VE280 Recitation Class Notes

YAO Yue

VE280 SU17 TA Group

December 25, 2022

Source code available at https://github.com/tripack45/VE280-Notes

Table Of Contents I

Table Of Contents II

RC Week 2

OK, VE280. What is this course?

How to write good code and linked lists

Writing Quality Code

Taste What kind of code are considered "good"?

Motivation What benefit would such code give us?

Techique What is the recipe for such code?

Tools What language features does C++ provide for achieving this goal? How to use them?

Good (Bad?) News: Exams will (mostly) test for the last point.

Elementary Data Structure

Singly linked lists, Doubly linked lists, Circular Arrays ...

OK, "Quality Code"?

It's very hard to give a definition.

Good Code

- Good variable names
- Consistent indentation
- Well tested, documented
- D-R-Y, Don't repeat yourself
- High Coherence / Low coupling
- Open for extension, but closed for modification

Bad Code

- Bad Style
- 200+ lines in a one function
- Functions of 20+ Args
- Magic numbers everywhere
- Seriously you know bad code when you see one (write one).

Many of them are related to *Abstraction*.

Head First *Abstraction*

No abstraction

The Requirement

CubedEnix (CE) is trying to develop a third person shooting game called *World of Armored Blizzard* (WOAB). In this game, AI will control a tank that can fire upon the player. A programmer *Archer* is asked to implement this feature.

The Solution

Archer propose to have a global variable `TANK` of class `Tank` to represent the tank. Archer than writes the following code.

```
...
Player target = TANK.aim(); TANK.fire(target);
...
```

Archer feels happy, so does his boss.

Head First *Abstraction*

No abstraction

Change of Requirement

Players are complaining that the game is too easy. WOAB dev team decided to add 2 more AI controlled tanks. Again Archer is asked to implement it.

The Solution

Archer decided to use 3 global variables `TANK0`, `TANK1`, `TANK2` of class `Tank`. He copies the original code into 3 different places and modify each of them.

```
... Player target = TANK0.aim(); TANK0.fire(target);
... Player target = TANK1.aim(); TANK1.fire(target);
... Player target = TANK2.aim(); TANK2.fire(target);
```

Archer feels happy, so does his boss.

Head First *Abstraction*

No abstraction

Change of Requirement

Within the next 2 months they increased number of tanks to 10. 1 year later, they (finally!) decided to play a sound effect when a tank fires.

The Twist

Archer now has 10 copies! He needs to add a function call `playSound()` to each copy. Unfortunately he missed one location. Archer also doesn't test his code. Now Buggy code is released. Players are unhappy. His boss is unhappy. Archer is fired. Archer is unhappy. Lancer takes his job.

Head First *Abstraction*

With Procedual Abstraction

The Solution

Lancer decided to write a function that takes a `Tank` object as an argument.

```
// Argument: A non-null pointer to a Tank object.
// Effect  : Fires such tank on the current player.
void tankFire(Tank* t) {
    playSound();
    Player target = t->takeAim();
    t->fire(target);
}
```

With this new design Lancer can easily change the number of `TANK`s in the game, or modify the how tanks fire by simply changing this single function.

Head First *Abstraction*

With just procedual abstraction

Change of Requirement

You know what? Those player just can't be satisfied. WOAB dev team now decides to involve not just tanks but also battle ships and planes (and witches and dragons ...). Lancer is asked to implement "fire" feature for all of them.

The Twist

Lancer is now in a tough situation. Clearly Tanks, Ships, Planes... are different things. Their attack would have different effects. There won't be a single function that works for all of them.
Since Lancer didn't take VE280 seriously when he is in JI, Lancer falls back to copying the code for each "character" once. After 2 months, the code is no longer maintainable, and thus he is fired. Saber takes his job.

Head First *Abstraction*

With Data Abstraction

The Solution

The key is to see the elephant in the room. What's the common characteristic of tanks, ships, dragons and witches? They all attack our player!

```
class IAttackPlayer {
    virtual void fire(Player p) = 0;
    virtual Player takeAim() = 0;
}
class Tank : public IAttackPlayer;
class Ship : public IAttackPlayer;
....
```

Head First *Abstraction*

With Data Abstraction

The Solution

Then we can modify the function:

```
// Argument: a object implements IAttackPlayer.
// Effect  : Fires such tank on the current player.
void NpcFire(IAttackPlayer* t) {
    playSound();
    Player target = t->takeAim();
    t->fire(target);
}
```

In the above design Saber abstracted out what the thing "physically" is. She defines things by defining what it can do.

A few more words

It's okay if You don't fully understand above story

That's gives you a good reason to learn it well.

Can you talk something about projects?

Well, projects are mostly very direct. Be careful in the process and they should be pretty easy. Reserve around 5 days for them. You only have a limited number of trials each day.
Remember it's easy to simply finish them. But to finish them elegantly trust me when I say it's hard.
Yes, it's hard. But the outcome worth every minute spent.

RC Week 2

Your Linux Operating System

Linux as a operating system *kernel*

Operating systems as a resource manager

Operating systems manages your hardware. These includes computation resource (CPU, GPU), storage (Hard drive), communication resource (network)...

The core of OS, a.k.a. *kernel*

At the very heart of the OS there lies the kernel. This piece of software talks directly to the hardware. It provides a unified way of accessing your storage devices (Filesystems), graphic devices, network...

"Linux" is first the name of the kernel

This implies you can actually change your desktop environment: fancier? go with Unity/KDE/Gnome. Light weight & fast? go with LXDE or XFCE.

Flavors of Linux: *Distributions*

Common Choices

This flexibility of choice allows to tailor the system towards our own need. Some commonly accepted configurations become "Distributions".

Ubuntu Series

Maintained by Canonical, it's actually a series of distributions with different choice of desktop environment. Ubuntu / *Unity*. Kubuntu / *KDE*. Lubuntu / *LXDE*. Xubuntu / *XFCE*. Use *apt*.

Debian

Once the most widely use distribution, the father of Ubuntu [that's right :=)]. Package management system is *apt*

Fedora and RedHat Linux Enterprise

Maintained by Red Hat. Fedora is new and cutting edge. RedHat now focus on enterprises. Package Management is *yum*.

Flavors of Linux: *Distributions*

Uncommon choices

Open SUSE

Maintained by german company SUSE. Focus on stability.

Arch Linux

ROLLING UPDATE! Absolutely cutting edge. Great community and wiki. User is assumed to be experienced. Installation is done on command line. Package management system is *pacman*.

Linux From Scratch

Try this if you are absolutely interested in figuring out how everything works.

Gentoo

Think LFS is crazy enough? Well this things compiles every piece of software you use FROM THE SOURCE!

Bash on Windows Subsystem for Linux

Think big, with abstraction

For Linux programs, technically, you don't need an "real" linux kernel to run them. As long as you have some thing that looks like a real one (one that follows the same abstraction).

Windows Subsystem for Linux

This year Microsoft actually release one. Installation guide is available here(click me).
This solution is easy to use, clean. It saves you time copying files back and forth from the virtual machine. You can simply develop in Windows using your favorite tool (IDE and test it easily under "Linux". It's a good solution for the purpose of this course.

Aspects of Linux: Users

MultiUser	Multiple person can operate the system at once.
Group	Each User belongs to one or more groups
Home Dir	Each user has his/her own directory under `/home`
~/	Shorthand for home directory. This one is user specific.
Premission	Owner/Group/Other Read/Write/Execute Flags
root	A special super user `root` overrides all security
su	Command that starts a temporary shell as `root`
sudo	A command that temporarily grant superuser privilege to current user. Namely, "execute the following command as 'root'".

The Miranda warning of `sudo`:

```
We trust you have received the usual lecture from the local
System Administrator. It usually boils down to these three things:
        #1) Respect the privacy of others.
        #2) Think before you type.
        #3) With great power comes great responsibility.
root's password:
```

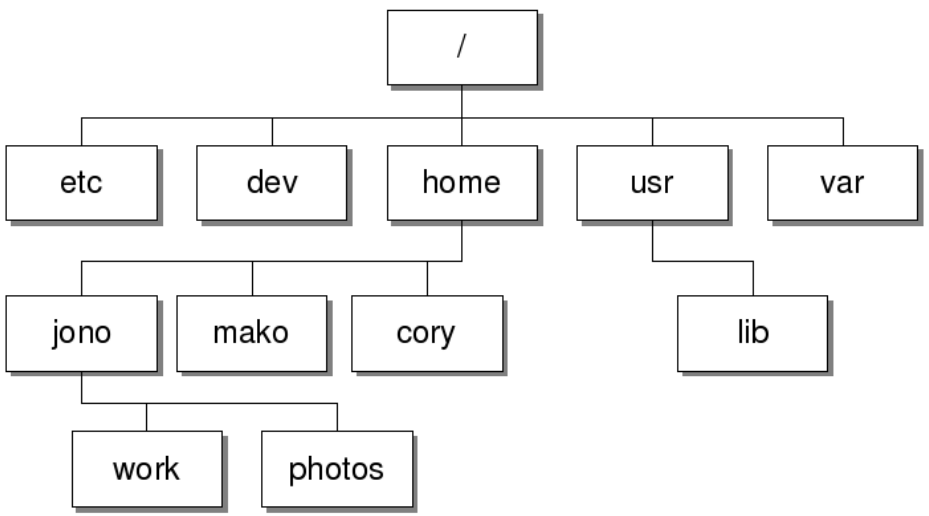
Aspects of Linux: Filesystem

Filesystem is an abstraction of storage

That's right, abstraction again. Filesystem hides the physical storage schema of your files. Instead it should reflect how your files are logically organized.

