CSE333 SECTION 6

Upcoming Due Dates

HW3 Due – Nov. 14th

Remember Const?

- Const means you cannot change the value

EX:

```
const int m = 333;
m++;  // Compiler Error
```

Possible Output:

const_error.cc:5:4: error: read-only variable is not assignable

```
m++;
```

So… Does it work?

```
void someFn(const int x) {
  x++;
}
```

Nope.

const_error.cc:5:4: error: read-only variable is not assignable

m++;

So… Does it work?

```
void someFn(int const x) {
  x++;
}
```

Nope.

const_error.cc:5:4: error: read-only variable is not assignable

m++;

So… Does it work?

```
void someFn(const int *ptr) {
 ptr++;
}
```

Yup.

So... Does it work?

```
void someFn(const int *ptr) {
 (*ptr)++;
}
```

Nope.

const_error.cc:4:9: error: read-only variable is not assignable

(*ptr)++;

So… Does it work?

```
void someFn(int *const ptr) {
 ptr++;
}
```

Nope.

const_error.cc:4:8: error: read-only variable is not assignable

(ptr)++;

So… Does it work?

```
void someFn(int *const ptr) {
  *ptr++;
}
```

Yup.

So… Does it work?

```
void someFn(int const * const ptr) {
  ptr++;
}
```

Nope.

const_error.cc:4:8: error: read-only variable is not assignable

(ptr)++

So… Does it work?

```
void someFn(int const * const ptr) {
  *ptr++;
}
```

Nope.

const_error.cc:4:8: error: read-only variable is not assignable

(ptr)++

What about…

```
int x = 333;
const int *ptr = &x;

int someFn(int *ptr) {
 ptr++;
 *ptr++;
}
```

Nope.

const_error.cc:3:6: note: candidate function not viable: 1st argument

('const int *') would lose const qualifier

void someFn(int *ptr) {

Passing const vars to non-const fn

You can’t do it. (Well you can, but don’t)

Breaking promises

But… *Sigh* We don't have to keep that promise

const_cast: Used to strip or add const-ness

```
void someFn(const int *x) {
  foo(x); // Bad
  foo(const_cast<int *>(x));  // Okay
}
```

So why const_cast?

Examples:

- You know a function will not change the state of your variables, but is declared non-const
- ???

Really. You probably just shouldn’t.

References

- The reference becomes an alias for the referenced variable
- You Cannot change what a reference refers to

Example

```
int x = 333;
int &y = x;

y = 344;  // x = y = 344
```

So… What does it do?

```
#include <iostream>
using namespace std;
int main(int argc, char **argv) {
  int i = 333;
  int &j = i;
  int &k = j;
  int &l = k;
  cout << i << ", " << j << ", " << k << ", " << l << endl;
  k++;
  cout << i << ", " << j << ", " << k << ", " << l << endl;
  j = 0;
  cout << i << ", " << j << ", " << k << ", " << l << endl;
  return 0;
}
```

It outputs

333, 333, 333, 333

334, 334, 334, 334

0, 0, 0, 0

So… Does it work?

```
int main(int argc, char **argv) {
  int x = 333;
  int y[2] = {1, 2};
  int z[2] = {3, 4};
  int *a[] = {y, z};

  int & b[] = a;

  return 0;
}
```

Nope.

C++ Standard 8.3.2/4:

There shall be no references to references, no arrays of references, and no pointers to references.

Why?

Indexing into an array is done using pointer arithmetic. But, pointers to references aren't defined nor is pointer arithmetic.

So… Does it work?

```
int main(int argc, char **argv) {
  int x = 333;
  int &y = x;
  int *z = &y;

  return 1;
}
```

Yep!

But, you just told me pointers to references aren't defined.

Recall: The reference becomes an alias for another variable.

From the previous slide:

If I print the address of x (the int) and z (the int pointer) I get

0x7fff53abfb7c

0x7fff53abfb7c

So… What is the size of a reference?

```
int main(int argc, char **argv) {
  int x = 333;
  int &y = x;

  cout << "size of a reference = " << sizeof(y) << endl;

  return 0;
}
```

The size of the variable it points to

4

So… What is the size of a class w/ references

```
#include <iostream>


class Test {
  public:
  int &i, &j, &l;
};
using namespace std;
int main(int argc, char **argv) {
  cout << "size of class Test = " << sizeof(class Test) << endl;
  return 0;
}
```

The size of 3 pointers!

24

What????

References are implemented using pointers.

So… Is the previous code useless?

Nope.

We'll see that those variables can still be initialized using initialization lists in classes.

C++ Classes

- Yes, there are actually classes.

Class Declaration Format:

```
Class Name {
  public:
    members;
    // Includes public variables, functions, …
  private:
    members;
    // Includes private variables, functions, …
};  // Note the semi-colon here!!!!!!
```

Constructor

```
ClassName::ClassName(parameters) {
  code;
}
```

Constructor with Initialization List

```
ClassName::ClassName(parameters) : field_(value),
                                   field_(value), …, field_(value) {
  code;
}
```

Member Function Declaration

```
returnType classname::functionName(parameters) {
  statements;
}
```

What do you think it means?

```
returnType classname::functionName(parameters) const  {
  statements;
}
```

What do you think it means?

```
returnType classname::functionName(parameters) const {
  statements;
}
```

It's a promise that this function call does not modify that state of the object.

Inlining

- Inline functions are like placeholders for the actual code that goes there
- Compiler replaces all the inline function calls with the actual code

How to use it: Add inline to the function declaration

Example:

```
inline void cse() {
  cout << “333” << endl;
}
```

So why inline?

Pros:

- It’s faster!
  - Why? Because function calls are more expensive than just executing some statements

Cons:

- Your file could become huge!
  - Copy paste a large function’s code tons of times.