Programmeren voor multimedia

Theory

Inhoud

1. Introduction: C++

1.1 The Good

- Very **powerful** language
- Many very **advanced language features**
- Very **efficient**
    - You get full control
    - Compilers are very good at optimizing
- Exists on **any platform**
- Many existing **libraries**
- **Used by many** (= lots of helpful resources)
    - Most games are made in C++
    - Most OS's are written in C/C++

1.2 The Bad

- C++ is a very **difficult** language
    - Many rules
    - Complex rules
    - Inconsistent rules
- Those who like C++ generally **don't understand** it (or they are masochists)
- **No safety measures**
- **Mistakes are punished** in the sense that they are not…
    - The app merrily goes on when mistakes occur
    - The error only becomes apparent later
- **Low productivity**: code runs fast but is written very slowly

1.3 The Ugly

- The joys of **undefined behavior**
- **Cryptic compiler errors**
- C++ = **three languages** for the price of one
    - Preprocessor
    - Templates
    - "Regular" code
- **IDE support is difficult**

1.4 C++ and compilers

- **Very complex specification**: current standard counts 1300+ pages
- Compilers have **different interpretations** of this standard
- Different compilers support **different features of different language versions**
- Compilers can introduce **their own features**
- **Compilers are very complex** and sometimes buggy

1.5 C and C++

- **C++'s greatest strength is its compatibility with C**
    - Familiar syntax
    - Existing code can be reused
- **C++'s greatest weakness is its compatibility with C**
    - Inherits C's weaknesses

- Leads to inconsistencies in language

- **Irony**
  - C itself has changed and has introduced incompatibilities

1.6 Java vs. C++

- **Java**
  - Primary design goal: security
  - Ease of use
  - "Programmers are fallible" mentality
- **C++**
  - Primary design goal: efficiency
  - Programmers are gods mentality

1.6.1 Array Indexing

```
int ns[] = {1, 2, 3};
ns[8] = 4;
```

- **Java**
  - No array indexing goes unchecked
  - `ArrayIndexOutOfBounds` thrown in your face
- **C++**
  - No checks whatsoever
  - Undefined behavior
  - Possibly overwrites a random other variable

1.6.2 Null Pointers

```
String s = null;
int x = s.length();
```

- **Java**
  - No member access goes unchecked
  - `NullPointerException` thrown in your face
- **C++**
  - No checks whatsoever
  - Undefined behavior
  - Possibly runs some random code

1.6.3 Protection

```
class Foo
{
    private int x;
}
```

- **Java**
  - x is safely locked away
  - No way to access it in Java
- **C++**

- Very easy to circumvent
- It's more what you'd call a guideline than an actual rule

1.6.4 Memory Safety

```
int* p = 0;
while ( *(++p) = 0 );
```

- **Java**
  - Memory safe
  - Can't peek nor poke around in arbitrary memory
- **C++**
  - Overwrites entire memory space

1.6.5 Garbage Collection

```
new Foo();
```

- **Java**
  - Garbage collector will find and recycle lost objects
- **C++**
  - Lost is lost, no safe way to recuperate it
  - Memory leak
  - You need to free the object's memory manually

1.6.6 Casting

```
Object o;
String s = (Object) o;
```

- **Java**
  - Every object knows its (dynamic) type
  - Casts are checked at runtime
  - `ClassCastException` if wrong cast
- **C++**
  - Many different types of casts
  - Some check, some don't
  - You can force a cast to any type you want

1.6.7 Primitive/Fundamental Types

```
int x = 2000000000;
x *= 10;
```

- **Java**
  - Integers are guaranteed to be two-complement 32-bit
  - Effects of overflow are predictable
- **C++**
  - No specific internal representation guaranteed
  - Undefined behavior

1.7 Conclusion

- C++ makes no compromises: **efficiency** über alles
- Throw all your assumptions overboard
- Take necessary safety precautions
- Why C++?

Ranking	Taal
1	Java
2	C
3	C++
4	C♯
5	Python

TIOBE Index for January 2017

  - Java takes many choices away from you
  - C++ gives you more options
  - C++ allows you to better understand the why behind things
  - Examples
    - **Pass by value/reference**
      - Java always passes arguments by value
      - C++ allows you to choose: pass by value, by pointer or by reference
      - You can give read-only access in C++
    - **Inheritance**
      - Java only supports single inheritance
      - C++ offers multiple inheritance
      - C++ offers public, protected, private inheritance
    - **Overriding**
      - In Java all methods are overridable
      - C++ has “regular” methods and virtual methods
    - **Generics**
      - Java has only class types
      - C++ allows any type to be used
      - C++ supports specialization
      - C++ supports template metaprogramming
    - **Allocation**
      - Three types of allocation
      - Java picks which type is used for what
      - C++ allows you to choose yourself
- **Applications written in C++**
  - All popular browsers
  - MS Office package
  - Windows/Linux GUIs
  - JVM
  - MySQL, MS SQL Server, MongoDB
  - Adobe Creative Cloud applications
  - Most games

2. Declarations and Definitions

2.1 Problems

2.1.1 Fitting everything in RAM

- **Compiling requires a lot of RAM**
- **Old computers had very little RAM** (couldn't even keep one entire source file and corresponding data)
- They wanted a **single pass compilation**
    - Read in a bit of code
    - Check the code for correctness (syntax, types, …)
    - Produce assembly code
    - Forget as much as possible (freeing up RAM)
    - Proceed to next bit of code

2.1.2 Forward references

```
double foo() {
  return bar();
}

double bar() {
  return 0;
}
```

- **Cannot compile** *foo* without knowing *bar*'s type signature
    - *Is it true that bar does not require arguments?*
    - *Is it true that bar returns a double?*
- Definition of *bar* only returns later
- **Solution**: put *bar*'s definition before *foo*'s…

2.1.3 Problem: cyclical references

```
int bar(bool x) {
  return foo(0);
}

int foo(double x) {
  return bar(false);
}
```

- … uh-oh.

2.2 Solution: Forward declarations

2.2.1 Declarations

- Compiler needs to know
    - **arity** (number of parameters)
    - **parameter types**
    - **return type**
- Compiler does not need to know
    - **function body**
- Forward declaration: **provide the compiler with minimal though sufficient information**
- For a function: **name + parameter types + return type**

```
// Forward declarations for both
int foo(double x);
int bar(bool x);

int foo(double x) {
  return bar(false);
}

int bar(bool x) {
  return foo(0);
}
```

2.2.2 Definitions

- A **declaration** provides **partial information**
- A **definition provides all information**
- All that is declared **should be defined** at some point later
- A **declaration is a promise**

2.2.3 Classes

- Rules for classes are **more complex**
- **Multiple possible ways** to declare classes
- Provide **different amount of information**
- Once you define internals of a class, you must **list them all**
- I.e. you are not allowed to add fields/methods afterwards
- Method bodies can (often) be deferred
- Examples:

```
// Declaration
class Counter {
private:
  int x;

public:
  Counter();
  void increment();
  int read();
};

// Member function definitions
Counter::Counter() : x(0) { }
void Counter::increment() { x++; }
int Counter::read() { return x; }
```

```
// Full definition
class Counter {
private:
  int x;

public:
  Counter() : x(0) { }
  void increment() { x++; }
  int read() { return x; }
};
```

```
// Mixed declarations and definitions
class Counter {
private:
  int x;

public:
  Counter() : x(0) { }
  void increment() { x++; }
  int read();
};

int Counter::read()
{
  return x;
}
```

```
// Mixed declarations and definitions
class Counter {
private:
  int x;

public:
  Counter() : x(0) { }
  void increment() { x++; }
};

// WRONG!
// Class declaration does not mention read
int Counter::read()
{
  return x;
}
```

- **Exceptions:**
  - Member functions can refer to other member functions **from the same class** that have not been declared yet

```
class Foo {
public:
  void foo() {
    bar(); // compiles just fine
  }

  void bar();
};
```

  - Some weird rules
    - Code below fails because compiler expects 0 parameters
    - Solution: use **::foo(5)**

```
void foo(int x);

class Bar {
public:
    void bar() {
        foo(5); // fails
    }

    void foo();
};
```

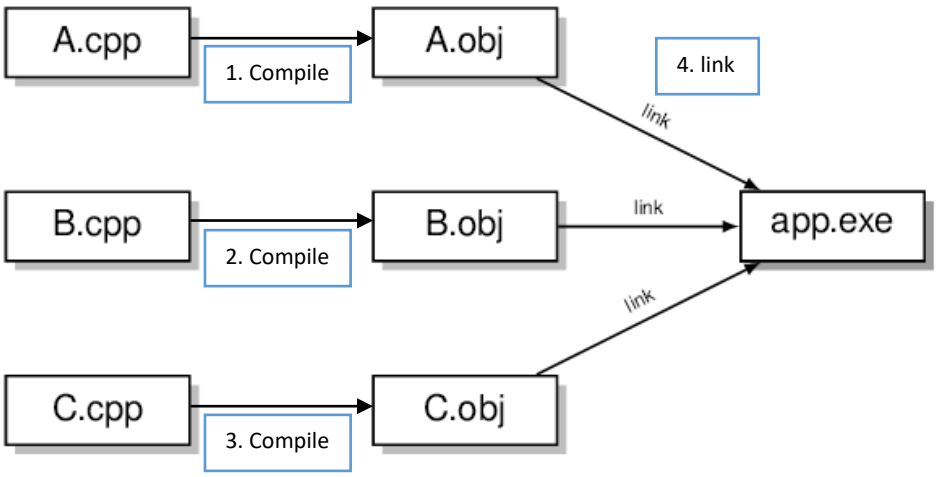
2.3 Summary

- Code gets **compiled in single pass**
- Compiler needs to be able to verify your code and generate correct results **without having to look ahead**
- **Partial information** (i.e. declarations) **will often do**
- **Examples**
    - Calling a function: requires signature to be known
    - Manipulating a variable of Type T: requires class T
    - Calling member functions on object of class T: requires declarations of **all** members of T
    - Instantiating a class T: requires declarations of all members of T

3. Preprocessor

3.1 The linker

- **Problem**
  - Large programs have large number of functions/classes
  - Compiler can't fit all this data in memory
- **Solution**
  - Divide codebase in small units: compilation units (CUs) → .cpp files
  - In each file we put **the minimal amount of declarations**
  - **Compile** each unit **separately**
  - **Link** results together
  - **Scheme:**



  - **Example:**

foo.cpp

```
double bar(int);
bool foo() { return bar(1)>0; }
```

bar.cpp

```
int qux(int);
double bar(int x) { return qux(x) / 2.0; }
```

qux.cpp

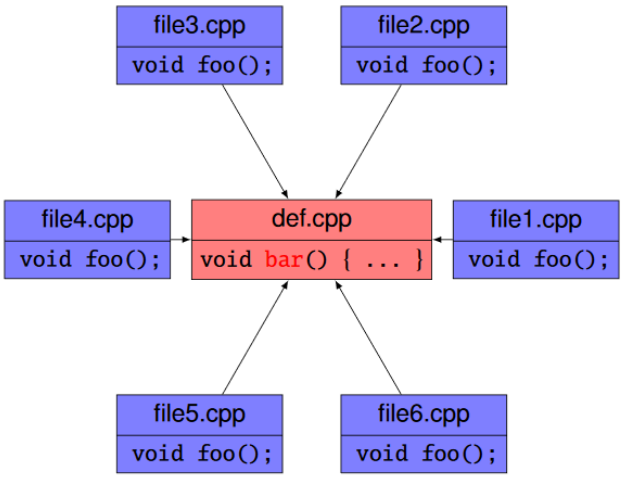
```
int qux(int x) { return x * x; }
```

  - Each file can be compiled separately
  - Each file contains minimal but sufficient declarations
- **The Linker**
  - The compiler will believe you promises (declarations)
  - If you do not define your declared function, compiler will **assume it's in other CU**
  - **Linker** is less naïve: will **check for promises**
  - **Missing definition** will lead to **linker error**

3.2 Header files

- **Example engineering nightmare**
  - Say the **definition of a function changes** (name, parameter or return type change)

- All CU's containing a declaration need to be **updated**
- Due to redundancy: same declaration in **many files**



- **Solution**
  - Make cpp file responsible for **generating a list of declarations** for all of its definitions
  - Put these declarations in **separate file**
  - **Include** the contents of this twin file in other cpp files
  - This twin file is called a **header file**
- **Example:**

funcs.cpp

```
void foo(int x) {
  ...
}

bool bar() {
  ...
}

char qux(double x) {
  ...
}
```

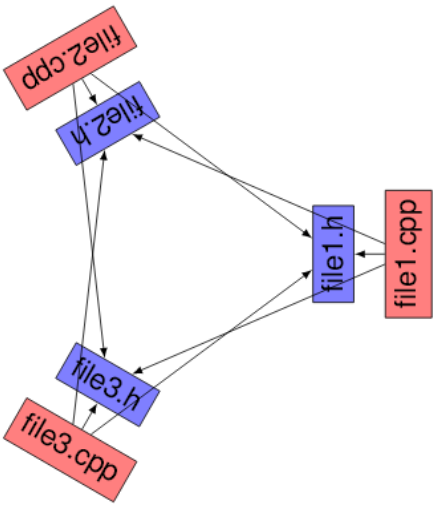
funcs.h

```
void foo(int);
bool bar();
char qux(double);
```

main.cpp

```
#include "funcs.h"

int main()
{
  foo(5);
  bar();
  qux(8.8);

  return 0;
}
```

- **Scheme:**



- **Typical cpp fille structure**
  - First include: preferably own header file

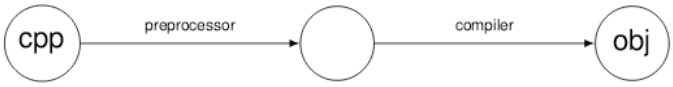
file.cpp

```
#include "file.h"
#include "dependency1.h"
#include "dependency2.h"
#include "dependency3.h"

// Definitions for functions
// and classes declared in file.h
```

3.3 Preprocessor

3.3.1 Directives

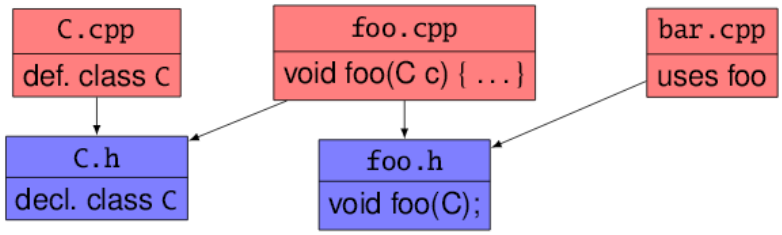
- Preprocessor is a separate mini-language
- All code gets processed by the preprocessor first



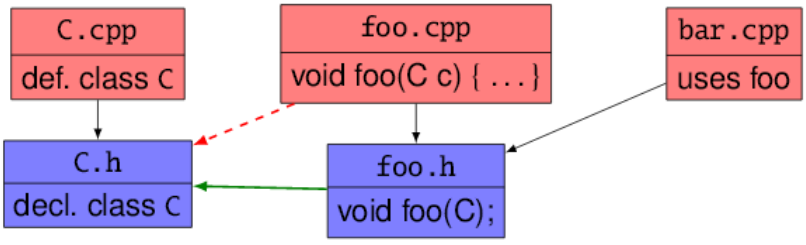
- #include "file" is a **preprocessor directive**
  - This is replaced with the contents of the file
  - Works with any file on any line in the cpp file
  - Preprocessor doesn't care about contents of files

3.3.2 Headers with dependencies

- **Problem: bar.cpp** doesn't know type C



- **Solution:** have foo.h include C



- **Rules of thumb:**
  - X.cpp includes its own header file X.h first
  - X.h includes minimal but sufficient selection of header files
    - *Litmus test: a cpp with only #include "X.h" must compile*
  - Let X.cpp include all extra header files it needs

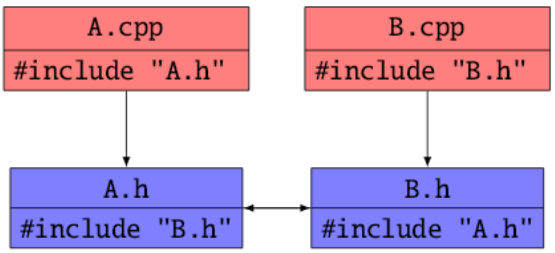
3.3.3 Standard library

- Standard C++ library accessible through header files
- Including using < > instead of " "
- < > looks for header files in different place
- C libraries: header files end in .h

- C++ libraries: no extension
- Examples
  - Strings: #include <string>
  - Vector: #include <vector>
  - IO streams: #include <iostream>
- Best to include these last

3.4 Preprocessor macro's

- **Problem: multiply included header files**
  - **Infinite loop**
  - It's **easy** for situations to occur where the same header files get included (typically with headers from the standard C++ library)
  - Is this bad? **Depends** on contents of header file
  - Multiple agreeing declarations are allowed, but there are **exceptions**



- **Solution:** preprocessor macro's

3.4.1 Macro's to define dependencies

- A macro is like a variable that can either be defined or undefined
  - `#define IDENTIFIER`
  - `#undef IDENTIFIER`
  - `#ifdef IDENTIFIER ... #endif`
  - `#ifndef IDENTIFIER ... #endif`
  - `#ifdef IDENTIFIER ... #else ... #endif`
  - `#ifndef IDENTIFIER ... #else ... #endif`
- **Example**:

Before preprocessing

X.h

```
#ifndef X_H
#define X_H
void foo();
#endif
```

X.cpp

```
#include "X.h"
#include "X.h"
#include "X.h"
```

After preprocessing
(what the compiler sees)

```
void foo();
```

3.4.2 Macro's to define constants

Before preprocessing

```
#define INIT 1000

void foo()
{
  int n = INIT;

  // ...
}
```

After preprocessing (what the compiler sees)

```
void foo()
{
  int n = 1000;

  // ...
}
```

3.4.3 Macro's to define functions

Before preprocessing

```
#define NULLCHECK(p) if ( p == nullptr ) \
                       fail();

void foo(int* ptr)
{
  NULLCHECK(ptr);

  /* ... */
}
```

After preprocessing

```
void foo(int* ptr)
{
  if ( ptr == nullptr )
    fail();

  /* ... */
}
```

3.4.4 Predefined macro's

- __FILE__: containing file
- __LINE__: line number on which this macro appears
- __DATE__: date on which the preprocessor is being run
- __TIME__: time at which the preprocessor is being run
- **Example:**

Before preprocessing

```
#define NULLCHECK(p) if ( p == nullptr ) \
    fail(#p " is null", __FILE__, __LINE__ );

void foo(int* ptr) {
  NULLCHECK(ptr);

  /* ... */
}
```

After preprocessing

```
void foo(int* ptr) {
  if ( ptr == nullptr )
    fail("ptr␣is␣null", "X.cpp", 5);

  /* ... */
}
```

3.4.5 Dangers

- **Example:**

Before preprocessing

```
#define MIN(a, b)  (a < b ? a : b)

int x = 3;
int y = MIN(++x, 7);

// x == ?, y == ?
```

After preprocessing

```
int x = 3;
int y = (++x < 7 ? ++x : y);

// x == 5, y == 5
```

3.4.6 Debug and release builds:

- *assert* macro to **check assumptions**

```
#ifdef DEBUGGING
#define assert(c) \
  if ( !c ) \
    abort( #c "␣failed", __LINE__, __FILE__ )
#else
#define assert(c)
#endif
```

- While debugging: **define DEBUGGING**
- For performance: don't define DEBUGGING
- Easy way to **turn on/off diagnostic code**
- Debug and Release builds generally known by IDE
- Release build: NDEBUG is defined
- **assert is already defined** for you
    - # include <assert.h>