Linux Filesystem

In Linux files are organized in a tree-like structure



Aspects of Linux: Filesystem

Caveats for the names

A word from Ken Thompson

Ken Thompson was once asked what he would do differently if he were redesigning the UNIX system. His reply: "I'd spell creat with an e."

/bin	**Bin**aries	/mnt	**M**ou**nt**ed filesystem
/dev	**Dev**ices	/etc	Configuration files
/boot	**Boot**strap	/usr	**U**nix **s**ystem **r**esources
/sbin	**S**ecure **bin**aries	/home	Places for the user's files

More at https://wiki.debian.org/FilesystemHierarchyStandar

Aspects of Linux: Shell

CLI and Shell

Command **L**ine **I**nterface is fast, consumes minimum resources and effective. **G**raphical **U**ser **I**nterface comes with a much larger cost. Management tasks are generally easier on CLI (think about coding!).

The program that interprets user commands and provides feedbacks is called a *Shell*. When you login to the computer, the shell runs automatically. You interact with the computer through the shell.

Differenct choices of shell

The "standard" one on most linux is `bash`, as in "**B**ourne's **A**gain **Sh**ell". But choices like `zsh`, `fish`, `csh` are much more easier to use (cooler). Checkout the "oh-my-zsh" project on GitHub if you are interested.

Aspects of Linux: Shell

The working directory

How does `fopen("test.txt")` work? I mean, how does the computer know where to find `text.txt`?
Each program is associated with a special directory called "working directory". Normally this is the directory where you execute the program. (Not where the program is!).

Related commands

pwd **P**rint **w**orking **d**irectory

cd **C**hange **d**irectory. Each directory has 2 very special "sub-directory". "**./**" and "**../**". They are logically sub-directory. Meaning you can do `cd /home/john/./` and `cd /home/john/../`, but the in fact, the first command takes you to `/home/john/` and the second one takes you to `/home/`.

"`cd ./`" keeps where you are and "`cd ../`"" takes you 1 level up.

Aspects of Linux: Shell

Executing programs

The general syntax is

`executable_file arg1 arg2 arg3 ...`

Conventions are:
1) arguments begin with - are called "switches" or "options"
2) one dash (one -) are called short switches, e.g. `-l`, `-a`
3) short switch always uses single letter to specify. Case / lower letters can have very different meanings! Be very careful. 3) multiple short switches can often be specified at once. e.g. `ls -al`
4) two dashes (one -) are called long switches, e.g. `--all`, `--mode=linear`. Usually long switches use whole words other than acronyms.
5) **THERE ARE OUTLIERS!**. Unfortunately `gcc` and `g++` are two of these naughty boys.

Aspects of Linux: Shell

Useful commands

ls **List** files & folders under a directory. This command takes zero or more arguments. If the argument is a directory, list that dir. If the argument is a file, show information of that specific file. If no arguments are given, list working directory.

- -a List hidden files as well. Leading dot means "hidden".
- -l Use **l**ong format. Each line for a single file.

mkdir **Mk** **dir**ectory, self-explanatory.

rm **Rm** ove files / directory. It is **extremely dangerous** to run `rm` with administrative privilege! See the bumblebee accident. https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/issues/123

- -r Deletes files/folders recursively. Folders requires this option.
- -f Force remove. Ignores warnings.

rmdir **Rm** ove **dir**ectory, only **empty** ones can be removed this way.

Aspects of Linux: Shell

Useful commands, Cont'd

touch Designed to change the time stamp of file. Commonly used to create an empty file.

cp **Co**p**y** files/folder. Takes 2 arguments `source` and `dest`. Be very careful if both `source` and `dest` are both existing folders. Try it yourself!

-r Copy files/folders recursively. Folders requires this option.

mv **Mov**e files / directory. Takes 2 arguments `source` and `dest`. If `source` and `dest` is the same location, this command essentially does a rename. Pay attention to the situation where both arguments are already existed.

Aspects of Linux: Shell

Useful commands, Cont'd

touch Designed to change the time stamp of file. Commonly used to create an empty file.

cp **Co**py files/folder. Takes 2 arguments `source` and `dest`. Be very careful if both `source` and `dest` are both existing folders. Try it yourself!

- -r Copy files/folders recursively. Folders requires this option.

mv **Mov**e files / directory. Takes 2 arguments `source` and `dest`. If `source` and `dest` is the same location, this command essentially does a rename. Pay attention to the situation where both arguments are already existed.

cat Con**cat**enate files. Takes multiple arguments and print their content one by one to `stdout`. When there is just one argument, it essentially displays the content.

Aspects of Linux: Shell

CLI Utilities

nano Command line file editor.

diff Compare the **diff**erence of two files.

- `-y` A side by side view, try it yourself.
- `-w` Ignore white spaces.

less Prints the content from its `stdin` in a readable way.

vi Advanced text editor

vim **v**i **im**proved, both `vi` and `vim` can be exited by first press `ESC` and type "`:q!`". You should know this in case your "friend" opens a `vi` window when you are away, so you won't get stuck inside.

grep Filters input and extracts lines that contains specific content. Very useful in debugging programs.

echo Prints its arguments to `stdout`.

Aspects of Linux: Shell

IO Redirection

When you are reading `cin` or writing to `cout`, you are essentially read/writing 2 special files. Again, abstraction, right?
It is possible to "switch" these two "virtual files" with real files before the actual execution of the program. This is called IO "redirection".
There are 4 most common redirections:

`exec < input`	Use `input` as `stdin` of `exec`
`exec > output`	Write the `stdout` of `exec` into `output`. Note this command always truncates the file. File will be created if it is not already there.
`exec >> output`	Similar to >, but it appends to ouput.
`prog1 | prog2`	Called a "pipe". Connects the `stdout` of `prog1` to `stdin` of `prog2`

Aspects of Linux: Shell

Globbing

Sometimes you don't care about one file, you care about **all** files. You can use * to represent any string in command arguments. This is called "globbing" and the * is called a wildcard.

- List every `.cpp` file in home dir: `$ ls -al ~/*.cpp`
- Delete everything in `/temp`: `$ rm -rf /temp/*`
- Combine all `.h` into one: `$ cat *.h > combined.h`

Looking for help

The "goto" location for help in Linux is the `man` command. This is a short hand of "**man**ual".

- Find out information about "vi": `$ man vi`
- Confused about cp: `$ man cp`

Example of shell commands

Setup

The following program `rc2xm` reads from it's standard input line by line and prepends `xm` at the beginning and print it out.

```
#include <iostream>
int main() {
    std::string str;
    while(std::getline(std::cin, str))
        std::cout << "xm" << str << std::endl;
    return 0;
}
```

We prepare an input file `xm.in` with the following content:

`xd↩ cg↩ hss ↩ xtt ↩ qs ↩ jcc ↩ xdtql ↩ zdnxd ↩`

We use ↩ to represent new-line to save space on slides.

Example of shell commands

Examples

For each of the following commands what is it's effect? What is the final output?

The commands

```
$ cat xm.in
$ cat xm.in > xm3.in
$ ./rc2xm < xm.in
$ ./rc2xm < xm.in > xm.out
$ cat xm* > xm2.in
$ cat xm*.in >> xm.out
$ cat xm* | ./rc2xm > out
$ ./rc2xm <out | grep "xd"
```

Output of last command

```
xmxmxmxd↩xmxmxmxdtql↩
xmxmxmzdnxd↩xmxmxd↩
xmxmxdtql↩xmxmzdnxd↩
xmxmxd↩xmxmxdtql↩
xmxmzdnxd↩xmxmxmxd↩
xmxmxmxdtql↩xmxmxmzdnxd↩
xmxmxd↩xmxmxdtql↩
xmxmzdnxd↩xmxmxd↩
xmxmxdtql↩xmxmzdnxd↩
```

An extra story: Package Managers

The need for a package manager

- Packages a fancy name for "a piece of software"
- Linux fs convention separates package into different locations.
- Packages reuse code from other packages, i.e. *dependencies*.

