

# Optimization in C++

---

With Examples From Project 1

# What we will cover

---

- Code organization
- Caching and Memory locality
- Data duplication
- Pass by reference vs pass by value
- String functions
- Over-optimization

# Code Organization – Objects

---

- If it goes together, then put it together
  - Keep relevant data in the same place
- From project 1 - Some kind of word structure
  - Actual string
  - Parent index
  - Modification to get from parent to this word
- If we group relevant data together we can take advantage of caching
- Cache is faster than RAM, and keeps copies of recently used RAM memory (we will discuss this in more detail later)

# Code Organization – Classes & Structs

---

- There is no difference in memory or time efficiency
- Only difference is that members default to public in structs, and private in classes
- When to use a struct vs a class?
  - Different people will tell you different things
    - I use structs if I just want to group some data together and access it
    - I use classes if I want it to represent some more complex group of data and functionality

# Code Organization – Inlining

---

- Instead of making a function call, compiler pastes code where the function call was
- Improves runtime efficiency because nothing needs to be placed on the call stack
- Makes executable bigger
- Functions from .h files can be inlined by any code that calls them
  - Functions from .cpp files are limited to functions in same file


# Code Organization – Inlining

main.cpp

```
Reader reader(begin, end, permitted_mods.length);  
vector<Word> words = reader.Read_input();  
int begin_index = reader.get_begin_index();  
int end_index = reader.get_end_index();
```

Reader.h

```
int get_begin_index() { return begin_index; }  
  
int get_end_index() { return end_index; }
```



```
Reader reader(begin, end, permitted_mods.length);  
vector<Word> words = reader.Read_input();  
int begin_index = reader.begin_index;  
int end_index = reader.end_index;
```

# Code Organization – Files

---

- Review from 280
  - Declarations go in .h files
  - Implementations go in .cpp files
    - An exception is for single line functions which can be inlined

# Code Organization – Files

```
class TraversalList {  
public:  
    TraversalList(bool stack_mode_, int end_index_) : end_index(end_index_),  
        stack_mode(stack_mode_), end_found(false), words_checked(0) {}  
  
    void push(int i);  
  
    void pop();  
  
    int next();  
  
    bool empty() { return list.empty(); }  
  
    bool found_end() { return end_found; }  
  
    unsigned Words_checked() { return words_checked; }  
  
private:  
    unsigned end_index;  
    bool stack_mode;  
    bool end_found;  
    unsigned words_checked;  
    std::deque<int> list;  
};
```

```
void TraversalList::push(int i) {  
    list.push_back(i);  
    words_checked++;  
    if (i == end_index) {  
        end_found = true;  
    }  
}  
  
void TraversalList::pop() {  
    if (stack_mode) {  
        list.pop_back();  
    }  
    else {  
        list.pop_front();  
    }  
}  
  
int TraversalList::next() {  
    if (stack_mode) {  
        return list.back();  
    }  
    else {  
        return list.front();  
    }  
}
```



# Code Organization – Functions

---

- Having one point of maintenance is extremely helpful

```
void Reader::add_word(vector<Word>& words, const string& word) {  
    if (length_change_permitted || word.size() == begin_word.size()) {  
        words.push_back(Word(word));  
        check_word_for_begin_end(word, words.size() - 1);  
    }  
}
```

- I call this function 10 times
  - Only a few lines but makes it trivially easy to add things like optimization for when length mode is not on

# Caching and Memory Locality

---

- Cache keeps copies of recently used RAM memory
  - Accessing cache is faster than accessing RAM
- Data in one instance of an object is held in the block of memory
  - This makes accessing everything else in the object faster after accessing just one thing



