendly support
- Spare us most of the “nitty-gritty” of C strings
We still use C strings sometimes (e.g. `char *argv[]`)

C++ `strings`

Q: How long can a `std::string` be?

A: Arbitrarily long

Q: Who worries about the memory?

A: C++ library does

- “Backing” memory is dynamically allocated and adjusted as needed
- When `std::string` goes out of scope, associated memory is freed

C++ strings

- Initialization:
 - `std::string s1 = "world";`
 - initializes to "world"
 - `std::string s2("hello");`
 - just like `s2 = "hello"`
 - `std::string s3(3 , 'a');`
 - `s3` is "aaa"
 - `std::string s4;`
 - `s4` is the empty string ""
 - `std::string s5(s2);`
 - copies `s2` into `s5`

C++ strings

- Operators:
 - `s = "wow";`
 - assign literal to string
 - `std::cin >> s;`
 - put one whitespace-delimited input word in `s`
 - `std::cout << s;`
 - write `s` to standard out
 - `std::getline(is , s);`
 - read to end of line from input stream `is`, store in `s`
 - `s1 = s2;`
 - copy contents of `s2` into `s1`
 - `s1 + s2;`
 - return the string that is `s1` concatenated with `s2`
 - `s1 += s2;`
 - same as `s1 = s1 + s2`
 - `== != < > <= >=`
 - relational operators, using alphabetical (lexicographic) ordering

C++ strings

- Member functions:
 - `length()`;
 - The length of the string
 - `capacity()`;
 - The maximum string length that can be represented without (internal) reallocation
 - `substr(start, size)`
 - The substring starting at index **start** with length **size**
 - `c_str()`;
 - A `const char*` version of the string

main.cpp

```
#include <iostream>
#include <cstring>
int main( void )
{
    std::string s = "hello";
    std::cout << s.length( ) << std::endl;
    std::cout << s.capacity( ) << std::endl;
    std::cout << s.substr( 1 , 3 ) << std::endl;
    std::cout << strlen( s.c_str() ) << std::endl;
    return 0;
}
```

```
>> ./a.out
5
15
ell
5
>>
```

C++ strings

- Accessing:
 - `s[5];`
 - Gets the 6th character

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string s( "Nobody's perfect" );
    for( size_t pos=0 ; pos<=s.length() ; pos++ )
        std::cout << s[ pos ] << " ";
    std::cout << std::endl;
    return 0;
}
```

```
>> ./a.out
N o b o d y ' s   p e r f e c t
>>
```

C++ strings

- Accessing:
 - `s[5];`
 - Gets the 6th character
 - `s.at(5);`
 - Gets the 6th character but first checks that the memory access is in bounds

main.cpp

```
#include <iostream>
#include <string>
int main( void )
{
    std::string s( "Nobody's perfect" );
    for( size_t pos=0 ; pos<=s.length() ; pos++ )
        std::cout << s.at( pos ) << " ";
    std::cout << std::endl;
    return 0;
}
```

```
>> ./a.out
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::at: __n (which is 16) >= this->size() (which is 16)
N o b o d y ' s   p e r f e c t Abort (core dumped)
>>
```

C++ strings

- See C++ reference for more string functionality
 - www.cplusplus.com/reference/string/string/
- Don't miss:
 - `clear` – set to empty string
 - `append` – just like `+=`
 - `push_back` – like `append` for a single character
 - `insert` – insert one string in middle of another
 - `erase` – remove stretch of characters from string
 - `replace` – replace a substring with a given string

Outline

- C++
- I/O
- Namespaces
- *strings*
- Review questions

Review questions

1. What is the difference between C and C++?

Classes, templates, STL, overloading, more convenient text input & output

Review questions

2. What is a `namespace` in C++?

A context for types that allows us to use items with the same names without confusion / shadowing.

Review questions

3. Why should you not use `using` in header files?

Every source file that includes the header will necessarily be `using` the namespace.

Review questions

4. How do you read and write in C++ (i.e. standard inputting/outputting)?

`std::cout <<` and `std::cin >>`

Review questions

5. What is the difference between C strings and C++ strings?

No null terminators. Their own type. Don't have to worry about memory (allocation, reallocation, or deallocation). Supports operators like assignment, concatenation, and comparison.

Review questions

6. How long can a C++ `string` be?

As long as it needs to be (and the heap can support).