The `apt` package manager

- Standard choice for Debian (and its derivatives)
- `apt-get install` for installing new package.
- `apt-get upgrade` for upgrading existing package.
- `apt-get update` refreshes the list of available software.
- `apt-get autoremove` removes installed package.
- `apt moo` for a tiny surprise :)

RC Week 3

Building a C++ program

The problem of *building* complex programs

Building is different from *compiling*

- Compiling refers to the process of translating code to binaries.
- Building is **piecing together** from its components.
- A program might depend on other package.
- A program might use a pre-compiled library.
- A program might involve more than one source files.
- A program might need to be built for different platform
- Sometimes you not only needs to build just one executable, but also documentations / test suites / libraries for the sake of other programs.

How complicated is Linux kernel version 3.2?

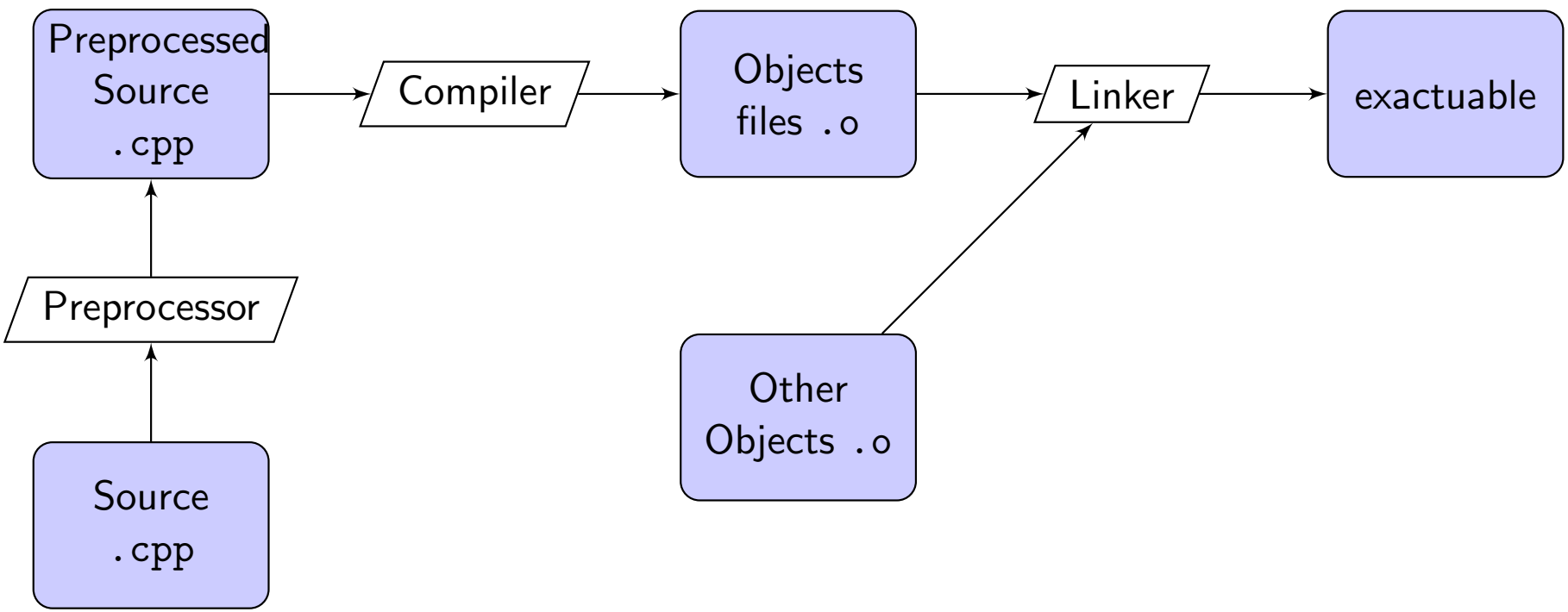
- 37,626 – The number of files
- 15,004,006 – The number of lines of code

Building a multi-file C++ program

The golden rule

Each source file (`.cpp`, `.c`) compiles independently.

The *building* process



The g++ tool chain

g++ as a all-in-one tool

- Preprocessor, compiler and linker used to be separate.
- Now g++ combines them into one.
- By default g++ takes source files and generate executable.
- Using different switches you can perform individual step.

Options for g++

-o out	Name the output file as `out`. Outputs `a.out` if not present.
-std=	Specify C++ standard. Recommend `-std=c++11`.
-Wall	Report all warnings.
-O{0123}	Optimization level. `-O2` is the recommended for release.
-c	Only compiles the file (Can not take multiple arguments).
-E	Only pre-processes the file (Can not take multiple arguments).

An example

This example contains a "main" source file accompanied with multiple other source files. All files are compiled separately into object files. We link some of them together and see what happens.

Keep in mind variables/function must be first declared before used.

-- > code/rc3build/main.cpp

```
#include <iostream>
using namespace std;
extern int number[], size;
int reduce(int n[], int s);
int main() {
    for (int i=0; i<size; i++) cout << number[i] << " ";
    cout << "\nReduced to " << reduce(number,size) << endl;
}
```

An example

```
--> code/rc3build/odd.cpp
int number[] = {1, 3, 5, 7, 9};
int size = sizeof(number) / sizeof(*number);
```

```
--> code/rc3build/even.cpp
int number[] = {2, 4, 6, 8, 10};
int size = sizeof(number) / sizeof(*number);
```

```
--> code/rc3build/sum.cpp
int reduce(int number[], int size) {
    int sum = 0;
    while (--size) sum += number[size];
    return sum;
}
```

An example

-- > code/rc3build/prod.cpp

```
int reduce(int number[], int size) {
    int prod = 1;
    while (--size) prod *= number[size];
    return prod;
}
```

The following file is a C source file. This file is given just for you to know you can do pretty weired things if you know the deal.

-- > code/rc3build/sum_large.c

```
int _Z6reducePii(int* number, int size) {
    int sum = 0;
    while (--size)
        if (number[size] > 3) sum += number[size];
    return sum;
}
```

An example

We compile the source files one by one.

```
$ g++ -o main.o -c main.cpp
$ g++ -o odd.o -c odd.cpp
$ g++ -o even.o -c even.cpp
$ g++ -o sum.o -c sum.cpp
$ g++ -o prod.o -c prod.cpp
```

Next one is compiled through gcc

```
$ gcc -o sum_large.o -c sum_large.c
```

Next step we are going to link (some of) them and execute it.
Linking in g++ is easy. If you supply `.o` files, g++ will know that is should link them instead of compiling them.
Pay extra attention to compiler errors (actually linker errors), they are the most interesting part.

An example

Now first standard examples

```
$ g++ -o main main.o even.o sum.o && ./main
$ g++ -o main main.o even.o prod.o && ./main
$ g++ -o main main.o odd.o prod.o && ./main
```

Now what if we link both `even.o` and `odd.o`

```
$ g++ -o main main.o odd.o even.o prod.o && ./main
```

Now what if we link both `prod.o` and `sum.o`

```
$ g++ -o main main.o odd.o sum.o prod.o && ./main
```

Now what if we leave out both `even.o` and `odd.o`

```
$ g++ -o main main.o prod.o && ./main
```

Now what if we leave out the `main.o`

```
$ g++ -o main even.o prod.o && ./main
```

An example

Surprises

Now we introduce something crazy. The name of the function in `sum_large.c` is really strange. But we just ignores that link its object file any way.

```
$ g++ -o main main.o even.o sum_large.o && ./main
```

Well it worked. The question is how on earth can this work. In fact g++ is doing some crazy renaming when compiling your source code. The reason why they did this is understandable when you think about it in the later period of the course.
Understanding linking actually allows you to do some crazy things. Try compiling the following file (with only one line of code) on your machine.

```
int main[-1u] = {1};
```

It tooks quite long to finish. How large is the executable?

Headers and inclusion

`#include<>` : Why we need them?

- Things must be declared before used.
- Each source file compiles independently. Needs a method to "export" functions defined in one file to other files.
- Avoid repeating declarations.

Preprocessing

- Preprocessing is purely **textual**.
- `#include` simply copy the content.
- *Conditional compilation directives* simply deletes unused branch. (`#ifdef`, `#ifndef`, `#else`, ...)

Header guards

problem

Whenever there is dependence of source files, there will be dependence of headers.
Consider the following `a.cpp`, `a.h`, `b.h` and `c.h`. Keep in mind that *everything in C++ is allowed to have at most 1 definition during compilation.*

`-- > a.cpp`

```
#include "a.h"
#include "b.h"
int main() {...}
```

`-- > point.h`

```
struct Point{
    int x, y;
}
```

`-- > a.h`

```
#include "point.h"
int area(Point a, Point b);
```

`-- > b.h`

```
#include "point.h"
void circle(Point o, int r);
```

Header guards

Solution

The idea is to use a unique macro to guard a header.

