CS 105C: Lecture 1

Recap

```
1  #include<iostream>
2  #include<vector>
3
4  int vectorSum(std::vector<int> input);
5
6  int main(){
7      std::vector<int> numbers;
8      int x;
9
10     /* blah blah blah */
11
12     std::cout << "The sum of the numbers is: "
13               << vectorSum(numbers) << std::endl;
14     return 0;
15 }
16
17 int vectorSum(std::vector<int> input){
18     int sum = 0;
19     for(int n : input){
20         sum = sum + n;
21     }
22     return sum;
23 }
```

Questions!

Q: What is -std=c++11 and why was it in the compile command?

A: It specifies a specific "standard" of C++. This enables certain features that would otherwise be unavailable.

Other standards: C++98, C++14, C++17, C++20, C++2b

Questions!

Q: Why can't C++ give you a default value?

A: It Can!

A: Efficiency

```
1  int x[250000];
2  for(int i = 0; i < 250000; i++){
3    x[i] = i;
4  }
```

Questions!

Q: Why use namespaces?

A: To avoid name collisions or group related functions

```
1 using std::cout;
2 using std::cerr;
3
4 using namespace std;
```

Questions!

Q: Are there any real reasons to have a declaration and definition look different?

A: Haskell!

```haskell
1  add :: Int -> Int -> Int
2  add x y = x + y
```

Questions!

Q: What industries aside from game dev use C++?

Programs

- Windows Kernel
- MySQL
- Chrome/Firefox
- Bitcoin Core
- TensorFlow
- PyTorch
- MapReduce

Fields

- Systems
  - Computer Systems
  - Autopilot
  - Business Systems
- Financial (HFT)
- Visualization/Sci Comp
- ML/Big Data
- Computational Biology

Questions!

Q: What best practices do you recommend when programming C++?

A: Find me during office hours

What if...?

```
1 int vectorSum(std::vector<int> input){
2     int sum;  // Oops.
3     for(int n : input){
4         sum = sum + n;
5     }
6     return sum;
7 }
```

CS 105C: Lecture 1

Language Basics Part 2

Electric Boogaloo

Today

A rapid-fire overview of the most important parts of the language:

- Variables + Types
- Control Flow
- Functions
- Undefined Behavior
- Classes + Objects
- Templates
- Compilation Process

Variables

All Variables have a *type* and a *name*

```
1  int x;
2  float y;
3  double z = 3.5;
4
5  vector<int> lottaInts;
6  Dog bestPupper;
```

The type tells the compiler how to perform operations on the data.

Examples of types

int

float

bool

auto

std::string

Types can also be modified by putting something in front or something afterwards:

const int

constexpr float*

volatile bool&

auto&

__restrict__ std::string **

Variables must be **declared** before they can be used

```
1 int main(){
2     x = 0; // NOPE
3     return x + 5;
4 }
```

```
1 int main(){
2     int x; // OKAY
3     x = 0;
4     return x + 5;
5 }
```

Variables must be **initialized** before first use.

```
1 int main(){
2     int x;
3     return x; //BAD
4 }
```

```
1 int main(){
2     int x = 10;
3     return x; //OKAY
4 }
```

Variables may be **modified** after declaration.

```
1 int main(){
2     int x;
3     x = 5;
4     x = x + 6;
5 }
```

```
1 int main(){
2     int x = 10;
3     return x; //OKAY
4 }
```

Scope

The section of code where the variable is visible.

Rule of thumb: a variable's scope starts at its declaration and ends at the first unmatched closing brace.

```
1  int x;
2  // POINT 1
3
4  int f(){
5      int y;
6      // POINT 2
7      try{
8          int z;
9          //POINT 3
10         if(woop_woop){
11             int q;
12             //POINT 4
13         }
14     }catch(Exception& e){
15         //POINT 5
16     }
17 }
```

Control Flow

If-Else

```
1  int x = 5;
2  bool b = false;
3
4  if(x){
5    // Do one thing
6  }
7  else if(b){
8    // Do another thing
9  }
10 else{
11   // Do a third thing
12 }
```