4. Allocation methods

- Different ways to deal with RAM
- **Trade flexibility for speed** (flexible but slow vs. rigid but fast)
- **In most languages abstracted away** as much as possible
- In **C++**, it's **crucial to know about the details**

4.1 Static allocation

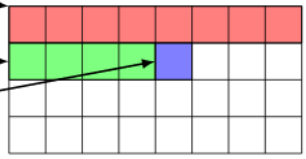
4.1.1 What

- **Simplest** allocation mechanism
- **Fastest**
- **Most restrictive**
- Used in **oldest** programming language (Fortran, early 1950s)
- **Warning:** following example are what-ifs, they do not show how C++ really works

4.1.2 How

- Compiler scans source code for variables (parameters, locals, globals…)
- The compiler assigns a fixed address to each variable

```
double global;

void foo(int param)
{
  char local;

  /* ... */
}
```



4.1.3 Consequences

- Compiler needs to have full knowledge at compile-time

```
// Size known at compile-time: ok
int ns[5];

// Size computed at runtime: not ok
int size = foo();
int* ns = new int[size];
```

- Local variables "remember" their values

```
int counter()
{
  int current = 0;

  return ++current;
}

counter(); // 1
counter(); // 2
counter(); // 3
```

- Recursion not possible

```
// Computes sum 1...n
int sum(int n)
{
  if ( n == 0 ) return 0;
  else return n + sum(n - 1);
}

sum(5); // returns 0
```

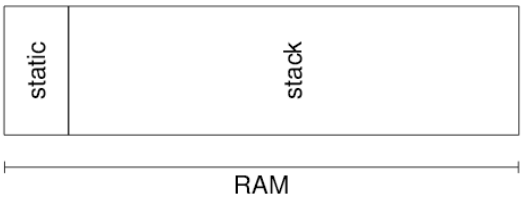
4.1.3 Pro's & con's

Pro's	Con's
• Very fast • No bookkeeping at runtime • No out-of-memory errors possible at runtime • Entire memory footprint is known beforehand	• Data structure size must be known at compile time • Full recompilation needed to allow larger data structures • Cannot make use of extra RAM: if it's been compiled to work with 1MB but you have 8GB, too bad • No recursion possible

4.2 Stack allocation

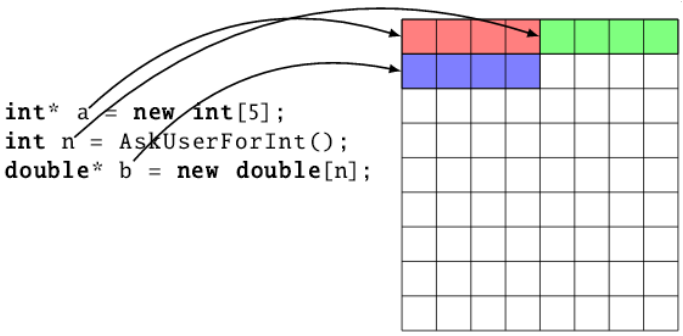
4.2.1 What

- Allocate your **variable using static allocation**
- Introduce the **`stacknew` keyword** that allows for **runtime allocation**
- Take the **remainder of RAM** (after static allocation) and act as if it were **a stack (LIFO) of available bytes**
- Allocation = take top *N* bytes

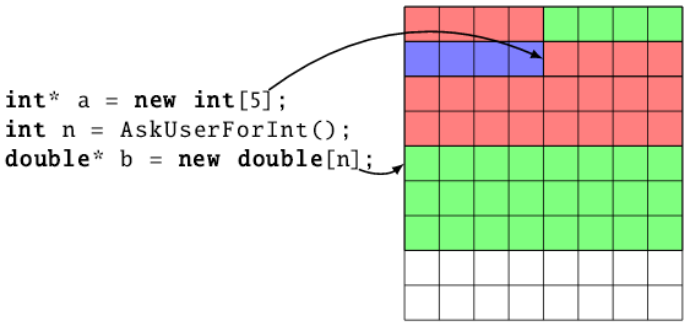


4.2.2 How

- **At compile time**: compiler assigns memory locations to globals (static allocation)



- **At runtime**: extra memory is allocated on stack at runtime



- Extra example: slides 16 – 33

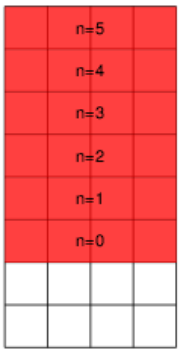
4.2.3 Consequences

- You can **keep allocating** but eventually you'll run out of memory (**stack overflow**)
- You need free memory: can only **deallocate what's on top of stack**

```
int* a = stacknew int[4];
double* b = stacknew double[3];
bool* c = stacknew bool[4];
char* d = stacknew char[12];

// Cannot be freed in different order
free d;
free c;
free b;
free a;
```

- **Recursion possible**

```
int sum(int n) {
  return n == 0 ? 0 : n + sum(n - 1);
}

sum(5);
```



4.2.4 Pro's & con's

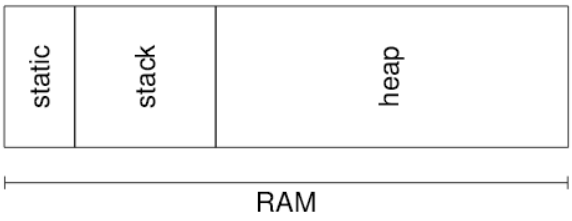
Pro's	Con's
• Fast allocation • Little runtime bookkeeping needed • Amount of memory to be allocated does not have to be known at compile time • Can allocate any amount of memory you want at runtime • Recursion made possible	• Problematic deallocation due to rigid order

4.3 Heap allocation

4.3.1 What

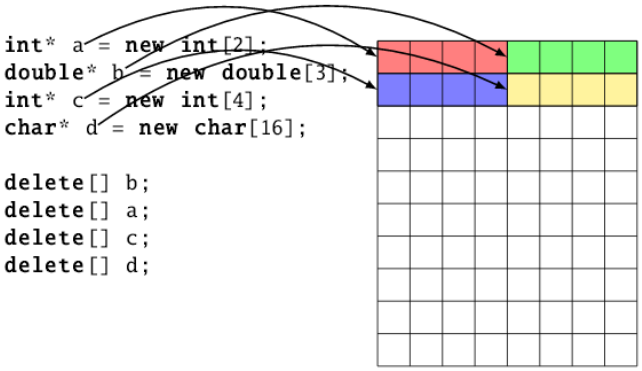
- Step 1: use **static allocation where possible**

- Step 2: **pick part of RAM** and declare it **stack**
- Step 3: **remainder** of RAM is **heap**
- Introduce keyword **new** as means for heap allocation

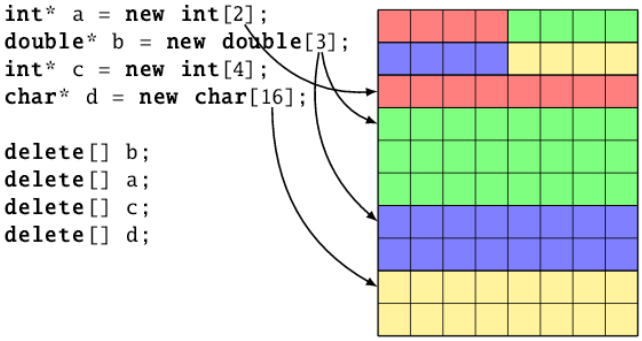


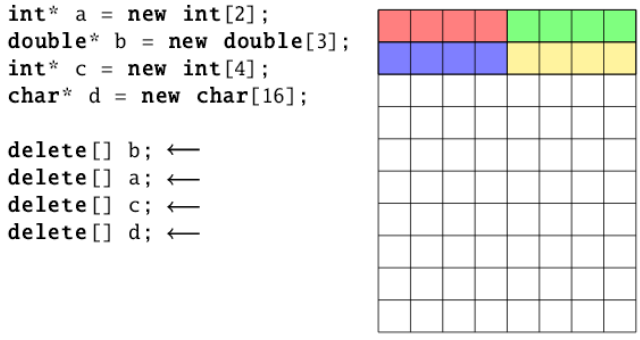
4.3.2 How

- **At compile time**: compiler assigns memory location to each global (static allocation)



- **At runtime:** arrays allocated on heap



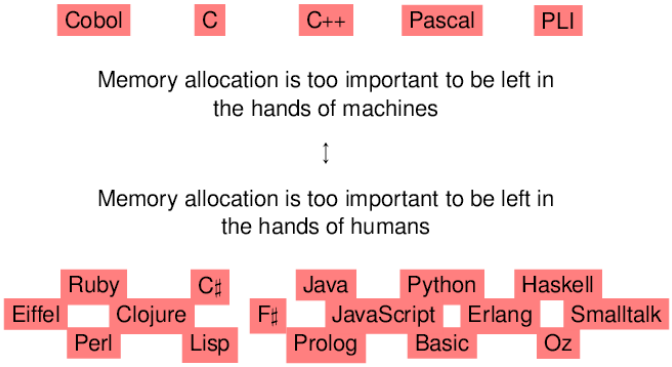


4.3.3 Pro's & con's

Pro's	Con's
• Allows allocation of arbitrarily large blocks • Deallocation possible in any order • Most general allocation method	• Requires some bookkeeping (keeps track of linked list of free blocks) • Deallocation necessary • Allocation/deallocation relatively slow • Fragmentation (slides 64 – 88)

4.4 Garbage collection

- **Abstraction of memory**
- Tries to **make it look as if** there's an **infinite amount of RAM**
- **Recycles unreachable memory**
- **Reduces memory leaks** (but does not eliminate them)



4.5 Java vs. C++

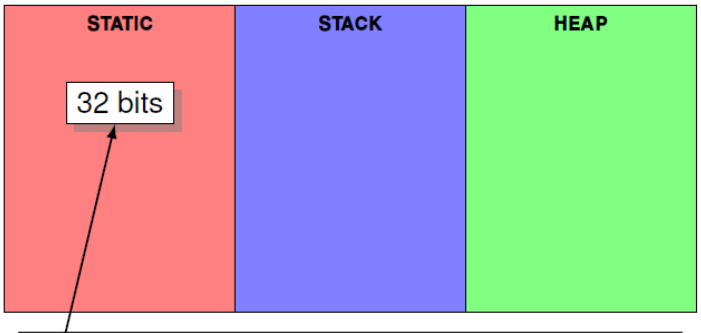
Allocation in Java

Static	Stack	Heap
• Class variables (static fields) (only primitives)	• Arguments (parameter values) • Locals (local variables) (only primitives)	• Object variables

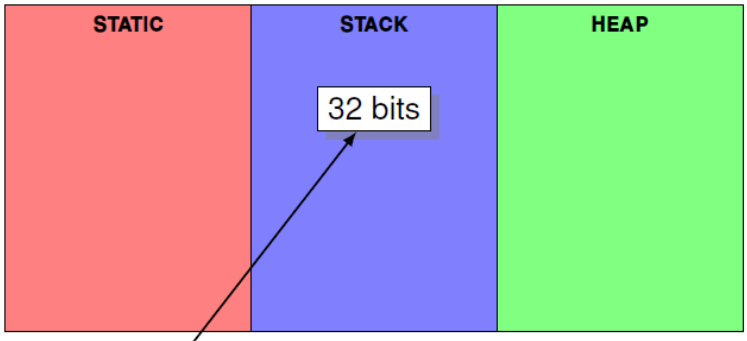
Allocation in C++

Static	Stack	Heap
• Globals • Class variables • Static locals	• Arguments • Locals	• Anything created using *new*

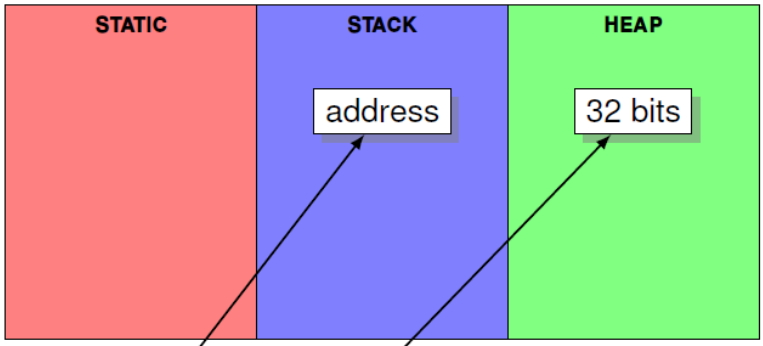
- C++ is much **more flexible**
- **Anything can be placed anywhere**




```
int x;

int main()
{
  x = 5;
}
```




```
int main()
{
  int x;

  x = 5;
}
```




```
int main()
{
  int* p = new int;

  *p = 5;
}
```

- Same applies on **all types:** same treatment for both primitives and objects
- E.g. objects can be put on the stack:

```
int main()
{
  // Allocates object on stack
  Person p;

  // p is ready for use
  int n = p.getAge();
}
```

5. Pointers

5.1 Argument Passing in Java

- **Primitive** types are passed by **value**
- **Objects** are passed by **reference**
- Why the inconsistency?
  - **Efficiency**
    - Passing **large object** by value uses a lot of time and memory
    - **References** have **constant size** (32 or 64 bit) and thus fit in a register
    - Give access **to arbitrarily large data structures** using a **single reference**
  - **Sharing**
    - References allow you to "**share**" objects
    - If objects were always passed by value, you could not pass the same object to two (or more) different objects
  - **Recursion and `null`**
    - Without references, **no recursive data structures** are possible
    - Example

```
class Person {
    private Person mother;
    private Person father;
}
```

      - `Person` would be **infinitely large**
      - Thanks to references, a `Person` Is only 64/128 bits large
      - **`null`** can be used to **end chains**
  - **Giving access**
    - Without references, functions would never be able to modify their arguments
    - Without references, **`this`** would be a copy
    - Methods would never be able to modify the object's fields

```
class Counter {
  private int x;

  public void inc() {
    this.x++;
  }
}
```

5.2 Pointers in C++

- C++ provides **pointers**
- Pointers fulfill the same role as references do in Java
- C++ does not treat primitives and classes differently
  - Java does not allow reference to primitive
  - C++ allows pointer to primitive
- **Syntax**
  - Value: `T x`
  - Pointer: `T* x`
- Pointers have a reputation
- Pointers are *not hard* (if you can work with arrays in Java, you can work with pointers in C++)

- Pointers are **fragile**
- **Easy to make mistakes**, but that's ok, because they're really simple

5.3 Memory: A Simple Model

- Memory can be seen as one **gigantic `byte[]` array**
- Every time you need memory, a little part of this array is given to you
- You could assign ranges to the different allocation methods, e.g.
  - Static allocation: first 1.000.000 bytes
  - Stack: following 1.000.000 bytes
  - Heap: the rest
  - but this distinction is not important right now

- If you need an `int x` variable, four consecutive bytes will be required to store `x`
- Since memory is a `byte[]`, every byte has its unique index
- We can write this index in hexadecimal
- In the above figure: `x` occupies bytes with indices `0x1256, 0x1257, 0x1258, 0x1259`
- The index of the first byte of `x` is its address; `0x1256`

5.4 A New Type: Pointer

- **Why**
  - You can ask for an address of a variable
  - If you know where a variable resides in memory, you can read and write to it
  - Example: division (full example slides 20 – 25)

```
// Not real C++ syntax
void div(int x,
         int y,
         address_of_int q,
         address_of_int r) {
  write_int_to_address(q, ...);
  write_int_to_address(r, ...);
}

void func() {
  int q, r;
  div(5, 3, address_of(q), address_of(r));
  // q and r now contain quotient and rest
}
```

- **Pass-by-address**
  - Passing q and r as regular ints **(pass-by-value) does not work**: copies are given to div and no matter what div does, it has no effect on func's q and r
  - Passing the **addresses** of q and r allows div to **access func's local variables**
  - This technique is called **out parameters**: using parameters as output instead of input
- **Syntax**
  - **A pointer is a variable that contains an address**
  - To access whatever is located at the address, you *dereference* the pointer
  - The `&` and `*` need to be balanced
  - Similar to arrays: given a `T[] xs` you need to index `xs` to reach a `T`

Syntax	Description
*type**	Pointer to a *type*
&*variable*	Address of *variable*
**pointer*	Dereference pointer

- Pseudosyntax

```
int x;

address_of_int address = address_of(x);
write_to(address, 5);
```

- C++

```
int x;

int* address = &x;
*address = 5;
```

- Example: slides 32 - 42

5.5 Dangers of Pointers

- Pointers can refer to arbitrary memory
- Reading/writing these pointers is dangerous → can overwrite other variables/objects/
- Can lead to randomly occurring crashes
- Example

```
int* p = 5;
*p = 1;
```

  - Initialize pointer with 5
  - `*p = 1` writes 1 to memory location with address 5
- **Dangling pointers**
  - Dangling pointer = pointer **pointing to free/arbitrary memory**
  - Using a dangling pointer leads to **undefined behavior**
    - Program might still seem to function properly for a while
    - Can lead to randomly occurring bugs later on
  - Multiple ways of ending up with dangling pointers
    - E.g. returning a pointer to a local, see slides 48 - 56
    - It's important to know what gets deallocated and when
  - This kind of **manual memory management is a major source of bugs**
- **Summary**

Syntax	Description
*type**	Pointer to a *type*
&*variable*	Address of *variable*
**pointer*	Dereference pointer

  - **Use pointers when**
    - You need to pass large objects
    - When you want to give write-access to functions
    - When you need nontrivial data structures
  - **Remember**
    - In Java, references are used all the time (except for primitives)
    - There's no use trying to avoid pointers in C++
    - They are a necessary evil

6. Heap Allocation

- **Slow** but **flexible**
- Using keyword **new**
  - **new** *type*
  - Returns a **pointer** to *type*
  - Memory contains random bits
- **Initialization**
  - Constructor-style syntax
  - Also works on primitive types
  - Example:

```
double* px = new double(3.1415);
```

- **Freeing memory**
  - **delete** *id*
  - Deallocating memory necessary
  - Illegal to use pointer after `delete`: dangling pointer
  - Example:

```
bool* p = new bool;

delete p;

// Forbidden
std::cout << *p << std::endl;
```

- **Memory leaks**

```
new int;

// We never stored pointer
// We cannot free the memory
```

  - Don't forget to delete
  - Leads to memory leaks
  - No way to find out if a pointer has already been deleted
  - Design often affected by manual memory management
  - Need for many helper classes to deal with complexity

7. Arrays

- Syntactically **resemble Java arrays**
- Arrays in Java are evil (too inflexible)
- Arrays in C++ are **particularly evil**
  - Weird, complicated rules
  - Shortcuts that complicate thing even more
- We focus on *heap allocated arrays*
- Couple of white lies for the sake of simplicity

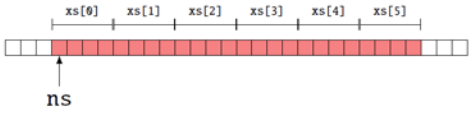
7.1 The Sane Part

- **What's an array?**

```
new int[6]
```

  - An array of Ts is a series of consecutive Ts in memory
  - Indexing is zero based
- **Type of arrays**

```
int* ns = new int[6]
```



  - An array of Ts is written T*
  - Is in fact a pointer pointing to first element
- **Size of arrays**
  - You **can't ask for an array's** size
    - You only know where the array starts
    - But you don't get to know how many elements there are
  - If you need size, **you'll have to keep track of it yourself**
- **Example of security vulnerability**

```
char* buffer = new char[10];

// Read string from console
std::cin >> buffer;
```

  - Maximum input length allowed: 9
    - If more, random memory gets overwritten
    - Buffer overflow
    - Very dangerous
  - Why 9, not 10?
    - Strings must end with `0`
    - Only true for strings!
  - Never use `char*` in C++, but `std::string`
- **Freeing an array**

```
int* p = new int[10];
// ...
delete[] p;
```