- Define that unique macro when the header is first included.
- Check if the macro is defined in future inclusion.

Now `point.h` becomes:

```
#ifndef _POINT_H_
#define _POINT_H_
struct Point {int x, y;}
#endif
```

The macro could be something else. Just don't use something common.

Build systems

The need for a build system

- Build process is complicated, avoid type every command.
- Project have dependence, need to manage dependence
- Compile minimum amount of code possible upon update.
- Many other reasons, abstract out actual compiler, compile for different platform / target.

Choices of build systems

GNU/make Our choice of make system. It has a very long history.

CMake A modern make system used by CLion and many other projects. Very flexible and reliable. It is also a cross platform solution.

MSBuild Build system used by Visual Studio.

Makefile and it's syntax

The *Makefile*

- The executable for GNU/make is simply `make`
- `make` requires a file that describes the building process. Such file is named `Makefile`.
- `Makefile` is made up of *targets*. A target can depend on other target, or some file.

Syntax

The following syntax defines a target. Note the `tab` key.

```
TargetName :  Dep1 Dep2 file1.o file2.o
⟶ Command1-to-run
⟶ Command2-to-run
```

Makefile : Example

This is a Makefile for our previous example. −− > code/rc3build/Makefile

```
all : sum_even
sum_even : objects
        g++ -o run main.o even.o sum.o
prod_odd : objects
        g++ -o run main.o prod.o odd.o
clean :
        rm -f *.o && rm -f ./run
onestep : main.cpp even.cpp sum.cpp
        g++ -o run main.cpp even.cpp sum.cpp
objects : sum.cpp prod.cpp even.cpp odd.cpp main.cpp
        g++ -c sum.cpp && g++ -c prod.cpp
        g++ -c even.cpp && g++ -c odd.cpp
        g++ -c main.cpp
```

RC Week 3

Misc of C++

Standardized C++

Once upon the time, programming languages are just conventions, design choices made by the language creator.

The standardize process

- Establishes program syntax, what are acceptable and what are syntax errors?.
- Language semantics, what's the "meaning" of an expression / language construct.
- Behavior, what are the expected behavior and what are undefined and left to the choice of compilers ...
- Standard library, what to include and what's the implementation constraint.

The latest standard is C++17 (3/21/2017). Major standards are C++98, C++03, C++11, C++14. C++ after C++11 is generally considered "modern C++".

Online reference for `std::to_string()`

The following information comes from
`http://www.cplusplus.com/reference/string/to_string/`

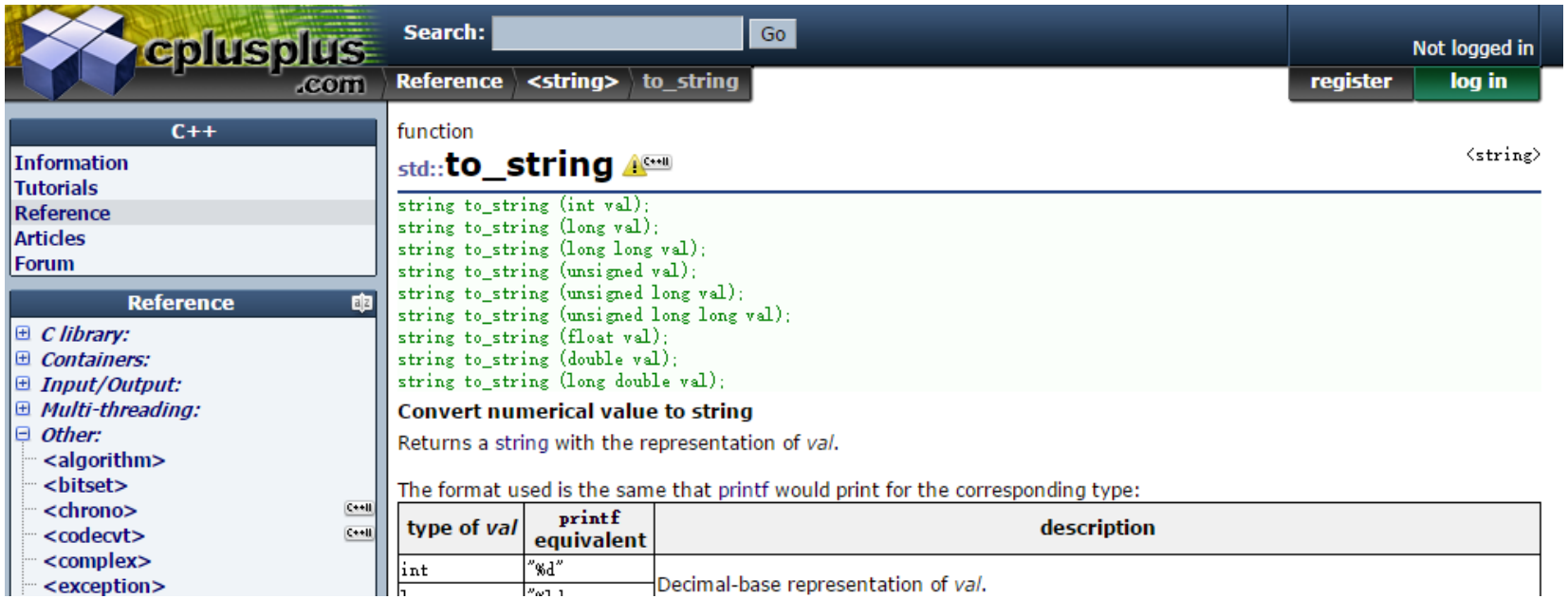


Figure: Online reference for C++11 library function `to_string`

- Notice the C++11 sign.
- Notice the overloads supported by this function.

Undefined Behaviors

One outcome from the standardize process is that, almost every true-or-false question about the C++ program can be answered with one of the following decisively. It is either YES, NO, or more importantly **undefined behavior** (*UB* for short).
Undefined behaviors are program whose output depends on a specific platform, or a specific implementation of the compiler.
You should always remember the following:

- It's an absolute waste of time trying to figure out what will happen given an code that contains UB.
- It's dangerous and to write code that contains UB.
- Anyone who test you with UB, is both stupid and ignorant.

There is a reason why UB exists. It's not that the committee doesn't know how to eliminate them, but they leave room for pretty impressive *compiler optimizations*.

Undefined Behaviors

Any (zero or more) of the following may happen if you trigger any of undefined behaviors:

- The compiler may refuse to compile.
- The compiler still compiles, but throw you an warning
- The compiler compiles silently.
- Your program crashes when executed.
- Your program malfunctions when executed.
- The compiler deletes all your photos.
- 72 fairies come out of your screen and dance around you.
- **Your program works perfectly.**

It's your job to avoid UB. We may refuse to answer the "why my program works locally but crashes on OJ" type of question.

Undefined Behaviors

dCommon cases

- Integer overflow (No, it's not guaranteed to be negative!)

```
int x = INT_MAX; x++;
```

- Dereferencing `nullptr` (No, it's not guaranteed to be crash!)

```
int* x = nullptr; *x = 2;
```

- Array out-of-bound (Even taking address is UB!)

```
int x[10] = {0};  x[10] = 1; int* x = &(x[11]);
```

- Dangling references (You could still get correct value)

```
int* x = int[10]; x[3] = 5; delete[] x; cout << x[3];
int* f(int t) {return &t;} int* x = f(10); cout << *x;
```

- Evaluation order and side effect :)

```
int i = 0; i = i++; // Yes that's UB
int j = i++ + ++i; // Yes that's UB too
f(j = i, i = j); // That's right UB again.
```

Declaration versus *Definition*

This whole discussion is based on one fact.

- C++ is *statically typed*
- Type of every identifier must be known when used.
- This "identifier" includes functions, variables and arrays ...
- However the actual implementation can be specified later.
- A *declaration* is a statement about what the type of an identifier (variable / function) is.
- A *definition* is a statements about what that object actually is. That is how much memory it consumes, what's it's data.

You have met declarations and definitions in Page.38. `extern int numbers[];` and `int reduce(int n[], int s);` are examples of declarations. The actual definition is in another file.
The `extern` keyword, why is it there?

Function *signature* / *prototype* and *definition*

A declaration for a function is called a function prototype. The "type" of a function is called it's *signature*. (Technically the "signature" is not the "type", but that's the idea.)
How much information do you need to describe the function?

- The name, we need to be able to refer to that function
- Return type, the "codomain" of the function.
- Number of inputs and their type. "domain" of the function.
- Formal parameter name.

Combining them, the standard form of a function declaration will be:

```
ReturnType functionName(T1 arg1, T2 arg2, ...);
```

I guess you are quite familiar with it. Note a definition of a function automatically introduce it's declaration (so the rule that everything must be declared before used is still intact).