# Caching and Memory Locality

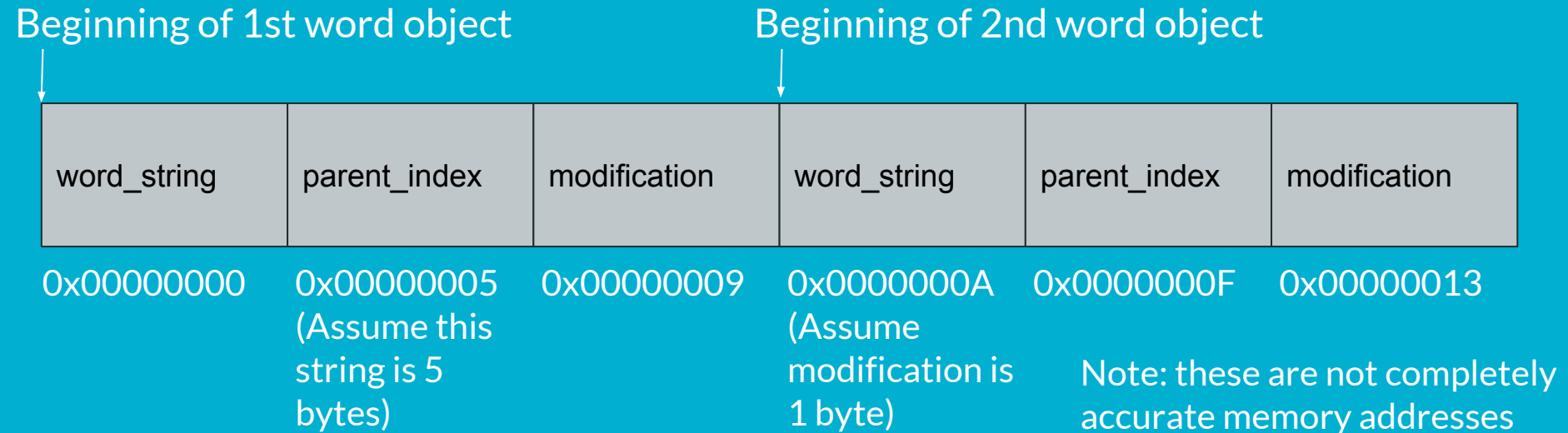
---

- What STL structure do we use a lot that can take advantage of caching?  
Vectors!
- If we store Word objects in a single vector, we can access the parent index and modification very easily once we try to access the string
- What if we store the string in one vector, and the parent index in a different vector?
  - We load the memory near the string after we access it, but then the index is not in the cache so we have to go to RAM to get it

# Caching and Memory Locality

---

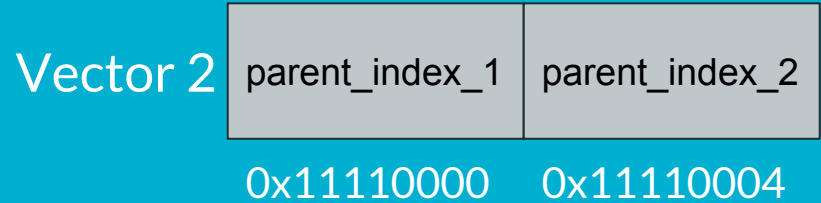
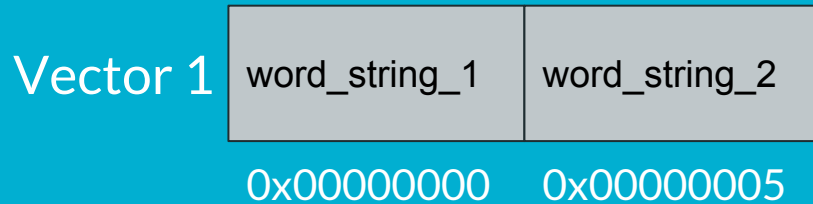
- If we store Word objects in a single vector, we can access the parent index and modification very easily once we try to access the string



# Caching and Memory Locality

---

- What if we store the string in vector 1, and the parent index in a vector 2?
  - We load the memory near the string after we access it, but then the index is not in the cache so we have to go to RAM to get it



# Data Duplication

---

- Things that take up lots of memory should be stored in as few locations as possible
- In Project 1, we only want 1 copy of each string and relevant data
  - Access everything else by references or pointers

# Data Duplication

---

- What do we mean by use the reference of a variable?

```
Word w;  
Word& w_ref = w;
```

- “w\_ref” is a reference to “w”
  - If you change “w\_ref”, you also change “w” and vice versa
- This is an explicit reference, but we can also “reference” a single spot in a vector with an integer index
  - Words in project 1 were referenced through their index in the dictionary

# A Note About Memory

---

- The size of pointers depend on the system
  - 32 bit systems have 32 bit pointers (4 bytes)
  - 64 bit systems have 64 bit pointers (8 bytes)
  - 8 bytes on CAEN

Type	Bytes
<u>size_t(g++)</u>	8
unsigned long	8
long	8
<u>size_t(VS)</u>	4
unsigned <u>int</u>	4
<u>Int</u>	4
unsigned short	2
short	2
unsigned char	1
char	1
long double(g++)	16
long double(VS)	8
double	8
float	4