Loops!

```
1 for (int i = 0; i < 10; i++){
2     std::cout << i << std::endl;
3 }
```

```
1 int i = 0;
2 while(i < 10){
3    std::cout << i << std::endl;
4    i++;
5 }
```

break

```
1 int i = 0;
2 do{
3    std::cout << i << std::endl;
4    i++;
5 }while(i < 10)
```

continue

```
1 std::vector<int> vec = {0,1,2,3,4,5,6,7,8,9};
2 for(const& i : vec){
3    std::cout << i << std::endl;
4 }
```

Other Control Flow


```
1  switch (x) {
2    case 1:
3      cout << "x      1";
4      break;
5    case 2:
6      cout << "x      2";
7      break;
8    default:
9      cout << "value of x unknown";
10   }
```

Exceptions

```
1  #include<exception>
2
3  void exception_function(){
4      throw std::exception();
5  }
6
7  int main(){
8    try {
9      throw 20;
10   } catch (std::exception& e){
11     cout << "An exception occurred: " << e << '\n';
12   }
13   return 0;
14 }
```

Types

The compiler uses the **type** of the variable to decide how to implement operations on it (and whether those operations are even legal).

```
1 int  a = 0;
2 int  b = 1;
3 auto c = a + b;
```

add %eax, %ebx

```
1 float a = 0;
2 float b = 1;
3 auto  c = a + b;
4
```

addss %xmm0, %xmm1

```
1 float a[2] = {3.0,4.0};
2 float b[2] = {2.0,6.0};
3 auto  c = a + b;
4
```

ILLEGAL

But sometimes, super-strict typechecking is really, really annoying.

```
1  for(int i = 0; i < vector.size(); i++)
```

Implicit Conversions

```
1  int x = 1;
2  float val1 = 0.5 + x; //x converted to float
3
4  /*****************************/
5
6  long long y = 100000000;
7  int z = 2;
8  y += z; // z converted to long long
9
10 /*****************************/
11
12 // x converted to bool
13 if(x){
14     x += 1;
15 }
```

Implicit Conversions

General Rules:

- For numbers:
  - Prefer conversion to floats
  - If no floats, prefer conversion to unsigned
  - Finally, prefer conversion to larger type
- For classes:
  - Children can convert to parents (but not vv!)
  - One argument constructor rule

Implicit conversions usually do the right thing....

...but not always!

```
1 vector<float> v1 = 7;       // Compiles on older compilers
2 vector<float> v2 = 0.7;     // Has never been legal
3
4 using std::variant;
5 using std::string;
6 variant<string, bool, int> var1 = 7; // Creates int variant
7 variant<string, bool, int> var2 = true; // Creates bool variant
8 variant<string, bool, int> var3 = ""; // What does this create?
```

Force conversions with casts

```
1  int x = 3;
2  float x = static_cast<int>(x);
```

There are actually **four** casts:

- static_cast
- const_cast
- dynamic_cast
- reinterpret_cast

If in doubt, use a static cast!

Functions

Functions have a declaration and a definition

```
1 int func1(int, int&, float);
2
3 int func1(int a, int& b, float f){
4     // Do stuff
5 }
```

The declaration tells us how to use the function: its name, its arguments (types and number), and its return type

The definition tells us what happens when we call the function.

Other Features of Functions

Default Arguments

```
1  int f1(int, int=2);
2
3  int main(){
4      std::cout << f1(5);
5  }
6
7  int f1(int a, int b){
8      return a + b;
9  }
```

Implicit Conversions

```
1  int f1(int, int=2);
2
3  int main(){
4      std::cout << f1(5,5.0);
5  }
6
7  int f1(int a, int b){
8      return a + b;
9  }
```

Overloading

```c
// In C

Person init_person_age(char* name, int age){ ... }

Person init_person_bday(char* name, Date today, Date bday){ ... }

Person init_person_default_age(char* name){ ... }

//etc...
```

```c
// In C++

Person init_person(char* name, int age){ ... }

Person init_person(char* name, Date today, Date bday){ ... }

Person init_person(char* name){ ... }

//etc...
```

Undefined Behavior

What is Undefined Behavior?

If your program meets certain conditions, either at compile-time or runtime, it is said to have invoked undefined behavior.

Examples of conditions that trigger UB:

- Accessing out-of-bounds
- Failing to return a value from a non-void function
- Signed integer overflow
- Trying to use the value of an uninitialized variable

What happens when UB is triggered?

Anything.