Example

Now a really simple example. Consider writing a declaration for the following add function.

```
int add(int x, int y) {return x + y;}
```

No doubt above is a function definition. We first write

```
int add(int x, int y);
```

Well, that's a right answer. But we could also do

```
int add(int elephant, int haskell);
```

Suprised? Well it makes sense since changing formal arguments doesn't change the function at all! $f(x) = x$ and $f(z) = z$ are the same function. But we could push this even further!

```
int add(int, int);
```

This will work as well, Why? This kind of declaration will be useful when you learn about function pointers.

lval and *rval*

Compare the following 2 expression, suppose arr is an array of integers

1. `arr[10]`
2. `arr[10] + arr[1]`

They both have the type `int` of course.

- `int *p = &(arr[10]);` makes sense.
- `int *p = &(arr[10] + arr[1]);` gives you compile error.

Further more

- `arr[10] = 10;` makes sense.
- `arr[10] + arr[1] = 20;` doesn't

Clearly the two expression are "different" in some sense. How? Think about memory! The first kind is called *left values* and the second is called *right values*. (Those are not technical definitions.)

References

What we discuss here applies only to non-const references!
Lvals always corresponds to a fixed memory region. This gives rises to a special construct called *references*.

```
int a = 1, b[10] = {2};
int& ra = a; int& rb3 = b[3];
a = 10; /* ra reads 10 */ ra = 20; // a reads 20
```

Think about references as aliases. Essentially, you are giving the memory region associated with `a` an extra name `ra` (memory region given by `b[3]` an extra name `rb3`).
Try resist the temptation to think reference as an **alias of variables**, but remember they are alias for the **memory region**.
References must be *bind* to a memory region when created. There is no way to *re-bind* of an existing reference.

Function argument passing

Syntacticly there exists 2 ways of argument passing:

Pass-By-Value

```
int f(int x) { return (x = 2);}
```

Pass-By-Reference

```
int g(int& x) { return (x = 2);}
```

We give the following code to demonstrate their difference:

```
int y = 10; f(y); cout << y; // returns 2, outputs 10
int z = 10; g(z); cout << z; // returns 2, outputs 2
```

From a language point of view, reference parameter allows the function to change the input parameter.

Some would argue there exists a third way of argument passing.

```
#define SQR(x) (x * x)
```

They have a point. We call this pass-by-name. But we choose to ignore that in this course.

Function argument passing

Remember we are discussing how C++ manages memory. Keep in mind a memory oriented point of view is of utter importance.
We comment these two in terms of memory:

- Using the pass by reference, the formal argument would be a reference to the actual argument, namely they refer to the same memory region.
- Using the pass-by-value, the formal argument would be an independent *copy* of actual argument.

Pass-by-reference sounded like that it is related to pointers. This is true, in many implementations, pass-by-reference is implemented through pointers.
In pass-by-value, the word "copy" is extremely interesting. The fact that we need to "copy" the argument, give rise to serious problems in the later sections of this course.

Function argument passing

This memory point of view discussion give rise to some argument:

- Reference introduce an extra layer of indirect access to the original memory object, which drags down the performance.
- Pass-by-value needs to copy the argument, which can be slow.

In light of these observation, we suggest the following:

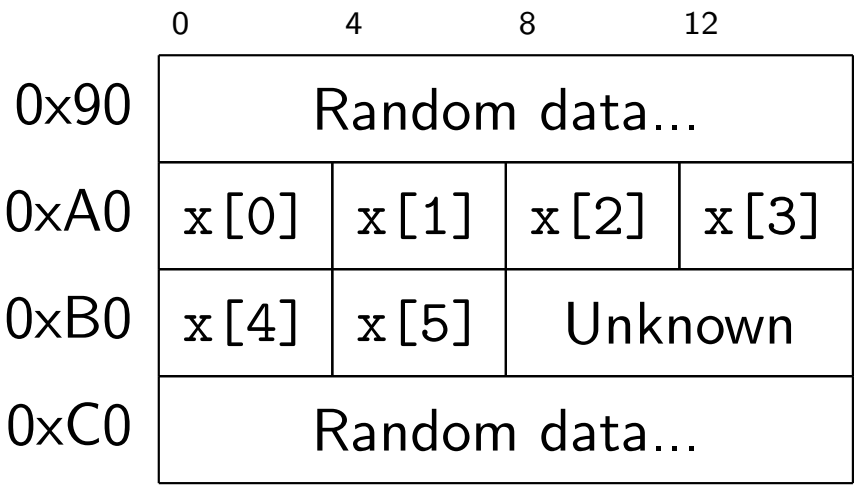
- Small types (`int`, `float`, `char*`...) better passed by value. The cost of indirect access is much more than copying them.
- Complicated structure, especially large ones, or class object, better passed through reference.

On the other hand, references allows the function to change the parameter, and sometimes would like to enforce invariance of arguments. This will be solved later.

Memory layout for *array* and *struct*

Array

Arrays are arranged in a consecutive way in memory. Consider `int x[6];`. Each `int` costs 4 bytes.



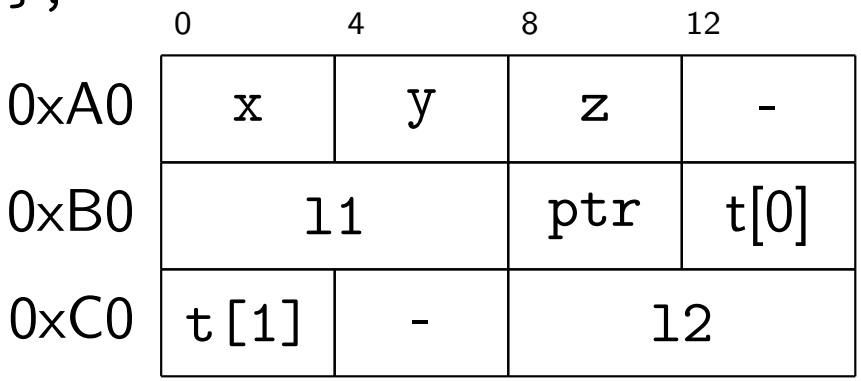
Note `x[i]` is always `*(x + i)`. `x` essentially holds address `0xA0`.

Structures

Warning: C++ standard does not fully specify memory layout.

Consider a structure (32bit)

```
struct S {
    int x, y, z; long long l1;
    S* ptr; int t[2];
    long long l2;
};
```



Remarks on arrays

Arrays are passed by passing the address of its first element. That **address is copied**, in this sense the actual content of the array is passed as references.

You might wonder the difference of the following 2:

```
int foo(int x[], int size);
int foo(int *x, int size);
```

In practice there aren't any (if not templates).

Remember the following expression does **NOT** make sense:

```
int foo(int x[size]);
```

Whenever you need to pass an array as an argument, remember to **pass it's size along with it**!

Footnote: In fact C++ considers the size of the array part of its type, meaning C++ believes for `int x[3];` and `int y[4];`, `x` and `y` has different type. It's just happens that both types can be converted into `int*` when passed as argument. See `code/rc3arr/run.sh`

Remarks on structures

- `sizeof` a struct is NOT the sum of `sizeof` its members.
- Structures arrange its members in the order of declaration.
- Members of a structure are put in the continuous region.

You need to think structures as a new type: structures a models a new type of data, a new concept. It is a **compound type**.
If you pass an structure by value, and if there is an array embedded in the structure, the array will be copied. This makes sense since the array is considered part of the value of the new datatype.
In C++, structures are simply "`class`"es with one change : members of a structure are by default `public`. This implies classes follows the same set of rules in terms of memory layouts.
We omit the discussion on pointers of structure.

Initialization of Arrays

The rules of initialization is extremely complicated in C++, this is due to two (competing) reasons:

- C++ tries to keep backward compatibility with C.
- C++ needs a unified modern syntax to initialize things.

This gets even more complicated if you take into account the fact that `structs` are essentially same as `classes`.

These rules applies also to `new`/`new[]` operator.

We first consider initializing an array:

```
int arr0[5] = {1, 2, 3, 4, 5};
int arr1[5] = {1, 2}; // {1, 2, 0, 0, 0};
int arr2[5] = {1}; // {1, 0, 0, 0, 0};
int arr3[5] = {0}; // {0, 0, 0, 0, 0};
int arr4[] = {1, 2, 3} // arr4[3] = {1, 2, 3}; useful!
int arr5[3]{1, 2}; // {1, 2, 0}; c++11 style;
int arr6[3]{}; // {0, 0, 0}; c++11 style;
```

Initialization of structures

Consider `struct S`:

```
struct S {
    int x, y;
    int arr[2];
    double d;
};
```

You might need to specify `-std=c++11` when using these initializations.

Now think about it, how to initialize `S sarr[2];`?

Curly-brace enclosed initializer

```
S s1 = {1, 2, 3, 4, 1.0};
S s2 = {1, 2, {3, 4}, 1.0};
```

We recommend the second one.