- `delete p` only frees first element → memory leak
- **`delete[] p`** is necessary for arrays
  - `delete[]` somehow knows the array's size
  - But how it knows, is a secret!
  - Must be stored somewhere…
  - Where is implementation dependent

- **Indexing**

```
int* p = new int;
int* q = new int[4];
```

  - To access array elements, use `[]`
  - **Indexing is allowed on all pointers**
    - Indexing on non-arrays
      - No difference between `*p` and `p[0]`, both ok
      - `p[1]` reads beyond element → undefined behavior
    - Indexing on arrays
      - Also no difference between `*p` and `p[0]`
      - `p[1]` gives second element
  - Little difference between `new int` and `new int[1]`
  - There's no separate array type in C++
  - No checks done
    - Negative indices allowed → undefined behavior
    - Indexing beyond bounds allowed → undefined behavior

- **Summary**

```
// Creation
T* ts = new T[n];

// Reading
T x = ts[i];

// Writing
ts[j] = x;

// Freeing memory
delete[] ts;
```

7.2 The Insane Part

- **Stack allocation**
  - You can allocate arrays on stack
  - **Size**
    - Must be known at compile time

```
void foo(int size)
{
    int ns[5];     // ok
    int ms[size]; // Not ok
}
```

    - You can ask for the size of a stack allocated array
    - But only in the same function that created it

```
// 32 bit platform
void foo(int ns[])
{
  std::cout << sizeof(ns) << std::endl; // 4
}

void bar()
{
  int ns[5];
  std::cout << sizeof(ns) << std::endl; // 20
  foo(ns);
}
```

- Reason behind size discrepancy
  - `bar`
    - `bar` prints `20`
    - `ns` has type `int[5]`
    - Compiler can compute `5 x 4 = 20`
  - `foo`
    - You cannot pass array types to functions
    - Type internally degrades to `int*` when calling `foo*`
    - `int*`, `int[]` or `int[5]` are all identical parameter types
    - `foo` prints 4 because that is size of pointer

7.3 What You Need To Know

- Only heap allocation
- Use `T*` as sole array type
- Assume size is always unknown

8. Containers

8.1 std::vector

- Equivalent of Java's `ArrayList`
- `#include <vector>`
- Is a template class (similar to Java generics)
- **Creation**

```
// On stack (no () allowed!)
std::vector<int> ns;

// On heap
std::vector<int>* ns = new std:vector<int>();

// On heap, shorter notation
auto ns = new std:vector<int>;
```

- **Indexing**

```
std::vector<int> ns;

int x = ns[index];
```

- **Setting**

```
std::vector<int> ns;

ns[index] = n;
```

- **Size**

```
std::vector<int> ns;

int size = ns.size();
```

- **Iterating**

```
std::vector<int> ns;

for ( int n : ns )
{
    ...
}
```

- **Other funcitonality**

Member function	Effect
`v.front()`	First element
`v.back()`	Last element
`v.data()`	Returns underlying array (fragile!)
`v.push_back(x)`	Add to end
`v.pop_back()`	Remove last
`v.clear()`	Remove all

8.2 Other Containers

- **List**
  - Doubly linked list
  - `#include <list>`

Member function	Effect
`lst.front()`	First element
`lst.back()`	Last element
`lst.push_front(x)`	Add to beginning
`lst.pop_front()`	Remove first
`lst.pop_back()`	Remove last
`lst.push_back(x)`	Add to end
`lst.pop_back()`	Remove last
`lst.clear()`	Remove all

- **Set**
  - Uses binary search tree (i.e. no hashing)
  - `#include <set>`
  - **Cannot** contain same element more than once
- **Multiset**
  - Uses binary search tree
  - `#include <set>` *(not multiset)*
  - **Can** contain same element more than once
- **Map**
  - Uses binary search tree
  - `#include <map>`
  - Associates **one value** with one **key**
- **Multimap**
  - Uses binary search tree
  - `#include <map>` *(not multimap)*
  - Associates **multiple values** with one **key**
- **Strings**
  - Type of strings: `std::string`
  - Strings also act as containers
  - `#include <string>`

8.3 Iterators

- Is an object that **points to element of container**
- It can be used to **read** and **write** to that location
  - `x` = *it reads element at that position
  - `*it = x` overwrites element at that position
- Can be **moved around in container**
  - `it++` moves to next element
  - `it--` moves to previous element
- **Why iterators?**
  - Iterators **abstract away the container specifics**
  - You can use vector, list, set,… iterators in the same way: iterators are smart enough to know how to deal with the underlying container
- **How to get an iterator**
  - Container classes offer `begin()` and `end()`
    - `begin()` gives iterator pointing to **first element**
    - `end()` gives iterator pointing **one past last element**

- **Example: zeroing elements**
  - You want to write `zero(container<int>*)`
  - Overwrites each integer in container with `0`
  - You don't write once for each container
    - `zero(std::vector<int>*)`
    - `zero(std::list<int>*)`
    - `zero(std::set<int>*)`
    - …

```
template<typename Iterator>
void zero(Iterator start, Iterator end)
{
    while ( start != end )
    {
        *start = 0;
        ++start;
    }
}
```

  - `template` is C++'s version of generics, but more flexible
  - Basically says: give me start and end of any type that
    - can be compared using `!=`
    - can be dereferenced using *
    - can be incremented with ++
  - **Why not inheritance?**
    - Same could be achieved using **subtyping**
      - Iterator superclass
      - Each container defines own subclass
    - Using templates is **more efficient**
      - More information at compile time
      - Compiler can optimize better
- **Algorithms**
  - Standard library provides functions on iterators
  - `#include <algorithm>`

Function	Effect
all_of	Checks if all elements satisfy condition
any_of	Checks if any element satisfies condition
none_of	Checks if no element satisfies condition
find_if	Finds first element that satisfies condition
find_if_not	Finds first element that does not satisfy condition
count_if	Counts elements that satisfy condition
count_if_not	Counts elements that do not satisfy condition

  - Example:

```
#include <algorithm>


bool is_prime(int x) { ... }
std::vector<int> vec;

// Are all elements primes?
if (std::all_of(vec.begin(),
                vec.end(),
                is_prime))
{
    ...
}
```

- **Reverse iterators**
  - Useful for **iterating backwards**
  - `xs.rbegin()` and `xs.rend()` to get them
  - `it.base()` to turn it back into a forward iterator

```
// Print each x in xs in reverse order
for ( auto it = xs.rbegin();
      it != xs.rend();
      ++it )
{
    std::cout << *it << std::endl;
}
```

9. Const

9.1 What

- Adding `const` to type gives a **read-only version**
  - **Read access**: allow no changes to be made
  - More efficient when passing large objects that should not be modified
  - Instead of passing by value, which is essentially copying the entire object
- `const T` can be seen as a **supertype** of T
  - You can give a T to someone asking for a `const T`
  - You cannot give a `const T` to someone asking for a T
- **What to use when**
  - **Pass by value** – read only access for small values
  - **Pass by pointer** – read and write access
  - **Pass by `const` pointer** – read only access for large values

	Read	Read+Write
Small	`T`	`T*`
Large	`const T*`	`T*`

- Example:

```
// Only needs read access to work
int sum(const std::vector<int>* ns)
{
    int result = 0;

    for ( int n : *ns )
        result += n;

    return result;
}
```

9.2 Nuances

- **`const std::vector`**

```
const vector<Person*> people;

people[0].set_age(30); // Error!
```

  - `const std::vector` works on two levels
    - the vector itself cannot be modified
    - the items inside the vector cannot be modified
  - You **cannot** have an immutable vector with mutable items
- **Constant pointers vs. constant pointees**
  - `const int* x`
    - Pointer to a constant int
    - The pointer itself is not constant

```
int x = 1, y = 2;
const int* p = &x;

*p = 5;    // Forbidden
p = &y;    // Allowed
```

- `int* const x`
  - Constant pointer to an int
  - The pointer itself is constant, the int isn't

```
int x = 1, y = 2;
int* const p = &x;

*p = 5;    // Allowed
p = &y;    // Forbidden
```

- `const int* const x`
  - Pointer cannot be redirected
  - Pointee cannot bee modified

```
int x = 1, y = 2;
const int* const p = &x;

*p = 5;     // Forbidden
p = &y;     // Forbidden
```

10. References

10.1 Syntax

- **Pointers**
  - Efficient way to give access to large amounts of data
  - Allow to give write-access to own variables
  - Clumsy syntax
- **References**
  - C only has pointers
  - **C++ introduced references**
  - Gives **same "power"** as pointers
  - Uses **pointers behind the scenes**
  - Much **cleaner syntax**
- **Syntax**

Syntax	Semantic
`T[]`	Array of `T`s
`const T`	Readonly view of `T`
`T*`	Pointer to `T`
`T&`	Reference to `T`

  - *Warning: do not confuse with address-of operator*

10.2 Usage

10.2.1 Pass by reference

```
void foo(int& arg);

int x = 5;
foo(x);
```

- `foo` receives a reference to `x`
- Gives same access as when using pointer
  - `foo` can read `x`
  - `foo` can modify `x`
- **Cleaner syntax**, no need for dereferencing
- Example:

```
void foo(int& x) {
    // Reading x
    std::cout << x;

    // Writing x
    x++;
}

int a = 1;
foo(a);       // Prints 1
// a == 2
```

10.2.2 Aliasing

- Introducing another name for a variable

```
int* stupidly_long_named_pointer;
int& x = *stupidly_long_named_pointer;

// x is shorthand for
// *stupidly_long_named_pointer
```

10.2.3 Returning references

```
int& max(int& a, int& b)
{
    if ( a >= b ) return a;
    else return b;
}

int a = 1, b = 2;
max(a, b)++;
// a == 1, b == 3
```

- Intuition: `max` returns the variable itself, not just its value
- Another example:

```
class Person
{
    private int _age;

    int& age()
    {
        return _age;
    }
};

Person p;
p.age() = 5;
```

10.3 Guidelines

- **Limitations of references**
  - References **cannot be retargeted**: once a reference refers to `x`, it does so forever
  - References **cannot be null**
- **When to use references**
  - **Rule of thumb**: prefer references over pointers
    - Unfortunately, pointers will often be unavoidable
  - **In practice**: mostly used with functions
    - `const T&` parameters
    - `T&` return values
  - **We introduce a convention**
    - Writing `foo(x)` should mean `x` remains unmodified
      → `foo` receives `T` or `const T&`
    - Writing `foo(&x)` should mean `x` could change
      → `foo` receives `T*`
    - The `&` makes it explicit that `x` could change
      → no need to look up `foo`'s signature

- **How to pass parameters:**
  - Read only access
    - Small values: pass by value
    - Large values: pass by `const` reference
  - Read and write access
    - Always by pointer

10.4 Summary

Syntax	Semantics
T	Plain old T
T*	Pointer to T
T&	Reference to T

Syntax	Semantics
x	Value of x
&x	Address of x
*x	Dereference x

```
void by_value(int x);
void by_pointer(int* x);
void by_reference(int& x);

// Calling syntax
int x;

by_value(x);
by_pointer(&x);
by_reference(x);
```

11. Classes

11.1 Basic syntax

- Members with **same access level** are **grouped**
- You can alternate as many times as you want
- Class **declaration ends with ;**
- Example:

```
// Declaration of Person
class Person
{
private:
  std::string name;
  int age;

public:
  Person(const std::string&, int);
};
```

- **Access modifiers**
  - `public` – accessible to everyone
  - `protected` – accessible to subclasses
  - `private` – accessible within class
  - Default access modifier: **private**

```
class Foo
{
    int x;
};
```

→ x is private

- **Member functions inside class declaration**
  - Preferably only for single statement methods

```
class Person
{
  int age;

public:
  int getAge() { return age; }
};
```

- **Member functions outside class declaration**

```
class Person
{
  int age;

public:
  int getAge();
};

int Person::getAge()
{
  return age;
}
```

- Indicates the member function's class
- Without it, C++ will think `getAge` is global function

- **Typical file layout**

Foo.h

```
#ifndef FOO_H
#define FOO_H

class Foo
{
  int _x;

public:
  int& x();
};

#endif
```

Foo.cpp

```
#include "Foo.h"

int& Foo::x() {
  return _x;
}
```

11.2 Constructors

- **Called upon object creation**
  - Which constructor depends on context
  - When explicitly creating object: same rules as Java
  - There are other situations where objects are created (see Constructors)
- **Constructor initializes all member variables**
  - Different rules than Java!

11.2.1 Initializer List

- **Initializes the classes' fields**
- Gets executed first
- Example:

```
class Foo {
    int a, b, c;

public:
    Foo() : a(1), b(2), c(3) { }
};
```

- Can use **constructor calls**:

```
class Foo {
public:
    Foo(int x) { }
};

class Bar {
    Foo foo1, foo2;

public:
    Bar() : foo1(5), foo2(6) { }
}
```

- **Omitting the initialization** of a field when there is no default constructor results in an error:

```
class Foo {
public:
    Foo(int x) { }
};

class Bar {
    Foo foo1, foo2;

public:
    Bar() : foo1(5) { } // Error
};
```

- C++ knows whether you are referring to a field or a parameter:

```
class Foo {
    int a, b, c;

public:
    Foo(int a, int b, int c)
        : a(a), b(b), c(c)
    {
    }
};
```

- Default constructors of primitive types do nothing: the fields will have random values:

```
class Foo {
    int x, y, z;

public:
    // No explicit init for x y z
    Foo() { }
}
```

11.2.2 Constructor Body

- After initializer list, body gets executed
- **Initialize as much as possible in the initializer list**
    - Otherwise field will get initialized twice
    - Example: `foo` gets initialized twice

```
class Foo {
public:
    Foo()    { }
    Foo(int) { }
};

class Bar {
    Foo foo;

public:
    Bar() {
        foo = Foo(5);
    }
};
```

11.2.3 Delegation

- Delegating to another constructor:

```
class OilTank {
  int contents, capacity;

public:
  OilTank()
    : OilTank(0, 1000) { }

  OilTank(int capacity)
    : OilTank(0, capacity) { }

  OilTank(int contents, int capacity)
    : contents(contents),
      capacity(capacity) { }
};
```

11.3 Destructors

- Object requires resources (memory, files,…)
- When object dies, **resources need to be freed**
  - Heap allocated memory must be freed
  - Files must be closed
  - …
- Constructor acquires resources
- **Destructor frees the acquired resources**
- Overview:

	Constructor	Destructor
Called	at creation	on destruction
Name	`classname()`	`~classname()`
Parameters	No restrictions	None
Overloads	Possible	Impossible

- Example:

```
class Foo
{
public:
  Foo();   // Constructor
  ~Foo();  // Destructor
};
```

11.3.1 Destructors in Java

- Java doesn't need to free memory thanks to **garbage collection**
- Files/streams/… *do* need to be released manually
- Java went through several solutions:

Solution 1: Finalize

- Object has **`finalize` method** (`protected`)
- Can be **overridden** in subclasses
- **Called by garbage collector when object became unreachable**
- Problems:

- Nondeterministic: you **never know when the GC will notice** object has become unreachable
- `finalize()` itself could **make the object reachable again: zombie objects**

```
class Foo
{
    public static Foo foo;

    @Override
    protected void finalize()
    {
        Foo.foo = this; // Uh oh
    }
}
```

Solution 2: Close

- Object has to be **closed manually by calling `close()`**
- Use with **try/finally**
- Problems
  - **Easy to forget**
  - Writing correct `try/finally` is actually complex
  - Compiler doesn't remind you
  - **Consistent naming not enforced** (e.g. `close`, `destroy`, `dispose`,…)

```
// Correct try-finally construct
Exception exception = null;
Resource res = new Resource();
try {
    // Do stuff
}
catch ( Exception e ) {
    exception = e;
    throw e;
}
finally {
    if ( exception != null )
        try {
            res.close();
        }
        catch ( Throwable t ) {
            exception.addSuppressed(t);
        }
    else
        res.close();
}
```

Solution 3: Closeable

- **Interface with `close` method**
- `try-with-resources`
- Introduced in Java 7
- Example:

```
try ( Scanner scanner = new Scanner(...) )
{
    // Use scanner
}
// scanner gets closed for you
```

11.5.2 Destructors in C++

- Destructors are **similar to `finalize`**
- Called **when object gets destroyed**

- Important difference: destructors are **fully deterministic**
- Examples:

```
void foo()
{
  T t;
  U u;
  X x;

  // ...

  return; // t, u, x get destroyed
}
```

```
T* t = new T;

// ...

delete t; // t's destructor gets called
```

```
T* ts = new T[10];

// ...

delete[] ts; // 10 destructors get called
```

```
void foo()
{
  T* p = new T;

  // NO destructor on T-object called!
  // It's the pointer p that's
  // going out of scope,
  // not the object itself
  // Use delete p to prevent leak
}
```

- **Usage**
  - Destructor **implicitly calls destructors of all fields**
  - Destructor only needs to clean up things that do not clean up themselves
  - A pointer's destructor does nothing: delete them in destructor
  - **Never call destructor yourself**
  - Examples:

```
class Foo
{
    int* p;

public:
    // Constructor allocates
    Foo() : p(new int) { }

    // Destructor deallocates
    ~Foo()
    {
        delete p;
    }
}
```

```
class Foo
{
  Bar bar;

public:
  ~Foo()
  {
    // bar gets cleaned up automatically
  }
};
```

11.4 Objects

- **Objects on the stack**

```
class Foo {
public:
    int x;
};

int main() {
    // Creates object on stack
    // Immediately ready for use
    Foo foo;

    foo.x = 5;
}
```

```
int main()
{
  // Calls default constructor Foo::Foo()
  Foo bar;

  // Calls constructor Foo::Foo(int)
  Foo baz(1);

  // WRONG, do not use parentheses!
  // It is interpreted as a function declaration
  Foo qux();
}
```

- **Objects on the heap**

```
#include <iostream>

int main() {
  Person* p = new Person("Jan", 20);

  std::cout << (*p).getName()
            << " is "
            << (*p).age
            << " years old"
            << std::endl;

  delete p;
}
```

- **Arrow operator**
  - `(*p).member` is **clumsy syntax**
  - `p->member`
  - Means the same thing
  - Shorthand

```
std::cout << p->getName()
          << " is "
          << p->age
          << " years old"
          << std::endl;
```

- **Differences with Java**
  - **Declaration**

```
Foo foo;
```

    - Java
      - `foo` is a reference to an object
    - C++
      - `foo` is the object itself
  - **Assignment**

```
Foo f, g;

f = g;
```

    - Java
      - After assignment, `f` and `g` refer to same object
    - C++
      - After assignment, `f` and `g` are still distinct objects
      - Assignment overwrites fields of `f` with fields of `g`
  - **Recursive data structures**

```
class Foo
{
public:
    Foo field;
};
```

    - Java
      - Each Foo refers to a Foo object
      - No infinite recursion, no infinite amount objects required
        - `field` could refer to owning object
        - `field` could contain null
    - C++
      - Invalid class declaration
      - A `Foo` actually contains another `Foo`
      - Matryoshka-like infinite recursion
      - Solution: pointer

12. Constructors

C++ has a few “special” constructors that get called in specific circumstances.