# A Note About Memory

---

- It is very easy for objects to become as large as or larger than a pointer
  - 8 letter c-string  $\rightarrow 8 \text{ chars} * 1 \text{ byte/char} = 8 \text{ bytes}$
  - Struct with 2 ints  $\rightarrow 2 \text{ ints} * 4 \text{ bytes/int} = 8 \text{ bytes}$
  - From project 1  $\rightarrow$  Word structure could have string, integer index of parent, modification information (type, index, letter)
    - Say the average string is small (only 5 letters long)  $\rightarrow 5 \text{ bytes}$
    - Integer index of parent  $\rightarrow 4 \text{ bytes}$
    - Character modification type  $\rightarrow 1 \text{ byte}$
    - Integer modification index  $\rightarrow 4 \text{ bytes}$
    - Char modification letter  $\rightarrow 1 \text{ byte}$

# A Note About Memory

---

- All of that data in our Word class was 15 bytes
  - This was not including any C++ string metadata and assuming we have short strings
- A small Word class is essentially 2X the size of a pointer
- Is a Word or Word\* cheaper to copy????

THE ONE THAT TAKES LESS SPACE!!(Word\*)

# Pass by Reference vs Pass by Value

---

- We know that we can modify outside variables when passed by reference but how does this work?
  - C++ essentially passes a pointer
  - Does some fancy things with syntax so that you can use the “.” operator rather than the “->” operator
- Suppose we have this Word structure

```
struct Word {  
    string word;  
    unsigned parent_index;  
    char modtype;  
    unsigned index;  
    char letter;  
};
```

# Pass by Reference vs Pass by Value

---

```
bool Words_similar_ref(const Word& a, const Word& b, bool length, bool change, bool swap) {  
    if (length) {  
        // check length similarity  
        // return true if words are similar  
    }  
    if (swap) {  
        // check swap similarity  
        // return true if words are similar  
    }  
    if (change && a.word.size() == b.word.size()) {  
        return Check_change(a, b);  
    }  
    return false;  
}
```

# Pass by Reference vs Pass by Value

---

```
bool Words_similar_ptr(Word* a, Word* b, bool length, bool change, bool swap) {  
    if (length) {  
        // check length similarity  
        // return true if words are similar  
    }  
    if (swap) {  
        // check swap similarity  
        // return true if words are similar  
    }  
    if (change && a->word.size() == b->word.size()) {  
        return Check_change(*a, *b);  
    }  
    return false;  
}
```

This is functionally the same as passing by reference, but has different syntax

# Pass by Reference vs Pass by Value

---

- What about pass by value?
  - We want a variable that has the same data as argument
  - Also want changing this variable to have no effect on value passed in
  - Need a copy of the argument (could use a const reference if we don't plan on making any changes)

# Pass by Reference vs Pass by Value

```
bool Words_similar_ref(const Word& a, const Word& b, bool length, bool change, bool swap) {  
    if (length) {  
        // check length similarity  
        // return true if words are similar  
    }  
    if (swap) {  
        // check swap similarity  
        // return true if words are similar  
    }  
    if (change && a.word.size() == b.word.size()) {  
        return Check_change(a, b);  
    }  
    return false;  
}
```

```
if (change && a.word.size() == b.word.size()) {  
    Word a_copy;  
    a_copy.index = a.index;  
    a_copy.letter = a.letter;  
    a_copy.modtype = a.modtype;  
    a_copy.parent_index = a.parent_index;  
    a_copy.word = a.word;  
  
    Word b_copy;  
    b_copy.index = b.index;  
    b_copy.letter = b.letter;  
    b_copy.modtype = b.modtype;  
    b_copy.parent_index = b.parent_index;  
    b_copy.word = b.word;  
    return Check_change(a_copy, b_copy);  
}
```

```
bool Check_change(Word a, Word b)
```

# When to Pass By Reference?

---

- In previous classes, you pass by reference if you want to change the value of a variable inside a function
- Now, we also want to be fast and not use too much memory
  - This takes us back to the memory chart



# When to Pass By Reference?

---

- A pointer on most of your computers is 8 bytes
- Most primitive types are  $\leq 8$  bytes
  - This means that passing primitive types by reference does NOT save time or memory
- For things that aren't primitive types, even if they seem to be only a few bytes, their bookkeeping data makes it so that they can take up a lot of space