Default constructor

```
S s1{};
```

This initializes all fields to zero.

In-place

```
struct P {
    int x = 1, y = 2;
    int arr[2] = {3 , 4};
    double d = 1.0;
};
```

Function arguments: pointers or references?

We have one last question. Consider the following code:

```
struct S {int arr[100];};
void foo(S* s); void bar(S& s):
```

Now two functions are functionally same, which should I perfer?

- Pro-ref: References guarantees non-`null`, which is safer.
- Pro-ptr: Pointers allow `nullptr`, allows for the idea like "not-applicable" or "default value".
- Pro-ref: References good for *ownership*, semantically clear.
- Pro-ref: Syntactically cleaner. No need to modify call sites.
- Pro-ptr: References are used like values, not intuitive. Readers of the code has to jump to decl. Example `std::getline`.
- Pro-ref: Pointers are easily confused with array.
- Pro-ptr: Pointers are the only way to work with array. (If you ignore `std::vector`).

References as an abstraction

We would like to quote the following words from the standard:

> *References are not objects; they do not necessarily occupy storage, although the compiler may allocate storage if it is necessary to implement the desired semantics.*

In this way references are kind of special, since all usual types are connected to some memory: an `int` is 4 bytes and there is a 32-bit binary value in those 4 bytes. Pointer are addresses, they occupies 4 bytes in the memory and there is a 32-bit address in those 4 bytes.

Reference is defined through *abstraction*: We have a contract with the language on how this thing should behave, but we make no assumption on how the compiler achieve such effect. Compiler could choose to use pointers to achieve references, but on the other hand it doesn't have to.

Size of a reference type

- References are often implemented by pointers.
- Now what is the size of the reference type.

Now `sizeof(int&)` will not work. Std. requires `sizeof` the reference type returns `sizeof` the referenced type, `sizeof(int)`. But we could try

```
struct S {int x; int& y;} cout << sizeof(S);
```

Will this work? Consider the code on the left side. Output?

```
struct S {int x; int& y};
int main() {
    int a = 3;
    S s = {a, a};
    cout << sizeof(s);
}
```

It could be 8
Implemented by pointers, 4 bytes.
It could be 4
Compiler discovered it is not used. This also obeys the std.
It could be 5, 6, 7, 9, 120. Why?

RC Week 4

Deciphering Type Declarations

Design choice of declarations.

Quote from *The C Progamming Language*

The syntax is an attempt to make the declaration and the use agree. (In it) ... parentheses are over-used.

Declaration	Intepretation
`int a;`	`a` is an `int`
`int arr[4];`	`arr[]`→`int`, `arr`→array of `int`
`int *p;`	`*p`→`int`, `p` →pointer to `int`
`int *(pa[4]);`	`*(pa[4])`→`int`, `pa[4]`→pointer to `int`, `pa`→array of pointer to `int`.
`int (*pp)[4];`	`*pp` →array of `int`, `pp` →pointer to array of `int`.

const modifier

Whenever a type something is const modified, it is declared as "immutable". Example:

```
const int a = 10; a = 2; // Compile error
struct P {int x = 1, y = 2;};
const P p; p.y = 3; // Compile error
```

Remember this immutability is enforced by the compiler at compile time. This has a very strong implication. The compiler does NOT forbid you from doing strange things intentionally.

```
const int a = 10;
int *p = const_cast<int*>(&a); // C++11 style cast
*p = 20; cout << a; // Will this output 20?
```

Well this is actually UB. const is not a guarantee of immutability, it is an **intention**. It asks the compiler to look out for you, if you know you shouldn't change something.

const and pointers

We now combine the previous discussions.

`const int *p`	`*p` is of type `const int`, thus changing `*p` is illegal. However this declaration does **not** say anything about `p`, thus changing `p` is possible. This is called *pointer-to-const*.
`int *const p`	Equivalent to `int *(const p)`.
`int *(const p)`	This declaration essentially says if you dereference `const p`, you will get `int`. Since `int` is not quantified by `const`, you can change `*p`. However, the pointer itself, is modified by `const`, so you can change `p`. This is called *const-pointer*.

Naturally you could have `const int *(const p)`. This declaration basically says both the pointer `p` itself and the dereferenced object (`const int`) cannot be changed.

`const` and reference

Recall that **references cannot be rebind once initialized**. The following definitions are equivalent. They are all *const references*.

- `const int& iref`
- `(const int)& iref`
- `int& (const iref)`
- `const int& (const iref)`

The second one makes the most sense, although the first one is the most commonly used.
The second one essentially says,`iref` is a reference, or an alias to a memory region, that is protected by the `const` modifier.

const reference and argument passing

There is something special about const references:

> Const reference are allowed to be bind to right values, while normal references are not allowed to.

Normally if a const reference is bind to a right value, the const reference is no difference to a simple const.

```
int a=5; const int& r=a+1; const int c=a+1;
```

In above example practically there is no difference between `r` and `c`. But when you pass arguments through const references, things become a little bit different.

```
int foo(const ReallySuperLargeStruct& s);
```

- We are passing by reference, this avoids copying.
- `const` enforces immutability.
- `rvals` can be passed directly into it (unlike pointers).

Example

```
int foo(int); int bar(int&);
int baz(const int&);

int q = 10; int& rq = q;
const int& crq = q;
int* pq = &q;
const int* cpq = &q;

void foobar() {
    crq = 5;    // Error!
    *cpq = 20; // Error!
    q = 30; cout << crq;  // 30
    *pq = 4; cout << *cpq // 4
}
```

```
int bazbok() {
    baz(q);   baz(20); // OK
    baz(*pq); baz(rq); // OK

    bar(q);   bar(*pq); // OK
    bar(10);   // ERROR
    bar(*cpq); // ERROR
    bar(crq)   // ERROR
}
```

const propagation and type coercion

Type coercion and type compatibility

You should be very familiar with types now. Types defines different kinds of things. Some of those things are *compatible*, meaning one could be transformed into another implicitly. But sometimes you need to explicitly *cast* them, or coerce them into another type.

```
int x = 20; long int y = 30; char* p = "hello";
y = x; /* OK */ x = y; // Compiler warning
int x = p; // Incompatible, compile error
```

A remark is that most of the things are incompatible, simply because these different type are different from the hardware point of view. Pointers are addresses, but int are number. `long int` occupies more space than `int` etc.

const propagation and type coercion

const modifier introduce incompatibility

The subtitle is summarized into the following rules:

- `const type&` to `type&` is **incompatible**.
- `const type*` to `type*` is incompatible.
- `type&` to `const type&` is **compatible**.
- `type*` to `const type*` is compatible.

Example:

```
int foo(int& x); int bar(int* px); int cfoo(const int& x);
void baz() {
    const int *q = nullptr; int *p = q; //Compile error
    const int& r = 10;  foo(r); //Compile error
    cfoo(*p); cfoo(*q); cfoo(r); // All OK
}
```

const propagation and type coercion

const progagation

Further more, const modifier attaches extra constrain when trying to take address or dereference / reference things.

- Take address of the const references gives pointer-to-const.
- Dereferences pointer-to-const give const references.

Example:

```
int foo(int& x); int bar(int* px); int cfoo(const int& x);
void baz() {
    const int q = nullptr; int *p = &q; //Compile error
    foo(q); bar(&q)                     //Compile error
    const int& r = q; foo(r); bar(&r); // Compile errer
}
```

This creates an important feature of C++ constness:
You either do it in full scale, or not at all.

`const` compatibility in terms of abstraction

Again I hope you could see something bigger. These rules are unlike those mentioned earlier. The compiler probably use the same representation for `type*` and`const type*`.
They are different, because they follow different abstraction. The abstraction for `type*` supports modification, or it is supported to appear on the left hand side of the assignment operator, while the abstraction of `const type*` does not.
This point of view also explains the propagation rules. The type system must be designed in a coherent way. If an object doesn't support modification, changing the reference to pointer won't suddenly equip it with modification. The abstraction must be maintained between operations.
This gives a very first look over the abstraction point of view of the concept "datatype". Now we see that datatype not just the memory layout of objects, but more about abstraction, operations that the type supports.

Functions Pointers: Why Pointers?

The Von Neumann View of functions

Functions are code, and code when compiled are simply binary number, i.e. data. The action of calling a function is simply pumping these binary numbers into the CPU (after you take VE370 you would find it's actually the other way around).

- Functions are just a bunch of numbers in the memory
- We could refer to the function by refering to the numbers
- These numbers has an *address* (think of arrays)
- We could use that address to refer to the function

Variable that stores the address of functions are called *function pointers*. By passing them around we could pass functions into functions, return them from functions, and assign them to variables.