12.1 Default constructor

- Constructor with **zero parameters**
- **When are they called?**
  - When creating an object **without giving arguments**

```
class Foo {
public:
  Foo() { std::cout << "D"; }
};

int main() {
  Foo* p = new Foo; // prints D
}
```

  - When **initializing a container object** (e.g. array, vector,…)

```
class Foo {
public:
  Foo() { std::cout << "D"; }
};

int main() {
  Foo* p = new Foo[5]; // prints DDDDD
}
```

- **Automatic generation**
  - **Generated automatically if you do not define any constructors**
    - Calls base constructors
    - Calls default constructors on member variables
    - Does not initialize member variables to zero. (default constructor for int does not set value to 0)
  - Generation rules are quite complex
  - It is easier to be explicit about default constructors
  - **Examples:**
    - Force automatic generation when you already have a constructor:

```
class Foo
{
public:
    Foo() = default;
    Foo(int) { }
};
```

    - Prevent automatic generation:

```
class Foo
{
public:
    Foo() = delete;
};
```

12.2 Copy constructor

- Constructor taking single `const C&`
- Should make **deep copy** of given object
- **When are they called?**
  - **When creating a new object T from another T**

```
class Foo {
public:
    Foo(const Foo&) { std::cout << 'C'; }
};

Foo f;     // default constructor
Foo g(f); // prints C
```

  - **When passing an object by value**

```
class Foo {
public:
    Foo(const Foo&) { std::cout << 'C'; }
};

void bar(Foo f) { }

Foo foo;
bar(foo); // Call by value, copies foo
```

  - **When returning an object by value**

```
class Foo {
public:
    Foo(const Foo&) { std::cout << 'C'; }
};

Foo bar() {
    Foo f;
    return f;
}

// bar's return value gets copied to x
Foo x = bar();
```

- **Ambiguous behavior**
  - Compiler is allowed to transform your code
    - Tries to make it run faster
    - Behavior should remain unchanged
  - In some cases, C++ standard allows optimizations that change the behavior of the program, so maybe copy constructor does not even get called
  - Because of changing behavior
    - Copy constructor should only copy
    - Copy constructor should not have side effects
    - It should not matter how many times it gets called
  - See slides 17 - 22
- **Automatic generation**
  - Automatically generated, **similar rules to default constructor**

- Having no copy constructor is rather limiting
- Generated constructor **copies fields one by one**

12.3 Unary constructor

- Constructors for T that take single non-T parameter
- Act as **implicit casts**
- **When are they called?**
  - **When creating an object with one argument**

```
class Foo {
public:
    Foo(int) { std::cout << "U"; }
};

int main() {
    Foo* foo = new Foo(5); // prints U
}
```

  - **When a cast could help (implicit casting)**

```
class Foo {
public:
    Foo(int) { std::cout << "U"; }
};

void bar(Foo) { }

int main()
{
    bar(5); // prints U
}
```

- **Dangers of implicit casts:**
  - Generally, you don't want implicit casts
  - The cast-like behavior c**an easily lead to bugs**

```
class Color {
    int r, g, b;

public:
    Color(int r = 0, int g = 0, int b = 0)
        : r(r), g(g), b(b) { }
};

Color bar() {
    return true; // returns Color(1,0,0)
}
```

  - **Prevent implicit casting by making it explicit:**

```
class Foo {
    explicit Foo(int) { }
};

void bar(Foo) { }

int main() {
    bar(5);        // error
    bar((Foo) 5); // ok
}
```

12.4 Move constructor

- Called when **object is "moved"**
- Optimization
- Move constructors are important to understand smart pointers, details not important
- Lets an object "steal" internal fields from other objects
- T&& is "r-value reference"
    - Yet another kind of pointer referency thingamajig
    - We won't discuss the details

```
class Foo {
public:
    Foo(Foo&&); // Move constructor
};
```

- **Example:**

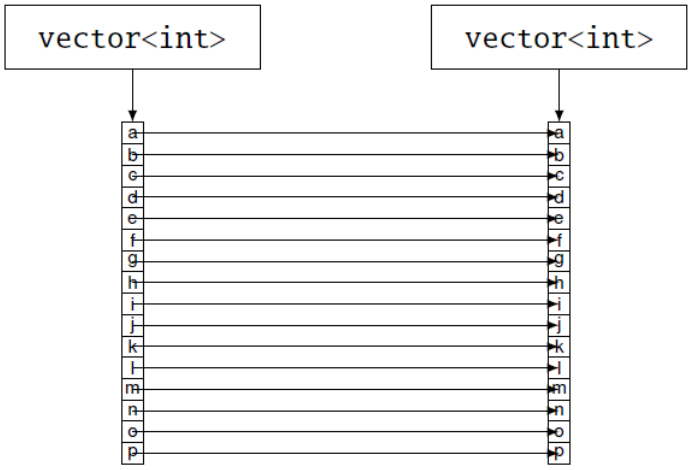
```
std::vector<int> foo() {
  std::vector<int> ns;
  // Fills ns with a zillion elements
  return ns;
}

std::vector<int> result = foo();
```

    - We're assuming no named return value optimization
    - When `foo` returns, `ns` must be copied to `result`
    - Since `foo` returns, `ns` must be destroyed too
    - So, we're making a copy only to destroy the original
    - Why copy at all?
    - Move constructor can be used in such circumstances
    - One object "takes over" all data from the original
    - The original object is "emptied"
    - Emptying original is important, as its destructor will still be called
    - Example:

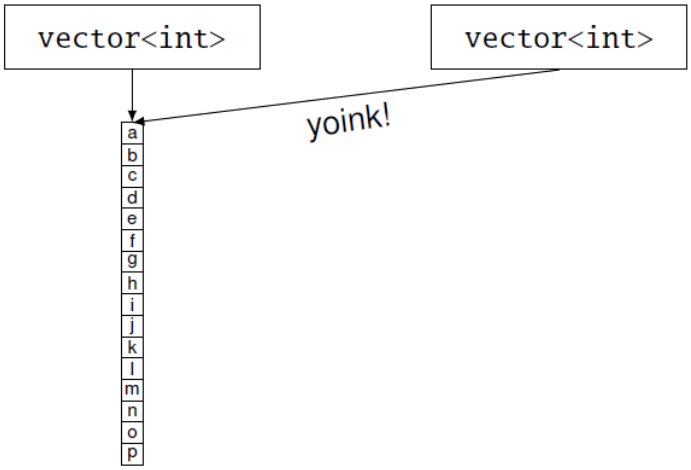
```
class IntArray {
  int *ns, size;

public:
  IntArray(int size)
    : ns(new int[size]), size(size) { }

  IntArray(IntArray&& other)
    : ns(other.ns), size(other.ns) {
    // DO NOT FORGET THIS
    other.ns = nullptr;
    other.size = 0;
  }

  ~IntArray() { delete[] ns; }
};
```

12.4.1 Copy vs. Move constructor

Copy constructor



Move constructor



13. Structs

- In C: let you group data together (only member variables)
- In C++: struct is **identical to `class`**
- One exception: **default visibility is `public`**

```
class Foo
{
  // x is private
  int x;
};
```

```
struct Foo
{
  // x is public
  int x;
};
```

14. Sizeof

- `sizeof` probably does not do what you think it does
- `sizeof` **returns size of data structure in bytes**
- Was needed in C
  - In C, you allocate heap memory using `malloc`
  - `malloc` needed to be told how many bytes to allocate

```
struct Foo {
    int x;
    char* p;
};

// C-style allocation
Foo* foo = (Foo*) malloc(sizeof(Foo));
```

  -
- **What `sizeof` is not**
  - `sizeof` does not count the number of items in a container
  - Don't use `sizeof`
- **What `sizeof` does**

```
class vector {
    int* items;
    int capacity;
    int size;

public:
    // Member functions
};
```

  - `sizeof` tells you the size of the object
  - `sizeof(vec) == 4 + 4 + 4 = 12` (assumbing 32 bit)
    - Will always be 96, regardless of actual number of items
    - Memory pointed to by items not counted by `sizeof`
- **Cake points!**
  - `sizeof(Foo) == 12`
  - `sizeof(Bar) == 8`
  - A `char` is usually 1 byte, ant `int` 4 bytes (depending on register)
  - Solution: **alignment**
    - An `int` is aligned on 4 bytes. This means the address of `Foo.b` must begin on a multiple of 4. A `struct` is also aligned on 4 bytes. A `char` is aligned on 1 byte.
    - Between `Foo.a` and `Foo.b`, there is 3 bytes padding. Same for `Foo.c` and the end of the `struct`.
    - Between `Bar.c` and the end of the `struct` is 2 bytes of padding.

```
struct Foo {
    char a;
    int b;
    char c;
};

struct Bar {
    int a;
    char b;
    char c;
};
```

15. Const correctness

- Type `const T` = readonly view of T
- Forbids write operations
- **How does compiler know which members are OK to call?**
- **Suffixing a method with `const`** means it's a read-only method
  - In non`const` method: `this` has type `T*`
  - In `const` method: `this` has type `const T*`
- **Compiler enforces `const`ness**
  - Trying to modify object in const method → error
- Example:

```
class OilTank {
  int contents, capacity;

public:
  int getContents() const;
  int getCapacity() const;
  int getFreeCapacityLeft() const;

  void fill(int amount);
  void drain(int amount);
  void transfer(OilTank&, int amount);
};
```

- **Overloading**
  - It is possible to define a **`const` and non`const` version** of a method
  - The `const` version will be called if accessed through `const` variable
  - The non`const` version will be called if accessed through non`const` variable
  - **Example:**

```
#include <iostream>

class Foo {
public:
  void bar() { std::cout << "A"; }
  void bar() const { std::cout << "B"; }
};

int main() {
  Foo* p = new Foo();
  const Foo* q = p;

  p->bar(); // prints A
  q->bar(); // prints B

  delete p;
}
```

  - **Another example:** see slides 11 - 15

16. Default Parameter Values

- Problem: we don't want the programmer to have to explicitly specify all parameter values
- e.g. if he wants to create a new writeable file, he should only have to mention the filename

16.1 Solution 1: Overloading

- **Example:**

```
file open_file( const std::string& filename,
                bool writeable,
                bool createIfMissing );

file open_file( const std::string& filename,
                bool writeable );

file open_file( const std::string& filename,
                bool createIfMissing );

file open_file( const std::string& filename );
```

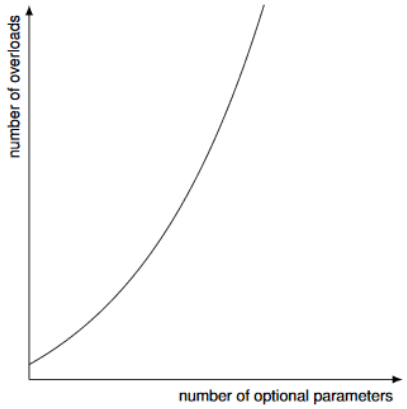
- **Problems:**
  - Overloads are not allowed to have same signatures (same parameter types)

```
file open_file( const std::string& filename,
                bool writeable );
// clashes with
file open_file( const std::string& filename,
                bool createIfMissing );
```

  - Order of parameters often arbitrary
  - Unclear frpm call syntax which parameter gets which value
  - Need IDE help/documentation to make sense of arguments

```
file f = open_file("file.txt", true, false);
```

  - Exponential number of overloads required
  - Lots of boilerplate code



16.2 Solution 2: Builder Design Pattern

- **Example:**

```
class open_file
{
public:
  builder& writeable(bool b) { ... }
  builder& createIfMissing(bool b) { ... }

  file finish();
};

file f = open_file().writeable(true).finish();
```

- **Problem**: lots of boilerplate code

16.3 Solution 3: Default Parameters

- **Missing parameters are filled in for you using default values**
- **Example:**

```
file open_file( const std::string& filename,
                bool writeable = true,
                bool createIfMissing = true );

open_file(fn, false);
// is same as
open_file(fn, false, true);

open_file(fn);
// is same as
open_file(fn, true, true);
```

- **Limitations:**

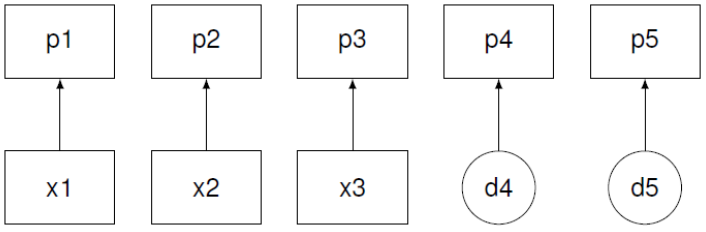
```
void foo(bool x = false, int y = 5);

foo(2);
// calls
foo(true, 5);
```

- **Rules**
  - **Declaring functions with default parameters**
    - Cannot skip parameters: if *i*-th parameter has default value, all subsequent parameters must also have default values

```
// Correct
void foo(int x    , int y    , int z    );
void foo(int x    , int y    , int z = 0);
void foo(int x    , int y = 0, int z = 0);
void foo(int x = 0, int y = 0, int z = 0);

// Wrong
void foo(int x = 0, int y    , int z    );
void foo(int x    , int y = 0, int z    );
void foo(int x = 0, int y = 0, int z    );
```

  - **Calling functions with default parameters**
    - *i*-th argument gets assigned to *i*-th parameter
    - No intelligence whatsoever

```
void foo(T1 p1 = d1, T2 p2 = d2, T3 p3 = d3,
         T4 p4 = d4, T5 p5 = d5);

foo(x1, x2, x3);
```



- **Single declaration**
  - Default values allowed in both declarations and definition
  - Compiler may only encounter default values once
  - Compiler has to know defaults when you use them
  - Best practice: **put default parameter values in the header file**

```
// Declaration
void foo(int x = 1);

// Ok: default value known from declaration
foo();

// Error! Second time default
// value is encountered
void foo(int x = 1) { ... }
```

```
// Declaration
void foo(int x = 1);

// Ok: default value known from declaration
foo();

// Definition
void foo(int x) { ... }
```

```
// No mention of default value
void foo(int x);

// Ok, since no need for default value
foo(3);

// Definition
void foo(int x = 1) { ... }

// Ok, default values are known
foo();
```

```
// Declaration
void foo(int x);

// Error: compiler thinks there no default values
// Cannot look into the future
foo();

// Defaults defined here
void foo(int x = 1) { ... }
```

17. Operator Overloading

17.1 Introduction

- **Overloading = giving multiple meanings to the same symbol**
- Abundant in mathematics, e.g. symbol + denotes addition of naturals, integers, reals, complex numbers, matrices,…
- In Java: + works on `int`, `double`, `String`
- Same symbol but **different internal implementation**
- Operator overloading: **to be able to define meaning of an operator for your own types**
- **Example:**
  - Example WITHOUT operator overloading

```
// Usage
Vector2D u(1, 2);
Vector2D v(4, 6);

Vector2D z = u.add(v).mul(5);
```

  - Example WITH operator overloading

```
// Usage
Vector2D u(1, 2);
Vector2D v(4, 6);

Vector2D z = (u + v) * 5;
```

17.2 Syntax

- **Unary operators**
  - Unary operators take **one parameter**

```
// Either as member function
struct T {
    T operator !() const;
}

// or as global function
T operator !(const T& t) { ...  }

// Usage
T t;
T anti_t = !t;
```

- **Binary operators**
  - Binary operators take **two parameters**
  - Any type combination possible
    - `Foo` + `Bar` can yield `Qux`
  - You cannot change existing type combinations
    - E.g. you cannot redefine `int + int`

```
// As member function
struct T {
    T operator *(const T& other) { ... }
};

// As global function
T operator *(const T& t, const T& u)
{
    ...
}

T t, u;
T product = t * u;
```

- **Member function vs. Global function**
  - You can overload operators in two ways
    - Defined as member functions
    - Defined as global functions
  - Either as member or global, but **not as both**, otherwise definitions would clash
  - Most of the time you can choose
    - It's a **matter of preference**
  - Sometimes only member function will work
  - Sometimes only global function will work
- **Global function**
  - When defined as a member of T, left operand is always a T
  - If you want another left operand type, use a global function
  - Example:

```
Vector2D operator *(double c,
                    const Vector2D& v)
{
  // Delegates to Vector2D::operator *(double)
  return v * c;
}
```

  - **Problem:** what if operator *must* be global and needs access to private members
  - **Solution: friend functions**
    - A class can give access to its private parts
    - It can declare functions to be `friend`s

```
class Vector2D
{
    // Private stuff

public:
  friend Vector2D operator *(double, const Vector2D&);
};

Vector2D operator *(double c, const Vector2D& v)
{
  // Can access private members of v
}
```

17.3 Literals

- **Example**
  - Say you have a class called Duration
  - It represents time durations
  - **How to create Duration objects?**
    - **Solution 1: constructor**

```
class Duration
{
public:
    Duration(double);
};

Duration duration(20);
```

      - Bad idea
      - 20 seconds? 20 days?

- **Solution 2: static factory methods**

```
class Duration
{
public:
    static Duration from_seconds(double);
    static Duration from_minutes(double);
    static Duration from_hours(double);
    static Duration from_days(double);
    // ...
};


auto d = Duration::from_seconds(5.3);
```

  - Much clearer
  - Rather verbose

- **Solution 3: literals**

```
Duration operator "" _s(long double x) {
    return Duration::from_seconds((double) x);
}

Duration operator "" _minutes(long double x) {
    return Duration::from_seconds((double) x);
}

Duration operator "" _hours(long double x) {
    return Duration::from_hours((double) x);
}

// Usage
Duration d1 = 5_s, d2 = 10_hours;
```

17.4 Examples

- **Indexing**
  - Indexing `[]` only defined on pointers/arrays
  - However, you can overload `[]` for your own containers
  - For example, `std::vector` overloads `[]`

```
template<typename T>
class vector
{
  T* items;

public:
        T& operator[](int index);
  const T& operator[](int index) const;
};
```

- **`toString`**

```
struct Date
{
    int day, month, year;
};

// Day of the devil
Date date(6, 9, 1993);

// We want this to print "6/9/1993"
std::cout << date;
```

  - How to make this work?

- **Solution:** overload << on `std::ostream`

```
std::ostream& operator <<(ostream& out,
                          const Date& date)
{
    out << date.day
        << "/"
        << date.month
        << "/"
        << date.year;

    return out;
}
```

- **Assignment**

```
class IntList {
    int* items;
    int size;
    // ...
};

IntList xs, ys;
xs = ys;
```

  - By default, assignment *between* objects **overwrites the fields of the left operand** with the values of the fields of the right operand
  - `xs` and `ys` are **still distinct objects** after assignment
  - `xs` and `xy` would **share same internal array**
  - We need `xs` and `ys` to have separate internal arrays

```
class IntList {
    int* items;
    int size;

public:
    IntList& operator =(const IntList& xs) {
        delete[] items;
        items = new int[xs.size];

        for ( int i = 0; i != xs.size; ++i )
            items[i] = xs.items[i];

        return *this;
    }
};
```

18. RAII

18.1 Resources

- **What are resources?**
  - Can be acquired
  - While acquired, can generally not be shared
  - Must be released so that others can use it
- **Examples of resources**
  - Memory
    - Example: Heap memory
    - Acquiring: `T* p = new T`
    - Releasing: `delete p;`
    - Sharing not allowed: two objects can never share memory
    - Not releasing lead to memory leaks, leads to memory exhaustion, leads to crash, leads to unhappy users, which we don't want
  - Files
    - Sharing is possible
    - Multiple readers
    - Single writer
    - Multiple writers: feasible but difficult
    - Files must be closed
    - Consequences of not closing:
      - Readers block writers
      - Writers block readers and writers
  - Ports (networking)
  - GPU (pre Vista)

18.2 Resource Management

- Correct resource management is important
- Always free your resources
- Resource exhaustion is a Bad Thing™
- Keep exceptions in mind
- Resource management in an Ideal World™:
  - You want the resource to be freed automatically
  - You don't want to be able to forget to free resources
  - You want to know when the resource gets freed
  - You don't want to have to take exceptions into account
- Java: try with resources (see 11.3.1)
  - Deterministic
  - No code necessary to release, hence cannot be forgotten
- C++
  - Has no `try/finally` or try-with-resources
  - **RAII: Resource Acquisition is Initialization**
  - Resource is represented by object
  - Object has destructor
  - Destructor is responsible for releasing resource

18.3 Resource management in C++

- **Explicit release**

```
void foo()
{
  int* x = new int[10];

  // ...

  delete[] x;
}
```

  - Horribly fragile
  - What if an exception is thrown inside the `// ...` ?
  - What if there are multiple returns? Multiple `delete`s needed.