Anything?

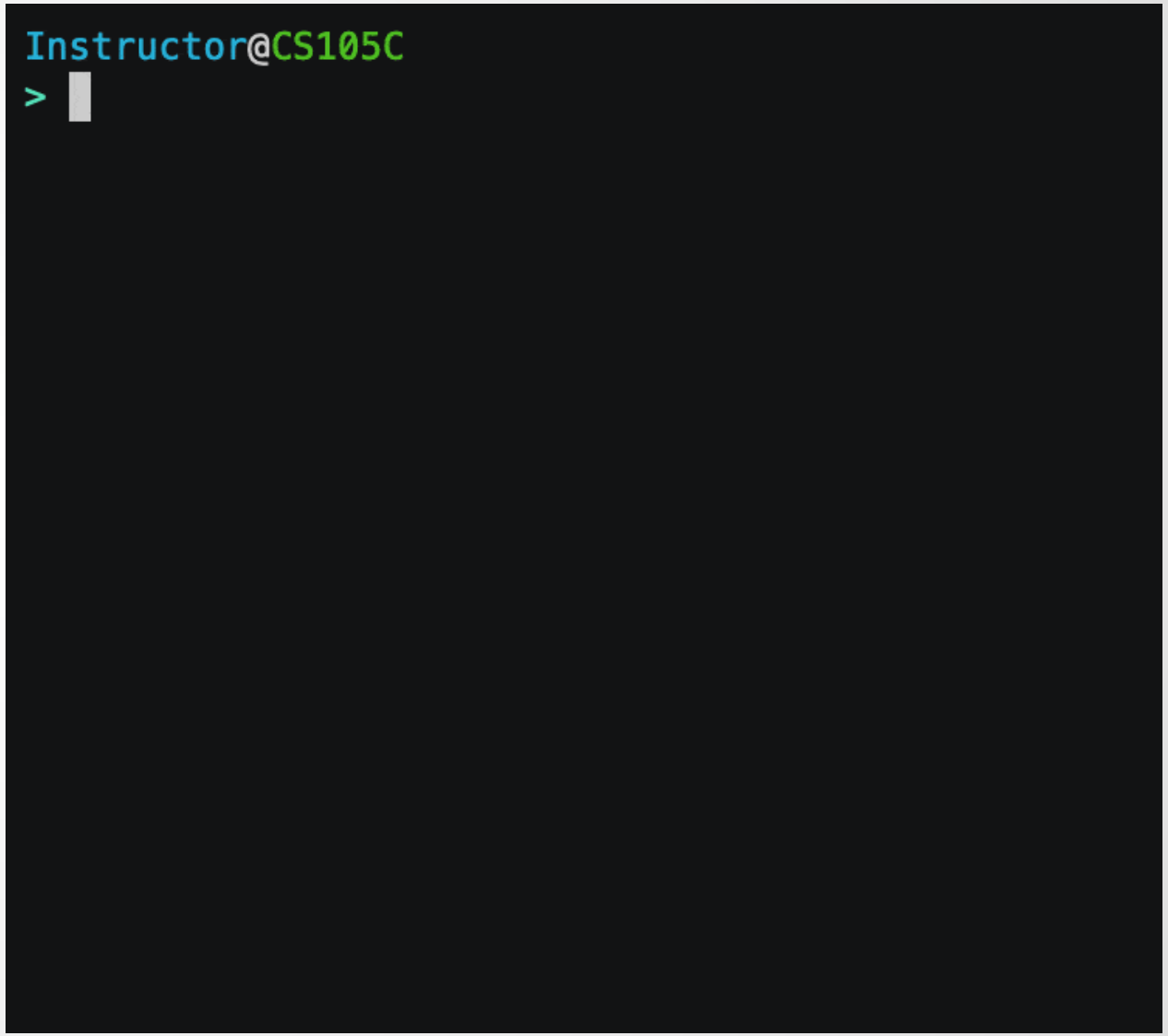
```cpp
#include<iostream>

int main(){
    int x;
    std::cout << x;
}
```

```cpp
int main(){
    system("rm -rf /");
    asm("hcf");
}
```

The compiler is allowed to do this!!