Functions Pointers: Type

But there is one question, what is the type of them?
Well by our previous understanding dereferencing a function pointer should give us a function, just like dereferencing `int*` gives us `int`. We would like to do a comparison:

Function decl. `foo`

- `void foo();`
- `int foo(int x, int y);`
- `int foo(int, int);`
- `int *foo(int, char*);`
- `char* foo(int[], int);`

Function pointer `bar`

- `void (*bar)();`
- `int (*bar)(int, int);`
- `int (*bar)(int, int);`
- `int *(*bar)(int, char*);`
- `char *(*bar)(int[], int);`.

Note `int *bar(int);` does **NOT** declare a function pointer. The grouping is `int *(bar(int))`, which is declaring a function. This is related to the operator precedence of C++.

Functions Pointers: Usage

Assignment from functions

In fact, the identifier (name) of the functions are actually values of function pointers.

```
int max(int x, int y) { return x > y ? x : y;}
int (*cmp)(int, int) = max;
```

Invoking a function pointer

You can invoke a function pointer by applying operator `()` to it.

```
int m = cmp(10, 20); // No need to dereference it
```

Invariance under *

Dereferencing a function pointer still gives back a function pointer.

```
int m = cmp(10, 20); // 20
int n = (*cmp)(10, 20); // 20
int p = (*******cmp)(10, 20) // 20
```

Example

The following code implements a simple calculator. Notice how function pointer helps to clarify the code.

−− > code/rc4fptr/fptr.cpp

```
#include <iostream>
int add      (int x, int y) {return x + y;}
int subtract(int x, int y) {return x - y;}
int (*fun[])(int, int) = {add, subtract};
using namespace std;
int main() {
    int op = 1, x = 0, y = 0;
    cout << "Select your operation (1,2): "; cin >> op;
    cout << "Numbers: "; cin >> x >> y;
    cout << "ANS = " << fun[op-1](x, y) << endl;
}
```

`typedef` storage modifier

The **typedef** keyword is actually a storage modifier (like `static`), means it goes to wherever `static` can appear.
Reading a `typedef` is actually very simple:

1. Ignore `typedef` and read the line as if it is a real declaration.
2. Now the "declared" variable would have a declared type
3. Finally we put back `typedef`. The variable now become an alias to the declared type.

Example:

```
typedef bool (*comparator)(double, double);
typedef const Comparator const_comparator;
const_comparator *p = nullptr;
*p = min; // Error!, p is pointer-to-const
comparator cmp[] = {min, max}; // Array of function ptrs
```

Semantic of typedef

Motivation

It is now to analyze the semantic of `typedef`: not how is used, but why is it used in such way. If you go through the standard library. you would find some code like:

```
typedef unsigned int size_t;
```

What's the point of doing this if when you can just use `unsigned int` ? Suppose you are trying to find your name with your student ID, which is an unsigned integer:

```
string getNameByID(unsigned int student).
```

This is not good, because `student` is not any `unsigned int`, it must be a student ID. You would like to emphasize on that:

```
typedef unsigned int student_id;
string getNameByID(student_id id);
```

Semantic of `typedef`

But here comes a problem:

`typedef` creates alias, not new types.

This essentially says both `student_id` and `size_t` are both alias for `unsigned int`. They are essentially considered the same thing. The following code does not **NOT** make sense:

```
size_t numberOfApples = 20;
string name = getNameByID(numberOfApples);
```

but the compiler lets you do this without even a warning.
Semantic of the types should prevent you from making these assignments (easily), but `typedef` breaks this promise.
On the other hand, `classes` are more modern. Two `classes` with same "composition" are considered incompatible different types.'

RC Week 4

Procedure Abstraction

Abstraction and *decoupling*

The motivation of all abstraction comes directly from the need for decoupling. We would like to design the software in such way, that the change in one portion of the code does not affect the others.

But after all some degree of coupling is inevitable, conceptually we would like to limit the coupling only to the interfaces, i.e. where modules connect.

Low coupling:

High coupling:

Low coupling for flexibility:

Independence of variation

We some times simply refer these interfaces, or connection points, *the abstraction*. Now we take a look at the *implementation* side of the abstraction.

If you the abstraction implementation is done right, you should find the following property of your abstraction.

- Locality. The implementation should only depend on other abstraction, not abstraction implementation.
- Substitutable. The implementation is not unique. Any implementation that obeys the abstraction should work.

The ultimate goal is the the only coupling point in your program should be the abstraction. So this is utterly import to always think through the abstraction before the actual implementation.

Elements in procedure abstraction

A common point of where modules connect is function calls. But think bigger, when you say `printf("%d", 1)`, you are requesting a service, requesting for certain operation. This idea that programs are driven by requests for operations sums up to the idea of *Procedure Abstraction*. Remember functions are just one realization of the idea of Procedure abstraction, but not the only one.

We now examine how functions interact with the outside world. These things will be very important in specifying the abstraction:

- Input and it's assumption
- Expected output, and the property of the output
- Impact on the environment, or side effects.

Specifying abstraction

Typically an abstraction is specified in the following manner:

```
int reallyUselessFoo(int& intArg, string str);
// REQUIRES : intArg > 0, str != ""
// MODIFIES : cout, intArg, UsefulGlobalVar
// EFFECTS  : do Blah blah blah and return blah blah
```

- *REQUIRE* clause and the function signature specifies the input. Number of inputs are fully specified by the signature, input range can be partially specified through type.
- *EFFECTS* clause and the function signature specifies the output. The expected output is partially specified by the type of the return value. It is important to note that the function name is also part of the specification, it should clearly and concisely reflect on what the function does.
- *MODIFIES* clause specify side effects.

Side effects

We have left out the term "side effects" for now. Let's take a look at them. Think of mathematical functions, you would expect the following of them:

Functions are maps

Functions should return the same if provided same input:

```
(fun(x) == fun(x)) == true
```

Functions are stateless

Programs should behave the same for the following two lines

```
int x = foo(y); int z = bar(x, x);
int z = bar(foo(y), foo(y));
```

The following 2 lines should also be equivalent.

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

Side effects

It is our intuition that all these assertions should be true. But unfortunately we can break them. Once we break these intuitions it will be very hard for programmers to keep track of program.

Through global/static variables

```
int x = 3; int foo(int y) {return y + x++;}
int bar(int y) {static int x = 0; return y + x++;}
```

Through modifying arguments

```
int x = 3; int foo(int& x) {return ++x;}
```

Involving IO

```
int x = 3; int foo(int x) {cout << "Hello"; return x;}
```

Functions are suppose to be maps, converting one value to another. In all these values the function does something else. They modify something that does not belong to them.
These behaviors are called *side effects*.

Function signature is important

A huge amount of information is provided in the function signature. In fact we believe the following should be true:

Good code should be self explanatory.
Good abstraction should be self documenting.

It means you should be able to guess the abstraction's usage, just by looking at it's function signature. Compare the following:

```
void dwcl(double a, double b, double c);
void drawCircle(Point p, double radius);
```

It is generally considered good to

- Avoid introducing too much arguments. Pack arguments that are related into structure / classes.
- Avoid using acronyms in function names. You have very good editors that provides extensive support for auto-completion.

Leaky Abstractions

Officially our discussion of procedural abstraction has ended. But I would like to make a few comments. We would like to ask:

Are there PERFECT abstractions?

Perfect abstractions are abstractions that completely hides information about the implementation, a complete black box. People often need a perfect abstraction, e.g. for cryptography. Compare the following, assume both arguments are positive:

```
int mul1(int x, int y) {return x * y;}
int mul2(int x, int y) {return x==0 ? 0 : y+mul2(x-1,y));}
```

Both implementation are correct. Can we differentiate them? The second one is significantly slower than the first one. If we provide a really large input, we would be able to differentiate them by the runtime! We say we have a *leaky abstraction* at our hand.

the Law of Leaky Abstractions

the Law of Leaky Abstractions (Spolsky, 2002) says:

All non-trivial abstractions, to some degree, are leaky.

How is this important? By now should be clear that the proper functionality of software (and computer system) often depends on the reliability of abstraction. But at some point, the abstraction will break. Examples:

- In C/C++ iterating through large 2D array horizontally is significantly faster than vertically.
- *L3-cache side channel attack*. Takes advantage a leaky abstraction inside the CPU to steal your "password".

As a measure to counter this problem you sometimes see the standard not only specify a function's behavior, but also it's runtime (in terms of time complexity).

Enforcing Abstraction in C++

Another very interesting thing here is what happens if you breaks the abstraction? How abstraction is enforced in C++?

- Input number and types are protected by the type system.
- Input But the properties of input (even?) are left unsaid.
- Output type and number (guarantees there exists one output if needed) is protected by the type system.
- Whether output fits the expected behavior is unchecked.
- Side effects are completely unchecked.

As you can see many things are left unchecked! Most of the checking is done by the type system. In fact, the type system plays a really important role in programming language. The fact that C/C++ is a statically typed language is greatly complimented and provides much help in writing correct programs.