Type	Bytes
<u>size_t</u> (g++)	8
unsigned long	8
long	8
<u>size_t</u> (VS)	4
unsigned <u>int</u>	4
<u>Int</u>	4
unsigned short	2
short	2
unsigned char	1
char	1
long double(g++)	16
long double(VS)	8
double	8
float	4

# String Functions

---

- Common string functions
  - `operator==`
  - `operator=`
  - `erase()`
  - `insert()`
  - `operator+`
  - `substr()`

# String Functions - operator==

---

- How do you compare if two words are equal?
  - Go through every character and make sure each one is the same
- This operator must do that check for each character
- $O(n)$  complexity

# String Functions - operator==

```
// returns the index of the swap
// or -1 if the words are not similar
int Find_swap_with_operator(string& first, string& second) {
    for (unsigned i = 0; i < first.size(); i++) {
        if (first[i] != second[i]) {
            swap(first[i], first[i + 1]);

            if (first == second) {
                swap(first[i], first[i + 1]);
                return i;
            }

            swap(first[i], first[i + 1]);
        }
    }
    return -1;
}
```

The equality check is essentially adding a full while loop

```
// returns the index of the swap
// or -1 if the words are not similar
int Find_swap_with_loop(string& first, string& second) {
    for (unsigned i = 0; i < first.size(); i++) {
        if (first[i] != second[i]) {
            swap(first[i], first[i + 1]);

            bool strings_equal = true;
            for (unsigned i = 0; i < first.size(); i++) {
                if (first[i] != second[i]) {
                    strings_equal = false;
                    break;
                }
            }

            if (strings_equal) {
                swap(first[i], first[i + 1]);
                return i;
            }

            swap(first[i], first[i + 1]);
        }
    }
    return -1;
}
```

# String Functions - operator=

---

- How do you make one string from another string?
  - Go through the string on the right hand side and copy every character
- $O(n)$  complexity

# String Functions - operator=

---

```
// goes through two strings and returns true if they are equal
bool strings_equal_assignment(const string& first, const string& second) {
    for (unsigned i = 0; i < first.size(); i++) {
        string first_copy = first;

        if (first[i] != second[i]) {
            return false;
        }
    }
    return true;
}
```

```
// goes through two strings and returns true if they are equal
bool strings_equal_loop(const string& first, const string& second) {
    for (unsigned i = 0; i < first.size(); i++) {
        string first_copy;
        for (unsigned j = 0; j < first.size(); j++) {
            first_copy.push_back(first[j]);
        }

        if (first[i] != second[i]) {
            return false;
        }
    }
    return true;
}
```

These are essentially the same! The function on the left looks like it's  $O(n)$ , but the string copy makes it  $O(n^2)$

# String Functions – erase() and insert()

---

- What is the underlying structure of a string?
  - Array
- If you insert or erase something in an array what do you have to do?
  - Move everything behind the letter being inserted/deleted
- $O(n)$  complexity

# String Functions – erase() and insert()

```
// returns index of length change
// returns -1 if the words are not similar
int Find_length_change_erase(string& shorter, string& longer) {
    for (unsigned i = 0; i < longer.size(); i++) {
        char temp = longer[i];
        longer.erase(longer.begin() + i);

        if (longer == shorter) {
            return i;
        }
        longer.insert(longer.begin() + i, temp);
    }
    return -1;
}
```

These are essentially the same!  
Extra two  $O(n)$  loops that are  
happening in string functions!!!

```
// returns index of length change
// returns -1 if the words are not similar
int Find_length_change_loop(string& shorter, string& longer) {
    for (unsigned i = 0; i < longer.size(); i++) {
        char temp = longer[i];

        for (unsigned j = i; j < longer.size(); j++) {
            longer[j] = longer[j + 1];
        }

        if (longer == shorter) {
            return i;
        }

        // assume that the we still have an
        // extra spot left in longer after erase
        for (unsigned j = longer.size(); j > i; j--) {
            longer[j] = longer[j - 1];
        }
        longer[i] = temp;
    }
    return -1;
}
```



# String Functions – operator+

---

- How do you make one string from two strings?
  - Go through the two strings on the right hand side and copy every character from each string
- $O(n)$  complexity

# String Functions – substr()

---

- This function *returns a string*!!!!
  - Creates a partial string from an existing string
- Therefore, substr() is usually also  $O(n)$