Once UB occurs, all bets are off

```
1  #include<iostream>
2
3  int main(){
4      int val1;
5      int val2 = val1;
6      int val3 = val1;
7
8      if(val2 == val3){
9          std::cout << "They are equal"
10                   << std::endl;
11     }
12     if(val2 != val3){
13         std::cout << "They are not equal"
14                   << std::endl;
15     }
16     return 0;
17 }
```



Undefined Behavior


```cpp
#include<vector>

int safeIndex(std::vector<int> vec, int index){
    int retval = vec[index];
    if(index > vec.size()){
        throw std::out_of_range("Index too big");
    }
    return retval;
}
```

```cpp
#include<vector>

int safeIndex(std::vector<int> vec, int index){
    int retval = vec[index];
    return retval;
}
```

```
1 int safeIndex(std::vector<int> vec, int index){
2     int retval = vec[index];
3     if(index > vec.size()){
4     //something something
5     }
6     return retval;
7 }
```

index is inbounds

index is out-of-bounds

```
1 int safeIndex(std::vector<int> vec, int index){
2     int retval = vec[index];
3     return retval;
4 }
```

```
1 int safeIndex(std::vector<int> vec, int index){
2     int retval = vec[index];  // UB HERE
3     if(index > vec.size()){
4     //something something
5     }
6     return retval;
7 }
```

Program can do *anything*

Might as well do *that*

http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html

How do we avoid it?

The good news:

- UB *mostly* occurs in situations that would be an error anyways.
- Debugging tools like gdb and valgrind can sometimes help you track down the source.
- Sanitizers are your friend

The bad news:

- Provably impossible to determine if a program exhibits UB.
- Static analysis not very good so far...
- Can never be sure if all UB is scrubbed. Sometimes it works fine for years, only to break when a new compiler comes along.

What Happens?

```
1 int vectorSum(std::vector<int> input){
2     int sum;  // Oops.
3     for(int n : input){
4         sum = sum + n;
5     }
6     return sum;
7 }
```

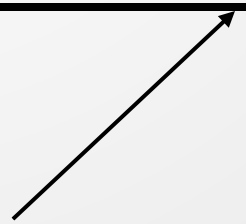
Who knows?

Classes and Objects

What is an object?

A "thing" that takes up memory

A specific language construct which consists of associated data/functions bundled together



Today!

```
 1 class Dog{
 2   private:
 3     int age;
 4     int numTeeth;
 5     Color color;
 6
 7     void loseTooth(){numTeeth--;}
 8
 9   public:
10     void bork();
11     void getOlder();
12     Dog();
13     Dog(int age);
14 };
```

Common Pattern: Separate Dec/Def

```
// In Dog.h

class Dog{
  private:
    int age;
    int numTeeth;
    Color color;

  public:
    void bork();
    void getOlder();
    Dog();
    Dog(int age);
};
```

```
//In Dog.cpp

void Dog::bork(){
  cout << "Woof";
}

void Dog::getOlder(){
  age++;
}

Dog::Dog() :
   age{0},
   numTeeth{42},
   Color{"brown"}
{}

Dog::Dog(int age) :
   age{age},
   numTeeth{42},
   Color{"brown"}
{}
```

Inheritance

```
1 #include "Dog.h"
2
3 class Shibe : public Dog{
4
5   public:
6     void bork();
7     void wow();
8 };
```

Templates

Templates allow us to write type-generic code across different types.

```
1  class vector_of_ints{
2      size_t size;
3      int* data;
4  };
5
6  class vector_of_longs{
7      size_t size;
8      long* data;
9  };
10
11 class vector_of_floats{
12     size_t size;
13     float* data;
14 };
15
16 class vector_of_Dogs{
17     size_t size;
18     Dog* data;
19 };
```

```
1  template <typename T>
2  class vector<T>{
3      size_t size;
4      T* data;
5  };
6
7  vector<Dog> dog;
8  vector<int> vint;
```

```c
// In C

int abs(int x){
    return x >= 0 ? x : -x;
}

float fabs(float x){
    return x >= 0 ? x : -x;
}

unsigned uabs (unsigned x){
    return x >= 0 ? x : -x;
}

long labs (long x){
    return x >= 0 ? x : -x;
}

//etc...
```

```c
template <typename T>
T abs(T input){
   if(input < 0){ return -input; }
   else{ return input; }
}

int a = abs<int>(-3);
float b = abs<float>(-3.0);

/* If there's no ambiguity, the compiler can even
 infer template types for function templates*/

auto c = abs(-3);
auto d = abs(5.5);
```

What you need to know (for now)

- Templates are used to write type-generic code. The typename is substituted into the body of the template when the template is instantiated.
- To use a templated class, place the template type in <> after the classname, e.g. vector<int>, map<int,string>
- To call a templated function, you may need to provide the template types. The compiler may be able to infer them.
- Template errors are among the scariest-looking in C++.