- **Deleter**

```
class deleter {
  int* p;
public:
  deleter(int* p) : p(p) { }
  ~deleter() { delete[] p; }
};

void foo()
{
  int* p = new int[10];
  deleter deleter_p(p);

  // ...

  // deleter_p goes out of scope here
  // its destructor gets called
  // destructor frees memory
}
```

  - `deleter` auxiliary objects
  - Destructor will be called whenever it goes out of scope
    - On function's end of body
    - At every `return`
    - At every exception
  - Still need to think about a `deleter`
  - Clumsy solution

- **Destructor**
  - Every **resource should be represented by an object**
  - The **destructor should always release the resource**
  - **No leaks possible**
  - **Example:**

```
class intarray {
  int* elts;

public:
  intarray(int size)
    : elts( new int[size] ) { }

  ~intarray() {
    delete[] elts;
  }
};

void foo() {
  intarray ns(10);
  // ...
  // ns's destructor gets called here
}
```

- **Example: memory resource**

```
void foo()
{
  std::vector<int> ns;

  // ...

  // ns goes out of scope here
}
```

  - `ns` allocates memory
  - This memory is deallocated automatically by `ns`'s destructor
  - `vectors` can be seen as "automated resource management for heap allocated arrays"

- **Example: file resource**

```
int read_from_file(std::string path)
{
  // Open file
  std::ifstream file(path);

  // Read data from file (same as std::cin)
  int x;
  file >> x;

  // Return read data
  return x;

  // file goes out of scope
  // destructor closes file automatically
}
```

19. Smart Pointers

19.1 Smart pointers

- **RAII**
  - RAII = resource management
  - Destructor is in charge of cleanup
  - No resource leaks anymore
  - Examples: `ifstream` for reading files, `vector`,…
  - What about heap allocated memory?
- **Heap allocation: solution 1**

```
void foo() {
  Person* p = new Person("Jorg");
  // stuff
  // Memory leak
}
```

  - - Memory leak
- **Heap allocation: solution 2**

```
void foo() {
  Person* p = new Person("Jorg");
  // stuff
  delete p;
}
```

  - + Memory gets deleted if all goes well
  - - Memory leak in case of exceptions
  - - Easy to forget delete
  - - Fragile if multiple exit points
- **Heap allocation: solution 3**
  - RAII to the rescue
    - What if we relied on destructors?
    - Give pointer to helper object
    - Helper object becomes responsible for freeing memory

```
template<typename T> class smart_pointer {
  T* p;  // Pointer

public:
  // Constructor receives pointer
  smart_pointer(T* p) : p(p) { }

  // Destructor frees memory
  ~smart_pointer() { delete p; }

  // Gives access to pointer
  T* get() { return p; }
};

void foo() {
  smart_pointer<Person> p(new Person);
  // stuff
}
```

  - + Memory guaranteed to be freed
  - + `smart_pointer` usable on all types
  - – Clumsy syntax to access members: `p.get()->member`

- **Solving the syntax issue**

```
template<typename T>
class smart_pointer {
  T* p;
public:
  // ...
  T* operator ->() { return p; }
};

smart_pointer<Person> p(new Person);

// Valid syntax
std::string name = p->getName();
```

  - Overloading `->` operator has **special rules**
    - Operator `->` must return either
      - a pointer
      - an object with `->` defined on it
    - When using `obj->m`
      - If object is pointer, standard meaning
      - If object is object, operator `->` is called, and `->` is again called on its return value
    - In other words, **`->` is called repeatedly on results until a pointer is returned**
    - Example: slides 12 - 20
- **Dereferencing smart pointers**

```
void foo(const Person&);

template<typename T>
class smart_pointer {
  T* p;
public:
  // ...
  T& operator *() { return *p; }
};

smart_pointer<Person> p(new Person);

// Works
foo(*p);
```

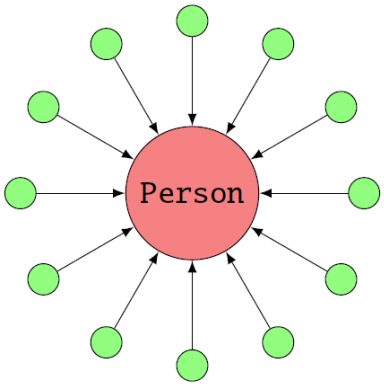
  - Overloading * operator
- **Dumb pointers vs Smart pointers**
  - Dumb pointer `T*`
  - Smart pointer: `smart_ptr<T>`
  - Smart pointers should automatically `delete` themselves when necessary
    - Achieved using destructor
  - Same syntax for dumb as for smart pointers
    - Achieved using operator overloading
    - Operators `->`, `*`, `[]`, etc.

19.2 Ownership

- **Problem**

```
void make_older(smart_pointer<Person> p) {
  p->age++;
}

smart_pointer<Person> q(new Person(20));
make_older(q);
std::cout << "Age: " << q->age << std::endl;
```

  - We create a new person `q`, aged 20,
  - We call `make_older`, passing the smart pointer by value
  - `make_older` increases the age
  - `p` goes out of scope, and it deletes the `Person`
  - We try to print `q`'s age but `q` points to invalid memory



  - When copying smart pointers, they point to the same object
  - When one smart pointer dies, it will `delete` the object
  - All other pointers become invalidated

- **Ownership**
  - **Ownership**: **you're responsible for freeing** the object
  - **No ownership**: **you're forbidden to free** the object
  - A **`smart_ptr` gives ownership**
  - A regular **`T*` gives no ownership**
  - Problem with previous code: 2 smart pointers = 2 owners = bad
  - Fix:

```
// make_older does not need ownership
// so we use Person* instead
// of smart_ptr<Person>
void make_older(Person* p) {
  p->age++;
}

smart_pointer<Person> q(new Person(20));
make_older(q.get());
std::cout << "Age: " << q->age << std::endl;
```

  - `make_older` will not free memory, since it is not given ownership over the `Person`

- **Prohibiting multiple ownership**
  - For every object, we want exactly one owner
  - We need to prohibit copying smart pointers

```
void bar(smart_pointer<T> p) { ... }

void foo() {
  smart_pointer<T> p(new T);

  // bar would receive a copy of p
  // there would be two owners
  // we need to prevent this
  bar(p);
}
```

  -
  - Remove copy constructor
  - Remove assignment operator

```
template<typename T> class smart_ptr
{
public:
  // Explicit "no copy constructor"
  smart_ptr(const smart_ptr&) = delete;

  // Explicit "no assignment"
  smart_ptr&
    operator =(const smart_ptr&) = delete;
};
```

  -
- **Transferring ownership**
  - Sometimes we do want to transfer ownership
  - Write a `transfer` function
  - Technical details not important

```
smart_pointer<T> p(new T);
smart_pointer<T> q(nullptr);

// Transfer ownership from p to q
q = transfer(p);

// p == nullptr
// q points to object
```

  - Example

```
class Class {
  std::vector<smart_ptr<Student>> students;
public:
  void add(smart_ptr<Student> student) {
    students.push_back(student);
  }
};

Class c;
smart_ptr<Student>(new Student("Brad"));

c.add( s ); // Fails
c.add( transfer(s) ); // Succeeds

// When c goes out of scope,
// all students will be deleted
```

- **`std::unique_ptr`**
  - Since C++11
  - Smart pointer present in standard library
  - Named `std::unique_ptr`
  - Creation: `std::make_unique<T>(args)`
  - Transferring: `q = std::move(p)`
  - `#include <memory>`
  - Example:

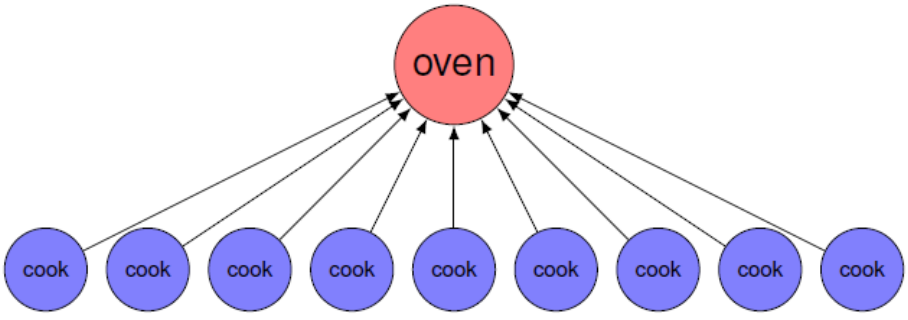
```
class Class {
  std::vector<std::unique_ptr<Student>> members;

public:
  void add(std::unique_ptr<Student> student) {
    members.push_back(student);
  }
};

class Student {
public:
  Person(std::string, int);
};

Class c;
auto p = std::make_unique<Student>("Jan", 20);

c.add(std::move(p));
```

  - `std::make_unique<T>` takes the same arguments as T's constructors

```
class Foo {
public:
  Foo(int);
  Foo(bool, double);
};

std::make_unique<Foo>(5); // ok
std::make_unique<Foo>(true, 5.0); // ok
std::make_unique<Foo>(false); // not ok
```

19.3 Shared ownership

- **Problem with single ownership**
  - Say you have an object O
  - Say many other objects need access to O
  - Who should have ownership?



- **Solution 1:**
  - Give one cook ownership
  - This cook takes oven with him to his grave
  - We have to make sure the oven owning cook dies last
  - Otherwise, all other cooks would be using a phantom oven
  - Critique
    - Solution seems quite arbitrary
    - Fragile
- **Solution 2:**
  - Introduce new object `Kitchen`
  - Kitchen owns oven and cooks
  - Kitchen has to survive cooks and oven
  - Critique
    - Structured
    - Relatively robust
    - Absolute need to keep kitchen alive
- **Solution 3:**
  - Introduce **shared ownership**
  - Each cook owns oven
  - Oven keeps track of number of cooks
  - New cook increments count
  - Dead cook decrements count
  - When no cooks left, oven self-destructs
  - Critique
    - Structured
    - Robust
    - There are shortcomings, but we'll ignore them

- **`shared_ptr`**
  - Standard library offers readymade solution **`std::shared_ptr<T>`**
  - Creation: `make_shared<T>(args)`
  - **No ownership transfer necessary**
  - When shared ptr is duplicated, reference count goes up
  - When shared ptr goes out of scope, reference count goes down
  - When reference count reaches 0, memory is freed
  - **We'll be using this pointer most from now on**
  - **No more T***
- **Summary**
  - **Stay away from dumb pointers** as much as you can
  - Use **`std::shared_ptr<T>`** instead of **`T`**
  - Use **`std::make_shared<T>`** instead of **`new`**

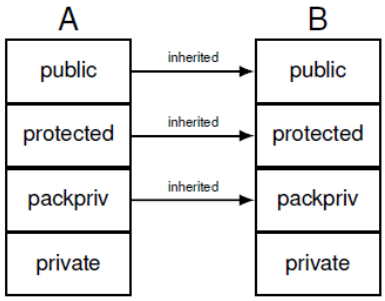
20. Inheritance

20.1 `protected`

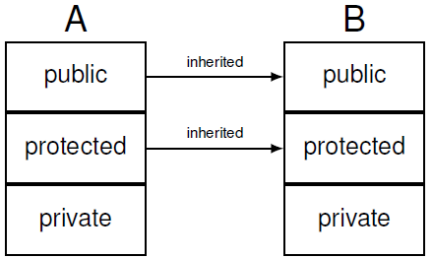
- Who can reach a protected member?
  - Objects of the same class
  - Objects of a subclass
- Who cannot reach a protected member?
  - Objects of a class that is not a subclass
- Not like Java: in C++ it is not possible for objects of the same package who are not a subclass to reach a protected member.

20.2 Types of inheritance

- **Java (public inheritance)**
  - When B inherits from A
    - A's public members → B's public members
    - A's protected members → B's protected members
    - A's package private members → B's package private members
    - A's private members are not inherited



- **C++ public inheritance**
  - **Scheme:**



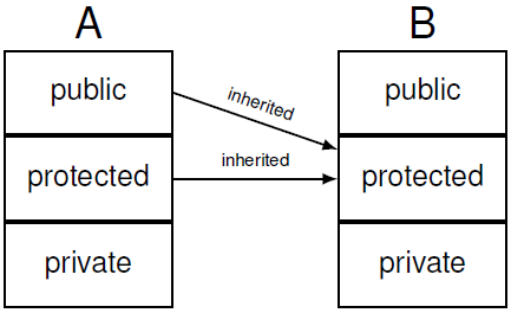
  - **Syntax:**

```
class A { ... };
class B : public A { ... };
```

  - **Example:**

```
class A {
public:
  int x;
};

class B : public A { };

void foo()
{
  B b;

  b.x = 5; // Ok, x is public member of B
}
```

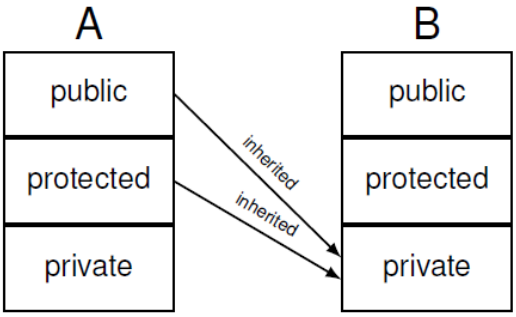
- **C++ protected inheritance**
  - **Scheme:**



  - **Syntax:**

```
class A { ... };
class B : protected A { ... };
```

- **C++ private inheritance**
  - **Scheme:**



  - **Syntax:**

```
class A { ... };
class B : private A { ... };
```

  - **Example:**

```
class A {
public:
  int x;
};

class B : private A { };

void foo()
{
  B b;

  b.x = 5; // Error, x not accessible
}
```

- **Liskov violated?**
  - Protected and private inheritance seemingly violate LSP
    - Superclass has public member X
    - Then every subclass should also have public member X
    - Not the case with protected/private inheritance
  - **Not violated:**
    - Subtype relation changes
    - In C++: public inheritance means B is a subtype of A (like in Java)
    - But nonpublic inheritance means **B is not a subtype of A**

```
class A { };
class B1 : public A { };
class B2 : private A { };

void foo(const A&);

B1 b1;
B2 b2;

foo(b1); // ok
foo(b2); // not ok
```

20.3 Virtual methods

- **What?**
  - **Methods that are overridable**
  - When calling a **virtual method, dynamic type determines** which version is called
  - When calling a **nonvirtual method, static type determines** which version is called
  - **Java**: all nonprivate object methods are virtual
- **C++**
  - If you want a member function to be overridable, you have to be explicit
  - To make virtual: add `virtual`
  - To override: add `override` (this is optional but encouraged)

```
struct A {
  virtual void bar();
};

struct B : public A {
  void bar() override;
};
```

- **Example**

```
struct A {
  virtual void foo() { print("A"); }
};

struct B : public A {
  void foo() override { print("B"); }
};

A* aa = new A();
A* ab = new B();
B* bb = new B();

aa->foo(); // A
ab->foo(); // B
bb->foo(); // B
```

```
struct A {
  void foo() { print("A"); }
};

struct B : public A {
  void foo() { print("B"); }
};

A* aa = new A();
A* ab = new B();
B* bb = new B();

aa->foo(); // A
ab->foo(); // A
bb->foo(); // B
```

- **Inheritance in C++**
  - When working with inheritance, **always use pointers/references**
  - When working by-value, inheritance won't work correctly

- Reason: object slicing

  "Slicing" is where you assign an object of a derived class to an instance of a base class, thereby losing part of the information - some of it is "sliced" away.

  For example,

  ```
  class A {
     int foo;
  };

  class B : public A {
     int bar;
  };
  ```

  So an object of type `B` has two data members, `foo` and `bar`.

  Then if you were to write this:

  ```
  B b;

  A a = b;
  ```

  Then the information in `b` about member `bar` is lost in `a`.

- **Abstract**
  - **Abstract method = virtual method without body**
  - C++ has **no abstract keyword:** to declare abstract method, **use = 0**

    ```
    struct Foo {
      virtual double foo() = 0;
    };
    ```

  - C++ class is **automatically abstract** when it contains 1 or more abstract methods
  - Abstract classes **cannot be instantiated**
  - **Examples:**

    ```
    struct Foo {
        virtual void bar() = 0;
    };

    int main()
    {
        Foo foo;           // Error
        Foo* p = new Foo; // Error
    }
    ```

    ```
    struct Shape {
      virtual double area() const = 0;
    };

    struct Square : public Shape {
      double side;

      double area() const override {
        return side * side;
      }
    };
    ```

20.4 Multiple inheritance

- **What**
  - **One class can have multiple superclasses**
  - **Has many problems**
  - Java simplified things by introducing interfaces, **C++ does not have interfaces**
  - You can **fake an interface**: classes with **only abstract methods**

- **Example:**

Java version

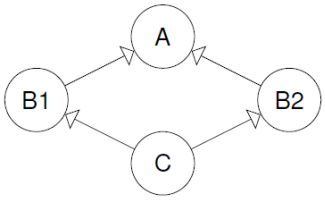
```
interface Quackable {
  void quack();
}

abstract class Bird {
  public abstract void fly();
}

class Duck extends Bird implements Quackable {
  @Override
  void fly() { ... }

  @Override
  void quack() { ... }
}
```

C++ version

```
struct Quackable {
  virtual void quack() = 0;
}

struct Bird {
  virtual void fly() = 0;
}

class Duck : public Bird, public Quackable {
  void fly() override { ... }
  void quack() override { ... }
}
```

- **Problems with multiple inheritance**
  - **Diamond problem**
    - C inherits from A twice
    - C inherits x twice: each C object has two x variables

```
struct A {
  int x;
};

struct B1 : public A { };
struct B2 : public A { };

struct C : public B1, public B2 { };
```



  - **Preventing diamond problem**
    - **Virtual inheritance**
    - A is only inherited once by C
    - Has some weird consequences
    - **Generally discouraged**
    - Just know that it exists

```
struct A {
  int x;
};

struct B1 : public virtual A { };
struct B2 : public virtual A { };

struct C : public B1, public B2 { };
```

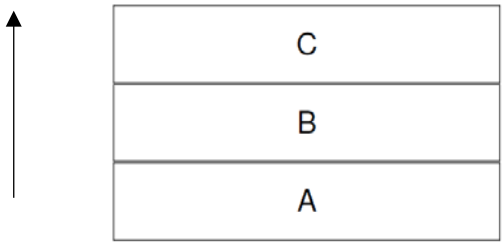
20.5 Other details

- **Constructors**
  - First thing a constructor must do is call a **superconstructor**
  - Akin to building a house: stories built from bottom to top
  - Same as in Java:

```
struct A {
  A(int);
};

struct B : public A {
  B() : A(0) { }
};

struct C : public B {
  C() { } // calls B::B() implicitly
};
```

  - **Stepwise object construction**

```
struct A { };
struct B : public A { };
struct C : public B { };

C c;
```

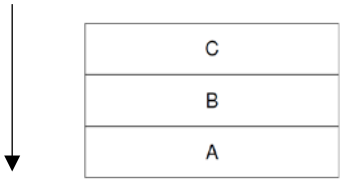


  - Calling **virtual functions in a constructor: don't do it**

```
struct A {
  A() { foo(); }
  virtual void foo() { print "A"; }
};

struct B : public A {
  B() { foo(); }
  virtual void foo() { print "B"; }
};

struct C : public B {
  C() { foo(); }
  virtual void foo() { print "C"; }
};

C c; // In Java: CCC, in C++: ABC
```

- **Destructors**
  - Last thing a destructor must do is **call a superdestructor**
  - Always happens **automatically**, no need to do it yourself
  - Akin to destroying a house: **stories removed from top to bottom**
  - **Stepwise object destruction**

```
struct A { };
struct B : public A { };
struct C : public B { };

C* p = new C;
delete p; // destruction
```