# String Functions – substr()

---

```
// return index of extra letter in longer string
int Find_length_change_substr(const string& shorter, const string& longer) {
    for (unsigned i = 0; i < longer.size(); i++) {
        string potential = longer.substr(0, i - 1) + longer.substr(i + 1);

        if (potential == shorter) {
            return i;
        }
    }
    return -1;
}
```

```
// return index of extra letter in longer string
int Find_length_change_loop(const string& shorter, const string& longer) {
    for (unsigned i = 0; i < longer.size(); i++) {
        string potential;
        for (unsigned j = 0; j < i; j++) {
            potential.push_back(longer[j]);
        }
        for (unsigned j = i + 1; j < longer.size(); j++) {
            potential.push_back(longer[j]);
        }

        if (potential == shorter) {
            return i;
        }
    }
    return -1;
}
```

This is very similar to what happens with erase() and insert()

# Over-optimization – Inlining

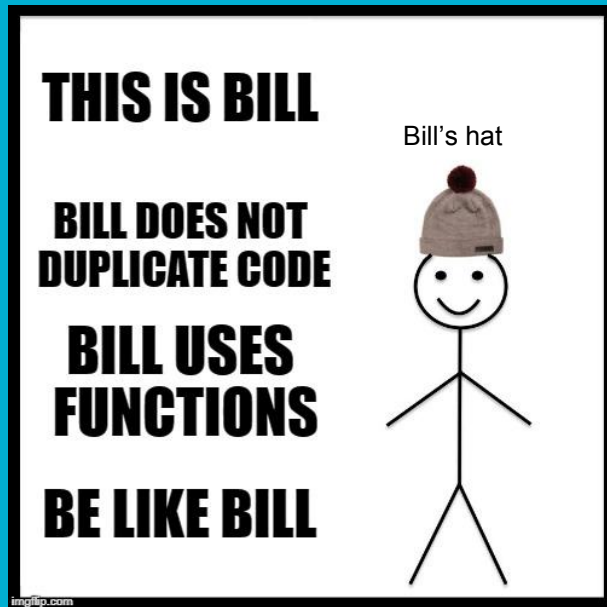
---

- Inlining can be more efficient, but don't sacrifice readability for the slight gain in efficiency!
- Keep functions that call each other often in the same .cpp file only if it makes sense (.cpp files should correspond to class or group of classes/functionality)
- Don't avoid using multiple .cpp files to try to make things fast!

# Over-optimization – Code Organization

---

- DON'T AVOID MAKING FUNCTIONS TO TRY TO MAKE YOUR CODE FASTER



# Over-optimization – Dense Coding

---

- With primitive types, the compiler can make optimizations so that when your code runs it has fewer steps than the way you actually wrote it
  - This makes it so that it is worth doing computation in parts and storing it in a variable if they are all done on primitive types

# Over-optimization – Dense Coding

---

- Code on the top is easier to read, and will be compiled to something similar to the bottom

```
double cone_area(double r, double h) {  
    double base_area = PI * r * r;  
  
    double slant_length = r + sqrt(h * h + r * r);  
    double slant_area = PI * r * slant_length;  
  
    return base_area + slant_area;  
} // cone_area()
```

```
double cone_area2(double r, double h) {  
    return PI * r * r + PI * r * (r + sqrt(h * h + r * r));  
} // cone_area2()
```

(From “Optimization Tips” doc on Canvas)

# Wrapping Up

---

- We now (hopefully) understand why the optimization tips we give you are relevant
- Can rank each of the optimizations from highest to lowest effect on your program runtime and memory
- I tried to make these general enough to be applicable to most projects, but do not use the following ranking as an absolute truth! It is meant to help you to direct your optimization efforts to the right areas



# Wrapping Up

---

1. Big O complexity
2. Passing very large objects by value (like the dictionary from P1) or passing objects by value in functions that are called extremely often (functions checking for word similarity from P1).
  - a. Recall from examples that copying things can even affect the big O complexity of a program
3. Using STL functions that do some expensive operation (specifically thinking functions discussed) → This can be fixed by reading documentation and understanding what each function you use does!
4. Duplicating data (storing something larger than an int in your deque)
5. Taking advantage of caching
6. Function calls

# Wrapping Up

---

- Thinking about how to optimize your code to provide for many cache hits and to reduce function calls are the LAST things to think about
  - Very rare that these will make a significant difference if you have written efficient code elsewhere
- Prioritize the complexity of your algorithm, and avoiding unnecessary copies and data duplication
- Code that has been written in a well-organized manner can be easily optimized, so organization is quite possibly the best thing you can do to help you write efficient programs!