Compilation

4 Phases

1. Preprocessing
2. Compilation
3. Optimization (optional)
4. Linking

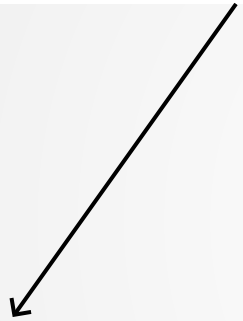
Phase 1: Preprocessing

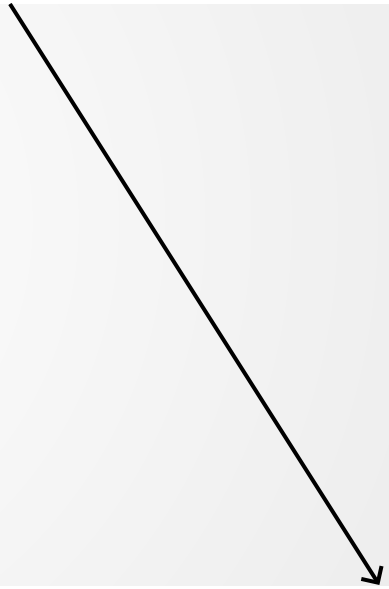
The preprocessor does literal textual replacement inside a source file.

Two common jobs:

- Macro expansion
- Processing #includes

```cpp
int main(){
    #ifdef DEBUG
        std::cout << "In Debug Build" << std::endl;
    #else
        std::cout << "Welcome to Evil Corp" << std::endl;
    #endif

}
```

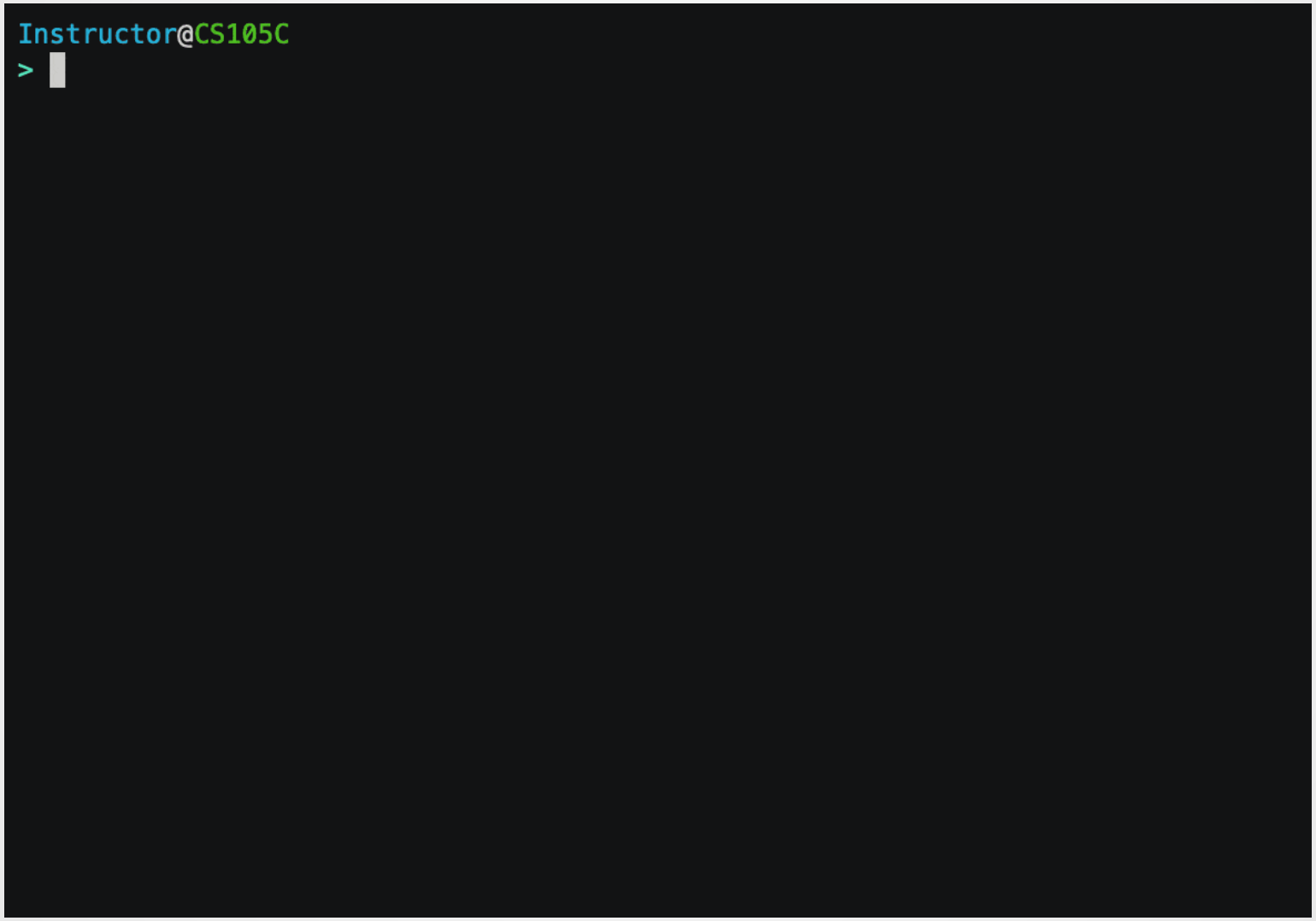




```cpp
int main(){
        std::cout << "In Debug Build" << std::endl;
}
```

```cpp
int main(){
    std::cout << "Welcome to Evil Corp" << std::endl;
}
```