  - **Important**

```
struct A { };

struct B : public A {
  int* data;

  ~B() { delete[] data; }
};

A* p = new B();

// Would call A::~A(), not B::~B()
// due to the fact that A::~A() is
// not virtual
delete p;

// Memory leak! data never freed
```

    - **`delete` needs to know the dynamic type** so as to call the correct destructor
    - **Destructor** must be virtual

```
struct A {
  virtual ~A() { }
};
```

    - **Can't be abstract**

```
struct A {
  virtual ~A() = 0; // wrong!
};
```

    - They can only be abstract if you provide a function body for the abstract constructor. The reason for this is that destructors are special operations during which all destructors in a class hierarchy are always called, **including the superclasses' destructor**. More information here: http://www.cplusplus.com/forum/general/12712/
    - Rule of thumb: **add virtual destructor to each class**
- **Destructors and smart pointers**
  - Smart pointers remove the need for virtual destructor
  - A smart pointer keeps track internally of which destructor to call
  - Still safer to always add virtual destructors in case someone works with your class through dumb pointers

21. Templates

21.1 Generics

- **What**
  - Java and C# support generic classes
  - **Classes parameterized** in one or more type variables
  - Typical use: container classes
    - `List<T>`
    - `Set<T>`
    - `Map<K, V>`
    - …
  - You can define your own
  - Example:

```
// C# code (Java is more complicated)
class List<T> {
  private T[] items;
  private int size;

  public List() { items = new T[10]; }

  public void Add(T x) {
    if ( size == items.Length )
       DoubleCapacity();

    items[size] = x;
    size++;
  }
}
```

- **Type checks**
  - As always, compiler type checks your code
  - Since T could be any type, **you cannot make any assumptions** about it
  - Code below won't compile

```
class Foo<T> {
  private T x;

  public void Frobnicate() {
    x.Muzzify();
  }
}
```

- **Type bounds**
  - If your Foo<T> class needs to interact with T objects, you can **demand that T is a subtype of some other type**
  - Foo now only accepts Ts that are muzzifiable

```
interface IMuzzifiable {
  void Muzzify();
}

class Foo<T> where T : IMuzzifiable
{
  private T x;

  public void Frobnicate() {
    x.Muzzify();
  }
}
```

21.2 C++ templates

- In C++, `std::vector<T>` is not called a generic
- It is called a **template**
- Templates **differ fundamentally** from generics
- Templates are much **more powerful than generics**
- Templates are also much **more fragile**
- **What are templates?**
  - Templates can be seen as "code generators"
  - Using `std::vector<int>` makes the compiler generate code from `std::vector<T>` with T replaced by `int`
  - Example:

```
template<typename T>
class Array {
  T*  _xs;
  int _size;

public:
  // Constructor
  Array(int size) : _xs(new T[size]), _size(size) { }

  // Destructor
  ~Array() { delete[] _xs; }

  // Indexing
        T& operator[](int i)       { return _xs[i]; }
  const T& operator[](int i) const { return _xs[i]; }

  // Size
  int size() const { return _size; }
};
```

- **Templates generated by need**
  - Exact generation rules are rather complex
  - Only the member functions you actually use are generated
  - **Function bodies not checked if you don't call them**
  - Example below would result in compiler error when calling `bar()`

```
// Everything still ok
template<typename T>
struct Foo {
    void bar() {
        djflksdjfl dskfljsd
        fjqipodf cm ckmal 778!
    }
};

int main() {
    Foo<int> foo;
}
```

  - **Consequences**
    - By need generation gives extra flexibility
    - **Allows you to define functions that only work for certain Ts**

```
template<typename T> class Array {
  T* _xs;
  int _size;

public:
  // Only sensible for addable types
  T sum() const {
    T result = 0;

    for ( int i = 0; i != _size; ++i )
      result += _xs[i];

    return result;
  }
};
```

- **What if T were a `std::string`?**

```
class Array<std::string> {
  std::string* _xs;
  int _size;

public:
  std::string sum() const {
    std::string result = 0;

    for ( int i = 0; i != _size; ++i )
      result += _xs[i];

    return result;
  }
};
```

  - **Summation = concatenation**
  - However, `std::string result = 0` does not what you want
    - It calls the constructor taking a `const char*`
    - It initializes the string with characters located at memory address `0` → **undefined behavior**

- Solution**: template specialization**
  - We need to associate a "zero element" with each type

Type	Zero value
int	0
double	0.0
std::string	""
⋮	⋮

```
T sum() const {
  T result = typeinfo<T>::zero();

  for ( int i = 0; i != _size; ++i )
    result += _xs[i];

  return result;
}
```

  - It is possible to give different definitions for different types

```
template<typename T>
struct typeinfo
{
    // Empty
};
```

```
template<>
struct typeinfo<int>
{
    static int zero()
    {
        return 0;
    }
};
```

```
template<>
struct typeinfo<std::string>
{
    static std::string zero()
    {
        return "";
    }
};
```

- Template functions
  - You can also define template functions

```
template<typename T>
T sum(const std::vector<T> ns)
{
  T result = typeinfo<T>::zero();

  for ( const T& n : ns )
    result += n;

  return result;
}
```

21.3 Implementation Details

- **Code organization**
  - For **non-templated** functions/classes
    - Put declarations in `.h` file
    - Put definitions in `.cpp` file
  - For **templated** functions/classes
    - Put **entire definition in `.h`** file
- **Reason**
  - Compiler needs to be able to generate code from template
  - For this, **compiler needs source code** of template definition
  - If template definition were in separate `.cpp` file, the compiler wouldn't have access to it

22. Casts

- **Casting = explicit type conversion**
- **Upcasting = casting to a supertype**
- **Downcasting = casting to a subtype**
- Upcasts are always safe, downcasts can go wrong:

```
struct Animal { virtual ~Animal(); };
struct Dog : public Animal { };
struct Cat : public Animal { };

Cat* cat = new Cat;
Animal* animal = cat;      // Implicit upcast
Dog* dog = (Dog*) animal; // Wrong downcast
```

- **Java**
  - Only one way to cast: `(T) x`
  - Java objects keep track of their "real type" (dynamic type)
  - Casts are checked at runtime
  - Wrong casts: `IllegalCastException`
- **C++**
  - C-style cast: `(T) x`
  - `static_cast<T>(x)`
  - `dynamic_cast<T>(x)`
  - `reinterpret_cast<T>(x)`
  - `const_cast<T>(x)`
- **C++ standard library**
  - `std::static_pointer_cast<T>(x)`
  - **`std::dynamic_pointer_cast<T>(x)` (most important one)**

22.1 C-style casts

`(T) x`

Do not use them.

22.2 Static and Dynamic Casts

- **`static_cast<T>(x)`**
  - Only checked at compile time
  - Wrong casts lead to undefined results at runtime
  - Examples:

```
Dog* dog = new Dog;

// Does not compile
Cat* cat = static_cast<Dog*>(dog);
```

```
Animal* animal = new Dog();

// Compiles: an animal could be a cat
// Compiler not smart enough remember
// previous statement
// It's still a bad cast and
// will lead to undefined behaviour
// at runtime
Cat* cat = static_cast<Cat*>(animal);
```

- **`dynamic_cast<T>(x)`**
  - Checked at compile and runtime
  - Wrong casts give `nullptr`
  - Is a bit slower due to runtime checks
  - Only works on classes with 1+ virtual member functions
  - Examples:

```
Dog* dog = new Dog;

// Does not compile
Cat* cat = dynamic_cast<Dog*>(dog);
```

```
Animal* animal = new Dog();

// Compiles: an animal could be a cat
Cat* cat = dynamic_cast<Cat*>(animal);

// At runtime, it is discovered that
// animal was not a cat, but a dog
// The bad cast is detected, and
// cat == nullptr
```

- **Smart pointers**
  - We said not to use regular pointers `T*`
  - Use `std::unique_ptr` and `std::shared_ptr` instead
  - Library offers **specialized cast operators for `std::shared_ptr`**
  - No specialized casts for `std::unique_ptr`

```
std::shared_ptr<Shape> shape;

// Equivalent for static_cast
std::shared_ptr<Circle> circle =
  std::static_pointer_cast<Circle>(shape);

// Equivalent for dynamic_cast
std::shared_ptr<Circle> circle =
  std::dynamic_pointer_cast<Circle>(shape);
```

22.3 `reinterpret_cast`

- **Anything goes**
- **No compile time checks**
- **No runtime checks**
- Bad casts lead to **undefined behavior**

```
Cat* cat = new Cat;
Cow* cow = reinterpret_cast<Cow*>(cat);

// Ben Stiller is vindicated
cow->milk();

// Anything goes
int* ns = reinterpret_cast<int*>(cow);
```

- Allows you to **probe memory**
- **Breaks encapsulation**

```
class Person
{
private:
  std::string name;
  int age;

  // Password is secret
  std::string password;

public:
  Person(std::string pw)
    : password(pw) { }
};
```

```
int find_password_index(const Person& person,
                        const std::string& password) {

  // p points to start of person
  const uint8_t* p = reinterpret_cast<const uint8_t*>(&person);

  // Scan person object
  for (int i = 0; i < sizeof(Person); ++i) {
    // __try: Windows-specific, to prevent crashes
    // to due illegal memory accesses
    __try {
      // Pretend p points to string
      const std::string* ps = reinterpret_cast<const std::string*>(p + i);

      // Check if we found the right spot
      if (*ps == password) return i;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
      // Do nothing in case of error
    }
  }
}
```

```
std::string find_password(const Person& person) {
  // First use person with known password
  // to find position of password in object
  Person dummy("xyz");
  int index = find_password_index(dummy, "xyz");

  // Look at same location in given person
  const uint8_t* p = reinterpret_cast<const uint8_t*>(&person);
  return *reinterpret_cast<const std::string*>(p + index);
}

int main() {
    Person person("secret");
    std::cout << find_password(person) << std::endl;
}
```

- **What's it good for?**
  - Example: FAT12

Offset	Length	Description
0x00	3	Instructions to jump to the bootstrap code
0x03	8	Name of the formatting OS
0x0B	2	Bytes per sector
0x0D	1	Sectors per cluster
0x0E	2	Reserved sectors from the start of the volume
0x10	1	Number of FAT copies
0x11	2	Number of possible root entries
0x13	2	Small number of sectors
⋮	⋮	⋮

```
struct FAT12 {
  uint8_t    bootstrap[3];
  uint8_t    os[8];
  uint16_t   bytes_per_sector;
  uint8_t    sectors_per_cluster;
  uint16_t   reserved_sectors;
  uint8_t    number_of_fats;
  uint16_t   number_of_root_entries;
  uint16_t   small_number_of_sectors;
  // ...
};
```

```
std::ifstream input("fat12.img", std::ios::binary);
FAT12 data;

input.read( reinterpret_cast<char*>(&data),
            sizeof(FAT12));

int disk_size = data.bytes_per_sector *
                data.small_number_of_sectors;
```

- If you **read raw bytes** from disk, network,…
- You can **interpret raw data for what they represent** (FAT data tables, IP address, TCP/IP packets,…)

```
byte* data = read_data();

IP_ADDRESS* ip =
  reinterpret_cast<IP_ADRESS*>(data);
```

22.4 const_cast

- Used to **remove `const` qualified**
- Only used for **old C libraries**
- **Don't use this either**

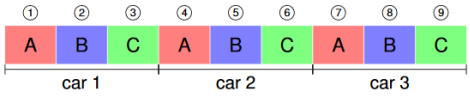
```
// Const-incorrect function
// Does not need write access,
// so parameter should be const
int current_age(Person& person);

void foo(const Person& person) {
  // Wants to get the age
  // Does not compile
  int length = string_length(person);

  // Works
  int length = string_length(const_cast<Person&>(person));
}
```

23. Performance

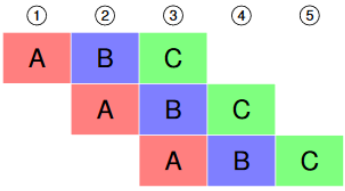
23.1 Branch Prediction

- How does the CPU execute instructions?
  - CPU receives stream of instructions
  - CPU executes these instructions one after the other (that's what it should look like)

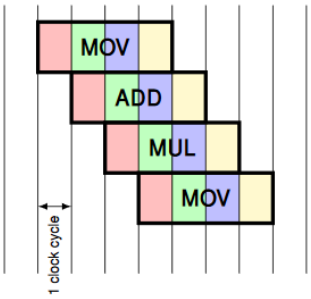
- **Car assembly analogy**
  - Imagine a car assembly line
  - Assembling a car takes many phases
    - E.g. 3 phases named A, B, C
  - Does it make sense to wait for one car to be finished before starting another?
  - Assembly of *n* cars takes 3*n* steps



  - Different phases of different cars can happen in parallel
  - Start with car 2 as soon as car 1 finishes phase A
  - Assembly of *n* cars takes *n* + 2 steps



- **Pipelining**
  - Same approach is used by CPUs
  - Instructions are split up in different stages
  - Different stages of different instructions execute in parallel
  - Allows instructions to **overlap in time**
  - **Instructions are executed in parallel**
  - To maximize efficiency, the pipeline needs to be fed new instructions **at all times**



- **Conditionals**
  - Conditionals cause trouble
  - cond determines which path to take
    - We **cannot add new instructions to pipeline as long as we don't know the value of cond**
    - Pipeline will be completely emptied before a or b executes
  - Really **bad for performance**

```
if ( cond )
    a;
else
    b;
```

- **Branch prediction**
  - Solution: try to **guess which path will be taken**
  - CPU picks path and starts executing as if there was no if
  - If chosen path turns out to be wrong path, CPU needs to **undo everything**
    - Good performance if guess was good
    - Bad performance if guess was bad

- How does CPU maximize good guesses?
- Many different approaches
  - **Static branch prediction**
    - CPU **always guesses the then-branch** will be taken
    - Programmer writes program so that then-branch is more probable

```
for ( int i = 0; i != 100; ++i ) {
    // Bad if: i is more likely
    // not to be prime
    if ( is_prime(i) ) {
        // A
    }
    else {
        // B
    }
}
```

```
for ( int i = 0; i != 100; ++i ) {
    // Switching things around will
    // yield better performance
    if ( !is_prime(i) ) {
        // B
    }
    else {
        // A
    }
}
```

  - **Dynamic branch prediction**
    - CPU **keeps count** at runtime
    - Tries to **detect patterns**

```
for ( int i = 0; i != 100; ++i ) {
    // Bad performance, as there
    // is no clear pattern
    if ( random() % 4 == 0 ) {
        // A
    }
    else {
        // B
    }
}
```

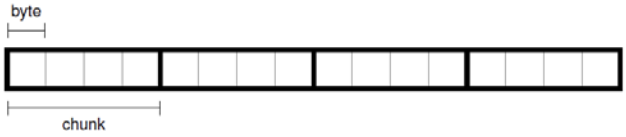
```
for ( int i = 0; i != 100; ++i ) {
    // Good performance, as there is a
    // simple pattern (TFFF TFFF ...)
    if ( i % 4 == 0 ) {
        // A
    }
    else {
        // B
    }
}
```

    - Example benchmark

Pattern	Time
F	5746
FT	6264
FFTT	5893
FTTT	5808
FFFTTTT	8692
FTFTTTTT	7465
FFTTTTTT	7832
FTTTTTTF	7059
FFFFFFFFTTTTTTTT	8355
FTFTFTFTTTTTTTTT	7814
FFTTFFTTTTTTTTTT	7355
FTTTFTTTTTTTTTTT	6931
FFFFTTTTTTTTTTTT	7735

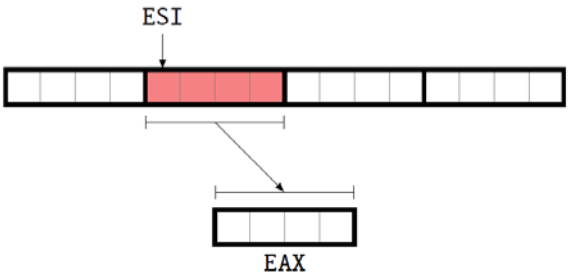
Measurements done on Core i7 6700, 3.4GHz

23.2 Memory Alignment

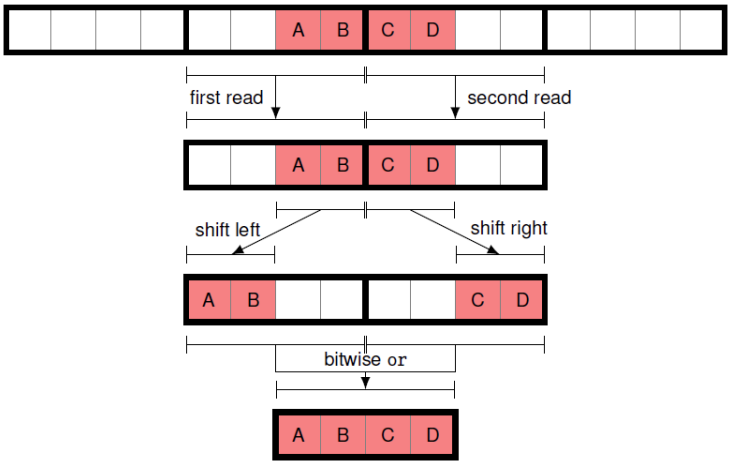
- Reading from RAM
  - RAM is not organized as a series of bytes
  - RAM is organized as a **series of chunks** (e.g. chunk of 4 bytes)
  - **Reading from RAM = reading one complete chunk**
  - 

- **Reading aligned block from RAM**
  - Reading a chunk from RAM to register happens in one step

```
MOV EAX, [ESI]
```



- **Reading unaligned block from RAM**
  - Reading 4 bytes that do not correspond to a chunk (i.e. nonaligned) takes many steps

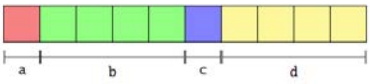


  - If the word you need to read does not fit in one chunk, multiple reads necessary
  - General rule of thumb**: align 2^N sized words on a 2^N-devisible memory address**
    - `uint8_t` (1 byte) can start anywhere
    - `uint16_t` (2 bytes) should start on address divisible by 2
    - `uint32_t` (4 bytes) should start on address divisible by 4
    - `uint64_t` (8 bytes) should start on address divisible by 8
  - This is generally done **automatically by the compiler**
- **Size of data structures**

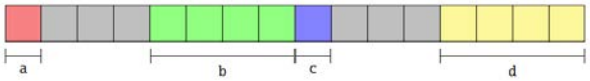
```
struct Foo {
    uint8_t  a;
    uint32_t b;
    uint8_t  c;
    uint32_t d;
};
```

- Should be 1 + 4 + 1 + 4 = 10 bytes large
- `sizeof(Foo)` returns 16
- Compiler leaves "gaps" so as to align `b` and `d` correctly

Without Memory Alignment



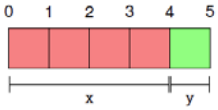
With Memory Alignment



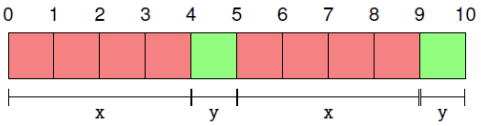
- Grey = unused memory
- IT would be better to rearrange members
  - a, c, b, d would be optimal
  - Compiler will always preserve order
  - You need to rearrange yourself

- **Arrays**
  - We focused on aligning the members of one object
  - What if we consider an array of such objects?
  - **Example:**

```
struct Foo
{
  uint32_t x;
  uint8_t  y;
};
```



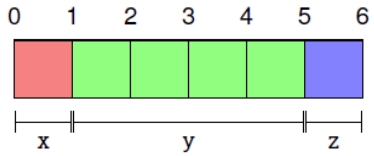
  - Seems OK, members are aligned
  - x is 4 bytes long and starts at 0
  - y is 1 byte long, can start anywhere
  - Array of 2 Foos:



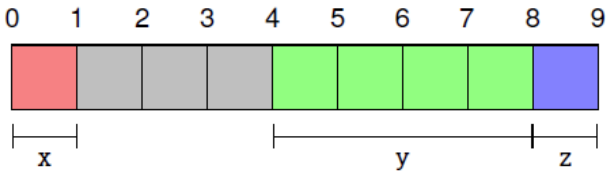
  - Second `foo`'s x starts at address 5 → not aligned correctly
  - To ensure that all members are aligned, even in arrays, **add extra padding at end**
  - General rule: **total size must be a multiple of the largest member**
  - Largest member is `uint32_t` → total size %4 = 0
  - **Example:**

```
struct Foo
{
  uint8_t x;
  uint32_t y;
  uint8_t z;
};
```

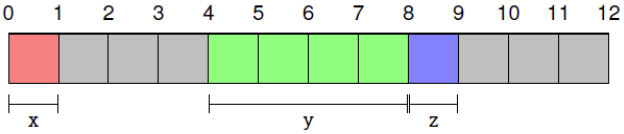
- Unaligned:



- Padding added to align members of single object:



- Adding padding to make total size a multiple of 4

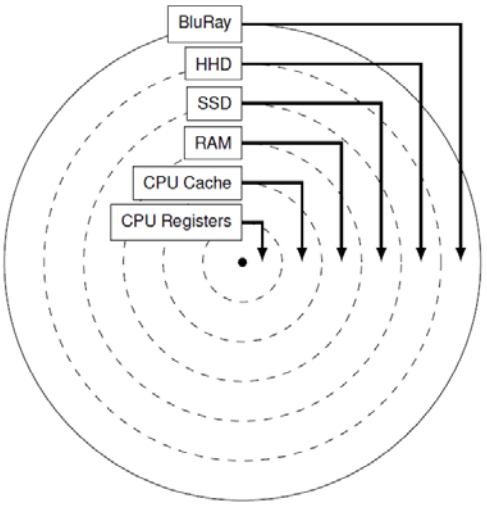


- This is final memory layout: `sizeof(Foo) == 12`
- *Note: it would be better to reorganize the members first*
- Benchmark: reading `uint64_t`

Memory Address	Time
$8k + 0$	2189
$8k + 1$	3506
$8k + 2$	3524
$8k + 3$	3611
$8k + 4$	3605
$8k + 5$	3560
$8k + 6$	3566
$8k + 7$	3547

23.3 Cache

- Cheap storage is often slow
- **Most often used data** on **fast storage** medium
- **Least often used data** on **cheap storage** medium

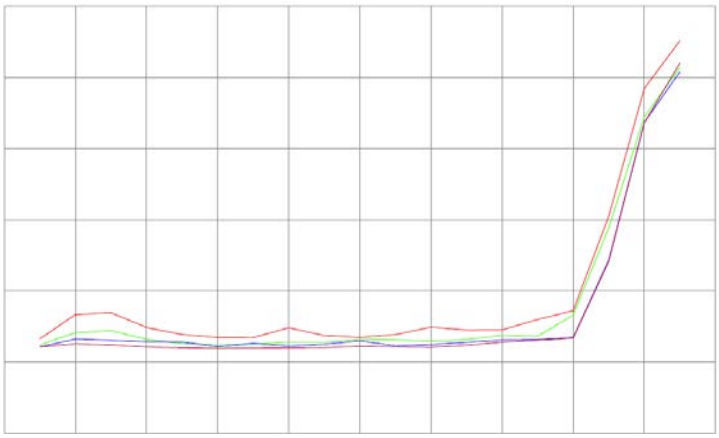


Storage	Speed (GB/s)
BluRay	0.027
HDD	0.15
SSD	3
RAM	30
L1 Cache	1000
CPU Registers	30 000

- **CPU Cache**
  - CPU registers are blazingly fast
  - RAM is relatively slow compared to CPU
  - Cache bridges the speed gap
  - How does it do that?
  - Optimization is always a matter of making assumptions

23.3.1 Temporal Locality

- **Temporal locality**
  - Assumption: code often **accesses same memory locations repeatedly in short time**
  - This idea is known as temporal locality
- **Cache**
  - Simplified explanation, see example slides 33 – 66
  - For cache to work well, it is **important that CPU keeps accessing the same memory locations**
  - If CPU jumps around too much, cache lines will be recycled each time and performance will be as bad as if there is no cache
- **Benchmark**
  - Setup
    - We start with large block of memory
    - We split this memory up in *N* blocks
    - For each block in turn, we access its contents *k* times
  - Expectations
    - First block access will be slowest (not yet in cache)
    - Subsequent access should be fast, if block fits in cache
    - If block too large, even subsequent access should be slow



23.3.2 Special Locality

- **Special locality**
  - Other typical usage pattern
  - CPU often **accesses memory range sequentially**
    - (e.g. accessing all elements in an array one after the other)
  - How to make use of this?
- **Prefetching**
  - If CPU starts accessing memory locations `p`, `p+1`, `p+2`, already ask RAM for `p+10`
  - By the time CPU is done processing `p...p+9`, the rest of the data has already been fetched from RAM

- **Benchmark**
  - Accessing array sequentially: 77ms
  - Accessing array at random indices: 1848ms

23.4 Compiler Optimizations

- Compiler does not translate your code verbatim (word by word)
- Compiler **only promises to generate code that behaves the same as your code**
- Compiler knows CPU better than you (they've been friends since they were young)
- Compiler **will transform your code so that it runs faster**
- Compiler has many tricks up its sleeve

23.4.1 Dead Code Elimination

- Dead code = code that will never be executed
- **Compiler removes dead code**
- Code below gets compiled to nothing

```
int long_computation(int x) {
    int result = 0;

    while ( x-- )
        result += x;

    return result;
}

// Nothing is done with result
long_computation(9999999);
```

23.4.2 Inlining

- Calling a function incurs some overhead
  - Save important register values
  - Prepare parameters
  - CALL
  - Execute body
  - Prepare return value
  - RET
  - Restore important register values
- This **overhead should remain small compared to time needed for actual body**
- This isn't the case with small functions (e.g. getters)
- Inlining copies the body of the *callee* into the caller's body:

```
void inc(int& x) {
    x++;
}

int x = 0;
inc(x);

// becomes

int x = 0;
x++;
```

- Inlinging is **good for short functions**
- **Could be bad for cache**
  - Cache works best if same code is executed often
  - Inlining makes copies, each called once
- Compiler can be smart enough to dispense with pointers/references
- Example:

```
void swap(int& x, int& y) {
    int temp = x; x = y; y = temp;
}

int x = 1, y = 2;
swap(x, y);
```

  - Is compiled to:

```
mov eax,ebx
mov ebx,edi
mov edi,eax
```

  - Notice lack of CALL/RET
  - Notice lack of pointer

23.4.3 Zero Cost Abstractions

- Similarly to inlining, **objects can be compiled away**
- If a `Foo` contains only a `double`, working with a `Foo` is as fast as working with a `double`
- There is no inherent overhead to objects
- **Example: angles**
  - Angles can be represented by `double`s
  - When given a `double`, is an angle expressed in radians or degrees?
  - No way of knowing
  - Introduce helper class Angle:

```
struct deg { };
struct rad { };

template<typename UNIT>
class angle {
    explicit angle(double value)
        : value(value) { }

    double value;
};

template<typename T>
angle<T> operator +(angle<T> a, angle<T> b)
{
    return angle<T>(a.size + b.size);
}
```

  - Type contains the unit (`rad`/`deg`)
  - No mistakes possible, compiler will catch them
  - Is just as fast as working with straight doubles

```
angle<rad> a = 1.2;
angle<deg> b = 180;

angle<deg> c = b + b; // ok
angle<rad> d = a + b; // compile error
```

23.4.4 Compile Time Execution

- Compiler can choose to execute code itself
- The resulting executable then does **not contain the computation**, but **only the result**
- **Example 1:**

```
int sum(int from, int to) {
  int result = 0;

  for (int i = from; i <= to; ++i)
      result += i;

  return result;
}

int result = sum(0, 5);
```

gets compiled to

```
mov  edx,0Fh
```

- Example 2:

```
class Predicate {
public:
    virtual bool check(int x) const = 0;
};

int count(int max,
          const Predicate<int>* predicate) {
    int result = 0;

    for (int i = 0; i != max; ++i)
        if (predicate->check(i)) ++result;

    return result;
}
```

  - Using inheritance (à la Java)
  - Time used: 2020ms

```
template<typename Predicate>
int count(int max)
{
    int result = 0;
    Predicate p;

    for (int i = 0; i != max; ++i)
        if (p(i)) ++result;

    return result;
}
```

  - Using templates forces things to be compile time
  - Time used: 571ms (~4 times faster)

23.4.5 Tail Call Optimization

- Recursive functions call themselves
  - Involves CALL and RET
  - Involves PUSH and POP
- Are iterative approaches (loops) more efficient?
  - Reuse of same local variables instead of always new ones
  - No CALL or RET needed

- **Tail call optimization**

```
void rec(int x)
{
  int a, b, c; // locals
  ...
  return rec(x - 1);
}
```

  - Tail call = returning result of recursive call without any postprocessing
    - `return rec()` is ok
    - `return 1 + rec()` is not ok
  - **Multiple `return`s allowed**, as long as they are all tail calls
  - Recursive call can then **recycle same locals**
    - No new stack frame needed for recursive call
    - Just reuse current one
  - **Tail call optimization transforms recursion to loop**
- Example:
  - Recursive algorithm

```
struct linked_list_node
{
    linked_list_node* next;
};

const linked_list_node*
  find_last(const linked_list_node* p)
{
    if (p->next == nullptr) return p;
    else return find_last(p->next);
}
```

  - **Generated assembly is iterative**

```
  mov  rax,qword ptr [rcx]      ; Fetch next
  test rax,rax                  ; Check for null
  je   exit                     ; Return
  nop  dword ptr [rax+rax]

loop:
  mov  rcx,rax
  mov  rax,qword ptr [rax]      ; Fetch next
  test rax,rax                  ; Check for null
  jne  loop                     ; Loop if not null

exit:
  mov  rax,rcx                  ; Return value
  ret
```

23.6 Loop Optimizations

- **Loop optimization**
  - Loops can be rewritten in many ways
  - **Loops are perfect candidates for optimization**
    - Most of execution time is spent in loops
  - Loops can be rewritten so as to
    - **Reduce loop overhead**
    - **Improve locality**

- **Loop unrolling**

```
// Before
for ( int i = 0; i != imax; ++i )
{
    foo(i);
}

// After
for ( int i = 0; i != imax; i += 4 )
{
    foo(i);
    foo(i+1);
    foo(i+2);
    foo(i+3);
}
```

  - Reduces amount of loop condition checks
- **Loop fission**

```
std::vector<int> xs, ys; // same size

// Before
for ( int i = 0; i != xs.size(); ++i ) {
    process(xs[i]);
    process(ys[i]);
}

// After
for ( int i = 0; i != xs.size(); ++i )
    process(xs[i]);

for ( int i = 0; i != xs.size(); ++i )
    process(ys[i]);
```

  - Before: CPU hops back and forth in memory
  - After: goes linearly over elements → **better for cache**
- **Loop interchange**

```
// Before
for ( int x = 0; x != width; ++x )
    for ( int y = 0; y != height; ++y )
        process( xs[y][x] );

// After
for ( int y = 0; y != height; ++y )
    for ( int x = 0; x != width; ++x )
        process( xs[y][x] );
```

  - Before: CPU hops around a lot
  - After: CPU processes **each array sequentially**
- **Strength reduction**

```
// Before
for ( int i = 0; i != imax; ++i )
{
    int k = i * 10;
    // ...
}

// After
int k = 0;
for ( int i = 0; i != imax; ++i )
{
    k += 10;
    // ...
}
```

  - Rewriting so as to **use less expensive operations**

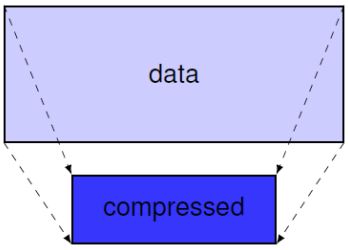
24. Technical details

See slides.

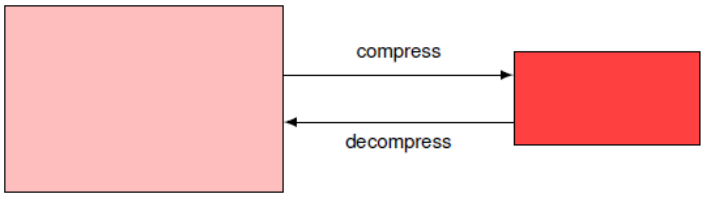
25. Compression

25.1 Intro

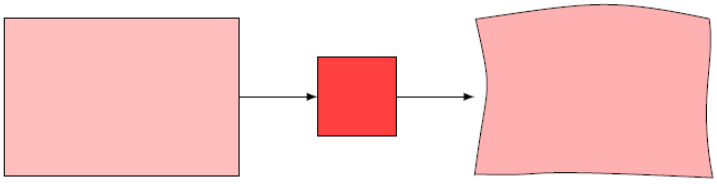
- **Encoding same information using less space**



- **Example: movie compression**
  - **Audio**
    - Dolby TrueHD
    - 6 channels, 192 kHz, 24 bit
    - Bitrate: 6 x 192.000 x 24 = 27.648.000 bps
  - **Video**
    - Blu-Ray HD video format
    - Resolution 1920 x 1080, 24fps, 24 bit colors
    - Bitrate: 1920 x 1080 x 24 x 24 = 1.194.393.600 bps
  - **Total**
    - 1.222.041.600 bps ≈ 146 MiB/s
  - **Streaming**
    - Streaming uncompressed requires 1222 Mbps connection…
    - Best average connection speed: South Korea with 26.7 Mbps
  - **Optical Disc**
    - PS4 reads discs at 27 MiB/s
    - Size of 120' movie: 8194 GiB
    - Industry standard: 50GiB for dual layer disc
- **Types of compression**
  - **Lossless**



    - Compressing then decompressing **yields the original data**
    - **No information is lost**
    - Examples: zip, 7z, rar, png
  - **Lossy**



    - Part of the **information is lost**
    - Generally used with **sound and images**
    - Examples: jpeg, mp3, DivX,…

- **Example: MP3**
  - MP3 takes into account the **human ear** (psychoacoustics)
  - Uses **several techniques**
    - **Minimal audition threshold**
      - Our perception of loudness depends on the pitch
      - Lower sounds need more energy to appear loud
      - MP3 removes sounds that are below a certain loudness
    - **Masking effect**
      - Loud noises mask soft noises
      - Soft noises can be removed
      - Forward masking: sounds can mask future sounds
      - Backwards masking: later sounds can make earlier sounds inaudible
      - Sounds in the same critical bandwidth are merged into single sound
    - **Joint stereo encoding**
      - **Locating sounds**
        - Brain uses differences in L/R to locate sounds
        - Human ear fails to locate very low and very high sounds
        - These sounds can be stored in mono (less data)
      - **Similarities between channels**
        - L/R channels are often similar
        - **Mid/Side instead of Left/Right**
        - Mid contains **sounds common** to L and R
        - Side contains **differences between** L and R
    - **Bytes reservoir**
      - Some parts are easy to encode (e.g. silent parts)
      - Some parts require more detail (e.g. many instruments)
      - Spend more bits on detailed parts
    - **Final encoding to bits**
      - Huffman encoding (see 27)

25.2 Quantising Information

- Example slides 3 – 18
- **Bits vs. Bits**
  - Physical bits: if you have 8 GiB storage, you have $2^{2^{36}}$ codes available
  - Distance can be measured in meters, time in seconds, weight in kilogram,…
  - **Information can also be quantified**
  - **The unit for information is "bit"**
  - To distinguish from the encoding bit, we'll use "**ibit**"
  - **Formula** to compute the **number of ibits**:

$$\log_2(number\ of\ outcomes)$$

- **Compression**
  - If you have **N ibits** of information, you will minimally need **N bits** of storage
  - Compression = **only using N bits of storage**
  - **Example**
    - Five exams
    - We want to know the exact grade

- 0, 1, 2, …, 19, 20 → **21 possible grades**
- $\log_2(21^5) \approx 21.96\ ibits$
- You'd probably use an `int` array: $5 * 32 = 160\ bits$
- In reality, you only need *22 bits*

25.3 Limitations of Compression

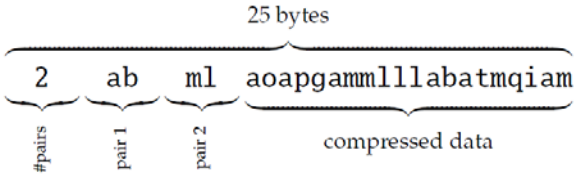
- **Infinite compression**
  - Can you continue compressing a file (can you zip a zip)?
  - **Lossless : ibits form lower bound**
  - **Lossy**: **no lower bound**, but compressing every picture to a black image (only 0 bits needed) is not useful
  - **Infinite compression is impossible**
- **Optimal Compression**
  - Math tells us what the best possible compression is
  - But it doesn't tell us how to achieve it
  - Best compression requires knowledge about the data
  - Example:
    - File with grades (0-20), you use one byte per grade
    - Byte has 8 bits, hence 256 codes
    - Grades: 21 possibilities → redundancy
    - But if you want to zip this file: zip does not know there are grades
    - Zip must assume all byte values are possible
    - How can zip compress data?
    - Zip does not know about grades, it will never be able to compress optimally
    - Zip *does* manage to compress data without seemingly making assumptions
    - We will discuss a general-purpose compression algorithm in 25.4
    - MP3 is opposite of zip: it knows the data represents sounds, and makes all kinds of assumptions about it (psychoacoustics) to better compress it

25.4 QU Compression Algorithm

- **Example: slide 33 – 51: QU compression**
- GQU: Generalized QU compression (see example)
- Given a **random piece of data**, we can **analyze if certain pairs of bytes occur often**
- We can keep a **table of "interesting pairs"** for which we apply our trick of omitting the second member of the pair
- Example
  - **Input:**
    - aboabpgabmlmllaabtmlqiabml
    - 26 bytes
  - **Pair frequencies**

Pair	Frequency	Pair	Frequency
ab	**5**	bo	1
oa	1	bp	1
pg	1	ga	1
bm	2	**ml**	**4**
lm	1	ll	1
la	1	aa	1
bt	1	tm	1
lq	1	qi	1
ia	1		

- **Compression algorithm**
  - Say we found *n* frequent pairs $\alpha_i\beta_i$
  - In example:
    - $n = 2$
    - $\alpha_1 = a$
    - $\beta_1 = b$
    - $\alpha_2 = m$
    - $\beta_2 = l$
  - We need to store these pairs for decompression
  - Encoding data:
    - $\alpha\beta\beta \rightarrow \alpha\beta\beta\beta$
    - $\alpha\beta \rightarrow \alpha$
    - $\alpha \rightarrow \alpha\beta$
- **Compressed data**:



  - 25 bytes
- **Gains**
  - One byte
  - Would work better on larger inputs
  - Some losses in the header (5 bytes)
  - Header = price for generalizing

25.5 Summary

- **Compression makes assumptions**
  - In order to compress data, we **need to make assumptions about it**
  - QU compression: assumes qu appears often
  - GQU compression: assumes certain pairs occur often
  - MP3: assumes it's audio meant for human ears
- **No compression is perfect**
  - A compression algorithm **will decrease the size of input that satisfies the assumptions**
  - But it will **increase the size of input that does not satisfy the assumptions**
  - **No compression algorithm can compress *all* inputs to something smaller**

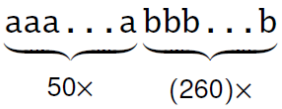
26. Compression: RLE

26.1 Run Length Encoding

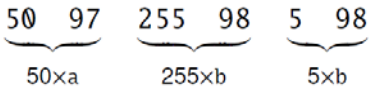
- **Run Length Encoding** is a very simple compression algorithm
- In order to compress, it must make some assumptions about input data
- RLE assumes **that long runs of the same bytes occur often**
- RLE is used in some **image formats**: drawn images do have long stretches of the same color (TGA, PCX, fax machines)
- **Example**
  - `aaaaabbbbccccccccaaaaaaaaddd` → `5a4b8c8a3d`
- **Example**
  - Text is encoded using ASCII/Unicode
  - `abcd` is represented by the bytes `97 98 99 100`
  - The decompression algorithm will interpret this as
    - Repeat `b` 97 times
    - Repeat `d` 99 times
  - Decompressor **cannot make difference** between
    - Byte representing how many times next byte needs to be repeated
    - Byte representing actual data
  - `abcd` needs to be compressed to
    - **`1a1b1c1d`**
    - **`01 61 01 62 01 63 01 64`** (hex)
- **Shortcomings**
  - It can compress runs of bytes very well, but fails on solitary bytes

26.2 Variations

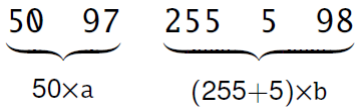
- Many **variations possible**
- **Example: how to encode *n***
  - Currently: single byte
  - Problems:
    - *n* = 0 is useless
    - *n* can't be greater than 255: longer series to be split up
- **Dealing with n = 0**
  - Make *n* = 0 useful
  - Interpret *n* = 0 as *n* = 256, so that you can support runs of 256 long
  - Or *n* = 0 means the following two bytes are solitary
- **Dealing with *n*'s maximum value**
  - We could use more bytes for *n*
    - 1 byte: *n* can go up to 255
    - 2 bytes: *n* can go up to 65535
    - 4 bytes: *n* can go up to $2^{32} - 1$
  - Is this interesting?
    - 1 byte good for when there are many short runs
    - 4 bytes good for when there are at least a few *very* long runs
  - **Variable length *n***
    - *n* = 255 could mean that the next byte should be interpreted as a count-byte and added to the current *n*
    - For example, consider text:



- In our original encoding, this would become (in decimal):



- In our variable length *n* encoding, we would get:



- **MIDI and UT_8**
  - Support integers of arbitrary length
  - Bits of the integers are split in groups of 7 bits
  - Each encoded byte contains a first group of 7 bit
  - The eight bit is used to indicate whether the following byte also belongs to the same integer (eight bit is "first" bit, read from left to right)
  - Example: encoding 100
    - 100 in binary: `1100100`
    - 7 bits needed
    - Only one byte needed: **`0`**`1100100`
    - `0` means "integer ends here"
  - Example: encoding 1000
    - 1000 in binary: `11 11101000`
    - 10 bits needed
    - Two bytes needed: **`1`**`0000111` **`0`**`1101000`
    - `1` means "integer continues in next byte"

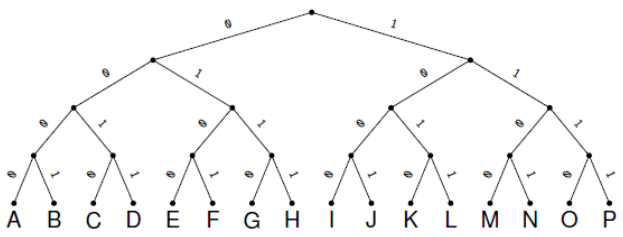
27. Compression: Huffman

Most examples assume we deal with text, but the principles apply to bytes in general.

- Letter frequency in "A Game of Thrones"

Letter	Frequency	Letter	Frequency
E	154 148	M	29 129
T	101 753	G	27 193
A	98 313	Y	25 888
O	92 254	F	25 074
H	86 942	C	22 459
N	81 018	B	20 154
S	80 389	P	14 831
R	78 068	K	14 469
I	71 945	V	9068
D	65 816	J	2513
L	53 882	Q	1050
W	32 485	Z	599
U	29 804	X	591

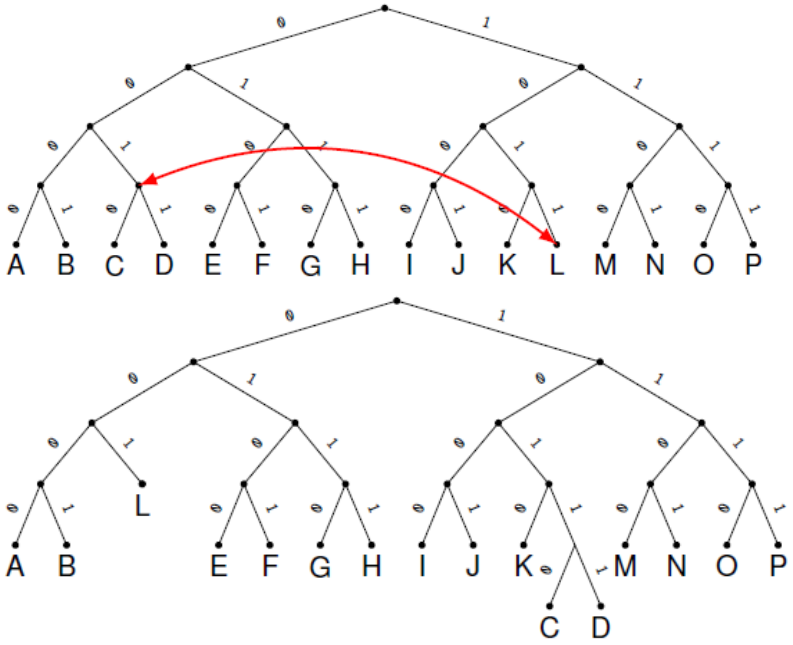
- **Bits vs Letter frequency**
  - Each letter is represented by a unique 8 bit code (ASCII)
  - The letter E is used 261 times more often than X
  - It would make more sense to use less bits for E than for X
- **Using less bits**
  - Say we use 4 bits for E: `0101`
  - EE corresponds to `01010101`, but so does U
  - In order to encode E with less bits, we need to change more than just E's code
- **How to build an unambiguous encoding**
  - Visualizing helps
  - First, we represent our codes by a tree
  - Next, we find out what operations we can apply on a tree
  - We use 4 bits because an 8-bit tree is dense to visualize



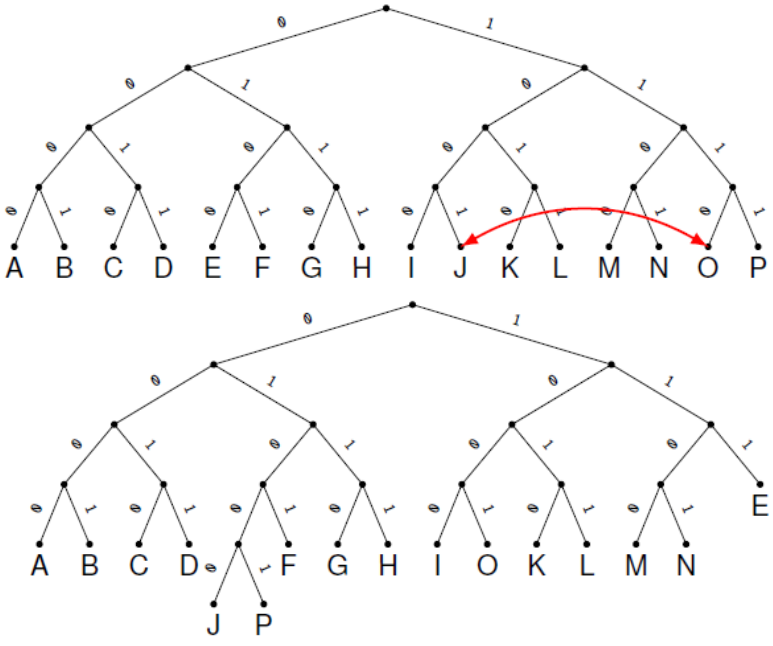
Letter	Code	Letter	Code	Letter	Code	Letter	Code
A	0000	E	0100	I	1000	M	1100
B	0001	F	0101	J	1001	N	1101
C	0010	G	0110	K	1010	O	1110
D	0011	H	0111	L	1011	P	1111

- **Modifying the tree**
  - How can we safely modify this tree?
  - The actual codes do not matter (whether A is `0000` or `1111` is not important)
  - What matters to us is
    - **the length of each code**
    - **the unambiguity of each code**

- **Swapping nodes**

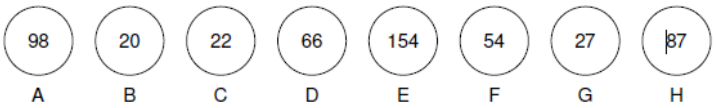


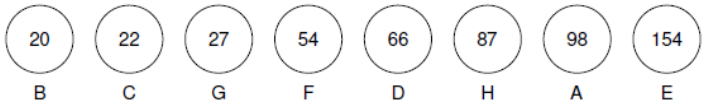
  - We can swap any two nodes in the tree
  - The resulting code remains unambiguous
  - The codes have changed
    - L's code became one bit shorter
    - C and D's codes became 1 bit longer
    - Gain 1 bit but lose 2: typical for compression
  - Goal of swapping:
    - Make frequent letter have shorter codes
    - Sacrifice rarely occurring letters
- **Optimal tree**
  - E occurs most often
  - J and P appear least often
  - We can steal from J ad P and give to E

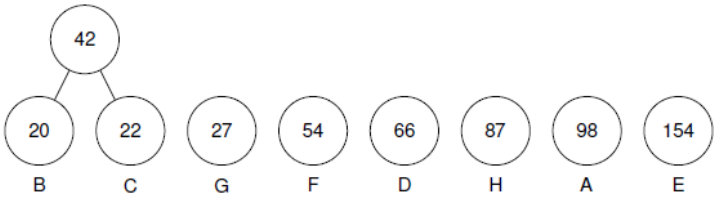


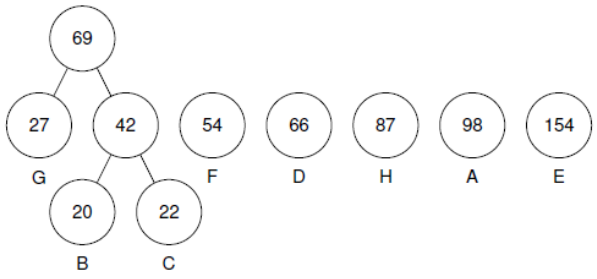
        - We want to make E shorter and P,J longer
        - We need to put P and J together
        - We can swap E with P,J
        - New codes: E=111, J=01000, P=01001
    - Is this new tree optimal?
    - We could make E two bits long by sacrificing other letters
    - We should perhaps also consider making T and A shorter
    - In order to find the best, we need a way to compare trees
    - Can we associate a metric with each tree?
- **"Measuring" tree**
    - Given a tree, we know how many bits are required for each byte
    - We can compute how long the compressed file would be
    - This is what we want to minimize
    - Approach to finding optimal tree:
        - Consider all trees
        - Picking the one with the smallest file size
    - Hardly efficient
    - Luckily, Huffman developed algorithm to find optimal tree
- **Building a Huffman tree**
    - Create a leaf node for each byte
    - Each node has a weight
    - The weight of a leaf node equals the frequency of the byte
    - **Repeat until one node left:**
        - Take the two nodes with the least weight
        - Join them in a new node
        - This new node's weight equals the sum of its children
- Example:

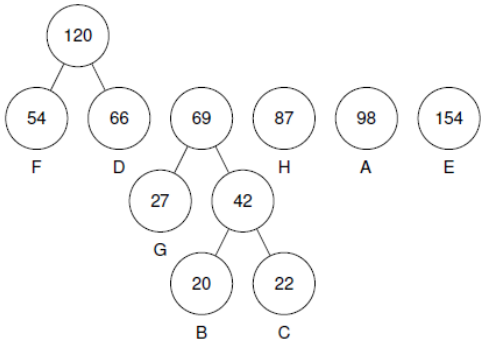
Letter	#	Letter	#
A	98	E	154
B	20	F	54
C	22	G	27
D	66	H	87

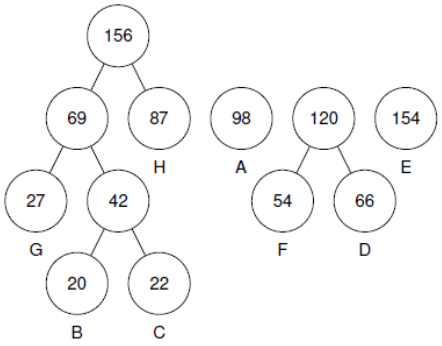
    - 

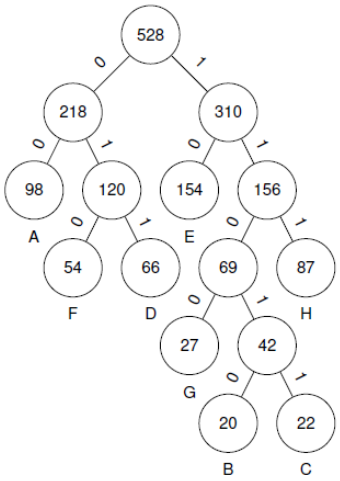
    - 

    - 

- 

- 

- 

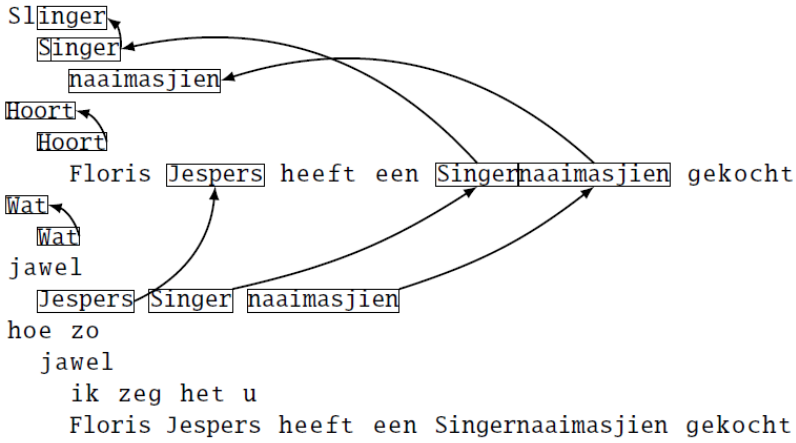
- 

Letter	Code
A	00
B	11010
C	11011
D	011
E	10
F	010
G	1100
H	111

- **Huffman compression**
  1. Count the frequency of all bytes
  2. Build Huffman tree
  3. Derive code table (byte to code)
  4. Write Huffman tree to file (decompressor needs this information)
  5. Write encoded bytes to file
  6. It Is possible to omit Huffman tree in the file by saving the information in a more efficient form, from which we can derive the tree later

28. Compression: LZ77

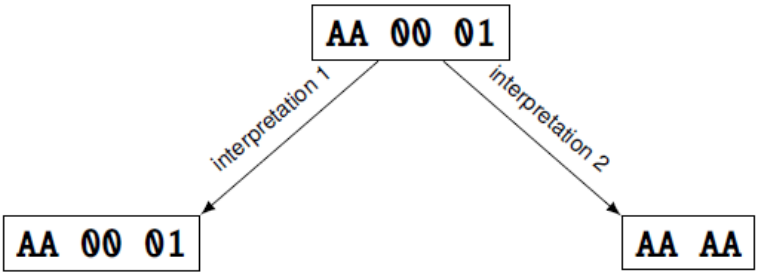
- Example input:



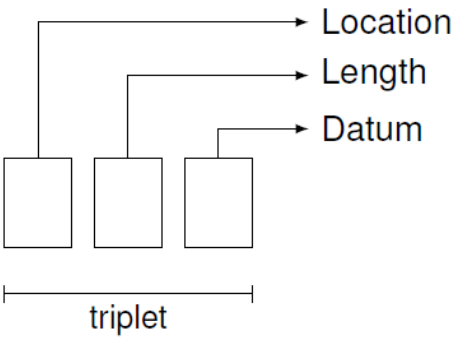
- **Observations**
  - Same words are repeated often
  - Happens often in certain data files (source code, text files, xml,…)
  - More generally: **certain series of bytes occur repeatedly**
- **Basic principle of LZ77**
  - If same word has **appeared before**
    - don't repeat it
    - but **refer to the previous occurrence**
- **Conceptual example**
  - Data to be compressed:
    `muxmuxbarbarmuxbarbarmuxmuxbarbarmuxbarbar`
  - Compressed form
    1. `mux`
    2. Repeat 3 letters starting from first letter
    3. `bar`
    4. Repeat 3 letters starting from 6th letter
    5. Repeat 6 letters starting from 3rd letter
    6. Repeat 21 letters starting from first letter
- **Technical difficulties**
  - Conceptually, LZ works by referring to past data
  - We still need to find a way to encode this into bytes
  - Need to make sure a decompressor can **unambiguously reconstruct** the original data
- **First attempt**
  - A back reference consists of two components
    - Start location of data to be repeated
    - How many bytes to repeat
  - Let's use a single byte for each part of the back reference:

`mux`	→ **6D 75 78**
Repeat 3 letters starting from first letter	→ **00 03**
`bar`	→ **62 61 72**
Repeat 3 letters starting from 6th letter	→ **05 03**
Repeat 6 letters starting from 3rd letter	→ **02 06**
Repeat 21 letters starting from first letter	→ **00 15**

- **Ambiguity problem**

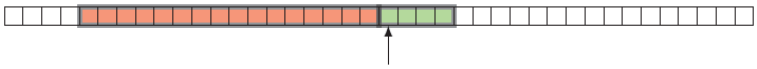


  - How to differentiate between
    - A literal byte
    - A reference to previous data
  - Interpretation 1: `00 01` is data
  - Interpretation 2: `00 01` is a reference to past data (start at index 0, length 1)
- **Ambiguity solution**
  - Many solutions possible
  - LZ77 imposes uniformity
  - LZ77 does not discern between literal bytes and back reference bytes
  - In LZ77, everything is a back reference
  - This of course leads to a new problem: how to reference to data that has never been encountered before?
- **LZ77's approach**



  - LZ77 sees data as series of triplets
    - Location of repeated data expressed in "how many bytes ago"
    - Length of repeated data
    - Single extra datum
  - If x is new byte → (0, 0, x)
  - **Example:** slides 32 – 51
- **Sliding window**
  - Finding longest match repeatedly is slow
  - Data structures can help (e.g. hash tables)
  - Would require a **lot of RAM** for long inputs
  - LZ77 uses a "sliding window"
  - Matches are only looked for inside this window (i.e. the last *N* bytes)
  - Pro's & con's
    - Less memory needed
    - Faster
    - Weakens compression
    - Its size can be chosen freely
    - Size determines how many bits needed for relative location

- **Lookahead buffer**
  - Triplet has “length of repeated data” component
  - Its number of bits limits the length of repeated data
  - Example
    - 4 bits for relative location → $2^4 = 16\ bytes$ sliding window
    - 2 bits for size → $2^2 = 4\ bytes$ lookahead buffer



- **Referencing the future**
  - The repetition can also include characters “from the future”
  - This can be decompressed on condition that **at least 1 character from the past is referenced**
  - See slides 134 – 138 & 141 - 147

29. Recap compression algorithms

- **GQU**
  - Example algorithm
  - Assumes pairs of bytes occur often
- **RLE**
  - Assumes long runs of the same value occur often
  - TGA, PCX, fax machines
- **Huffman**
  - Assumes certain bytes occur more often
  - Widely used (MP3, JPEG, zip, web,…)
- **LZ77/LZ78/LZSS/LZMA/…**
  - Assumes same series of bytes occur repeatedly
  - Widely used (zip, 7z, rar, png, …)