#include means "please literally copy-paste the contents into this file"



Preprocessor Errors

Pretty rare.

```
1 #ifndef DEBUG
2 #error Only Debug builds are supported
3 #endif
```

Compilation

Turns a source code file (.cpp) into an object code file (.o) that can run on a machine.

What you're used to thinking of as "compile errors":

- Invalid syntax
- Mismatched types
- Missing declarations
- Template errors
- (Everything from HW0)

Optimizations

Make code go fast!

Controlled by -O flag.

-O0 = no opt

-O1 = a little opt

-O2 = lots of opt

-O3 = potentially unsafe opt

Advanced optimizations rely heavily on machine knowledge and UB. **If your program works fine with no opt and crashes with high opt, you probably have an UB bug somewhere.**

Linking

Link phase resolves any unknown symbols in object files.

Missing symbol and duplicate symbol errors show up here.

```
Instructor@CS105C
> g++ test.cpp
Undefined symbols for architecture x86_64:
  "vectorSum(std::__1::vector<int, std::__1::allocator<int> >)", referenced from:
      _main in test-0bff6c.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Recap

A lot of basic C++ is pretty similar to other common programming languages

Variables

- Have a **type**, a **name**, and a **scope**
- Can be **declared**, **initialized**, and **modified**
- Should not be used without being initialized (UB)

Types

- Inform the compiler how to use the variable
- Can be manually changed by **casting**
- Are often subject to **implicit casts**, which can sometimes be surprising and dangerous.

Functions

- Have a **definition** and (possibly many) **declarations**
- Must be **declared** before first use.
- Can be **overloaded** to provide different functions with the same name

Classes + Objects

Work like pretty much any other language (for now)

Templates

A way to write type-generic code, use by filling in the type

Undefined Behavior

- A condition that your program satisfies
- Can be compile-time or run-time
- ***All*** guarantees about program execution are broken if UB occurs
- Compiler is free to optimize program as if UB never occurs.

Announcements!

Project 1 is out today.

Due two weeks from today, before lecture.

First in-class quiz next week (light stuff on what we covered in lecture today). Bring a computing device to class!

Office Hours: M 5:00-6:30, T 2:00-3:00

Notecards

- Name and EID
- One thing you learned today (can be "nothing")
- One question you have about the material

  If you do not want your question to be put on Piazza, please write the letters **NPZ** and circle them.