Enforcing Abstraction: How far can we go?

Here I list a couple of other language's idea. You may or may not know that language, but that's completely alright.

Matlab/Python Dynamically typed. The type of arguments are not declared and only known at runtime.

Bash Scripting language used by shell. Not even typed. Not even check the arguments of function called. Basically zero protection.

Haskell A language that is statically typed. The type system is so powerful that it requires the programmer to declare not only the inputs, but also the "side effects" (and checks them).

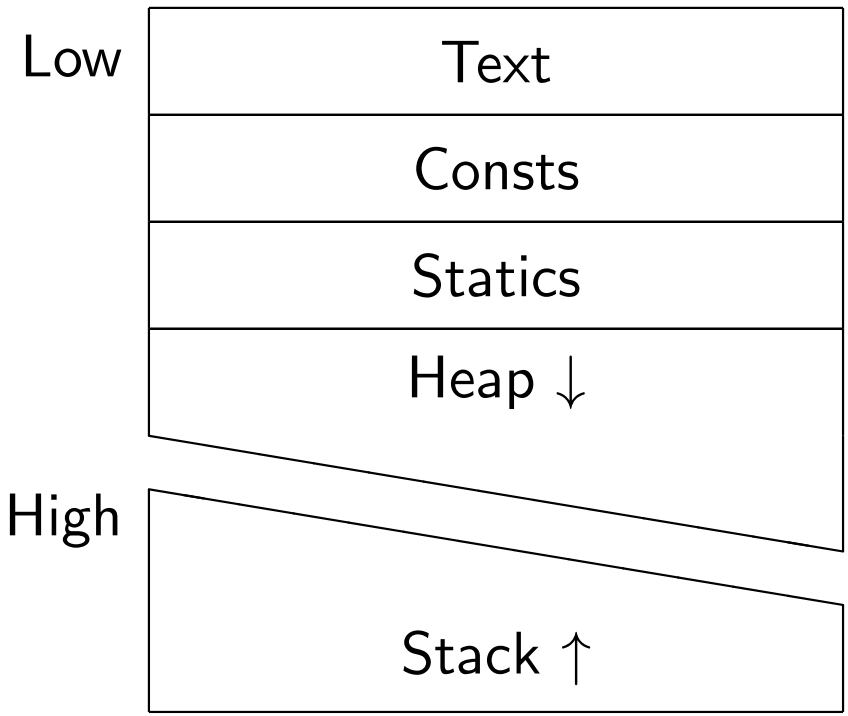
COQ A language that goes to the extreme. It not only asks the programmer to declare everything, it even requires the program to provide a proof (and checks the proof) of correctness.

RC Week 5

Mechanism behind function calling

Memory Layout

Before function calling we would like to first give a brief introduction to the memory organization of the system

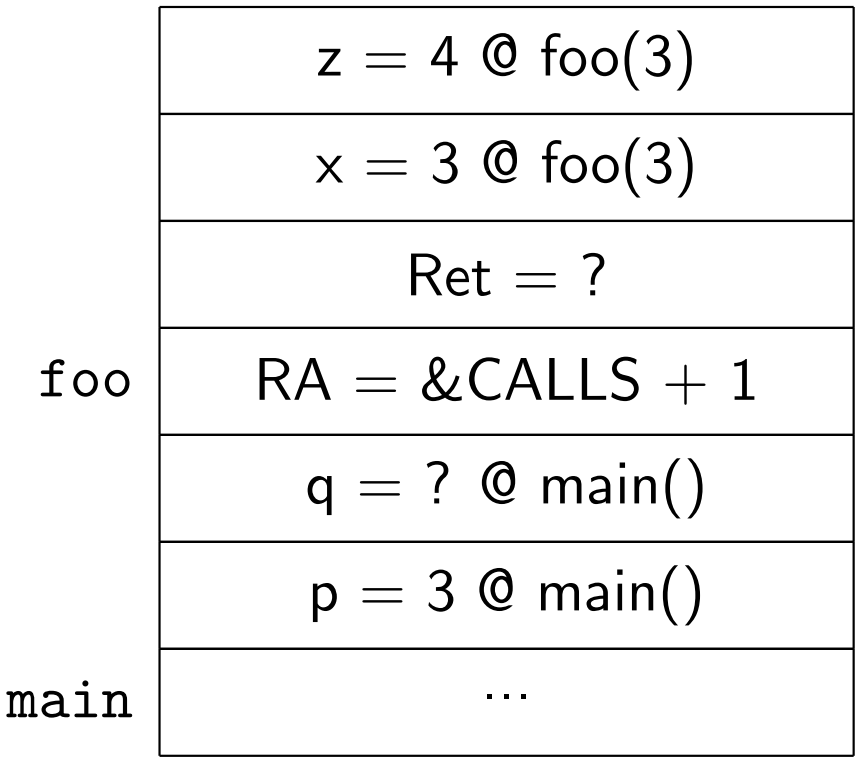


* This graph is up-side-down.

Explanation

- "Text", just "code"
- "consts", not like `const int const`, but string literals, `"Hello world"`.
- "Statics", global variables, static variables in functions. "static" refers to lifetime.
- "Heap", where you `new`.
- "Stack", everything local. Arugments, return values, return addresses ...

Element in a stack




```c
int foo(int x) {
    int z = x + 1;
    /* Here */
    return z + x;
}
int main() {
    int p = 3;
    p = foo(p); // <- CALLS
    int q = 10;
}
```

The stack maintains the following information

- Function arguments. They are evaluated and on to the stack.
- Local variables (arrays). They are reserved before initialized.
- Return value and return address. Return address tells which instruction to pick up when the call returned.

Remarks

Calling mechanism is about *Abstraction*

Calling mechanism is designed in such way to support procedural abstraction. In order for the abstraction to work, we require

Each function call is independent

This is especially important if you have recursion calls.

Calling mechanism is platform dependent

- Calling mechanism is neither specified by standard, nor unique!
- Whose responsibility to manage arguments? Caller / callee?
- Compiler might optimize unused variables out.
- Compiler might use register to store information.
- Compiler is allowed optimize the entire stack frame out.

Puzzles for FFFUUUNNNNN!

Under standing calling stack is most useful in finding out what has gone wrong when you observe strange behavior of your program. This is very much like solving puzzles.
We now give you a few such puzzles to entertain you. Note all the code we gave you below contains undefined behaviors so don't be surprised if you cannot reproduce this problem.
This is actually worth noting. Many undefined behaviors would cause different behavior since given different situation on the stack.

- This code works on my computer but crashed on OJ.
- This code crashes / malfunctions if I change unrelated things.
- Student: "TA, my code can't work!".
  TA : "Can you demo? I can't reproduce your problem"
  Student: "Suddenly I can't Either! But it crashes on OJ!"
- This code randomly crashes.

Puzzle 0: Why not VLA?

Variable Length Arrays (VLA) are arrays whose size are determined at compile time. For example in the following code `arrX` is a VLA, and `arrY` is a usual array.

```
void foo() {int t = 20; int arrX[t * t]; int arrY[400];}
```

But both C++ and C choose **not** to support this language feature (above code won't compile). Explain what's the underlying reason.

Puzzle 0: Why not VLA?

Variable Length Arrays (VLA) are arrays whose size are determined at compile time. For example in the following code `arrX` is a VLA, and `arrY` is a usual array.

```
void foo() {int t = 20; int arrX[t * t]; int arrY[400];}
```

But both C++ and C choose **not** to support this language feature (above code won't compile). Explain what's the underlying reason.

Explanation

What would be the impact of this feature? Variable length array will consume variable amount of memory.
If the array is a local variable. The stack frame size of the function cannot be determined in compile time.
The need to know the size of a function's compile time stack frame size is centric in language design.
‘

Puzzle 1: Orders matter

--> code/rc5pz1/a.cpp

```
#include <iostream>
using namespace std;
struct S{int x = 4; char a[4];};
void foo() {
    S s; cin >> s.a;
    while(--s.x) cout << s.a << endl;
}
int main() {foo();}
```

User inputed `Hello`, symptoms are:

- Program crashes after printing 4 times of `Hello`.
- Program runs fines if change input to `Bad`.
- Program doesn't crash if switch `char a[4]` and `int x = 4`. But the program keeps printing `Hello`.

Solution 1

4B	s.x = 4 @ foo()
1B	s.a[0] @ foo()
1B	s.a[1] @ foo()
1B	s.a[2] @ foo()
1B	s.a[3] @ foo()
`foo`	RA = !!
`main`	...

Return address is corrupted.

1B	s.a[0] @ foo()
1B	s.a[1] @ foo()
1B	s.a[2] @ foo()
1B	s.a[3] @ foo()
4B	s.x = !! @ foo()
`foo`	RA = &main + 1
`main`	...

Variable `s.x` is corrupted.

Puzzle 2: Missing return value?

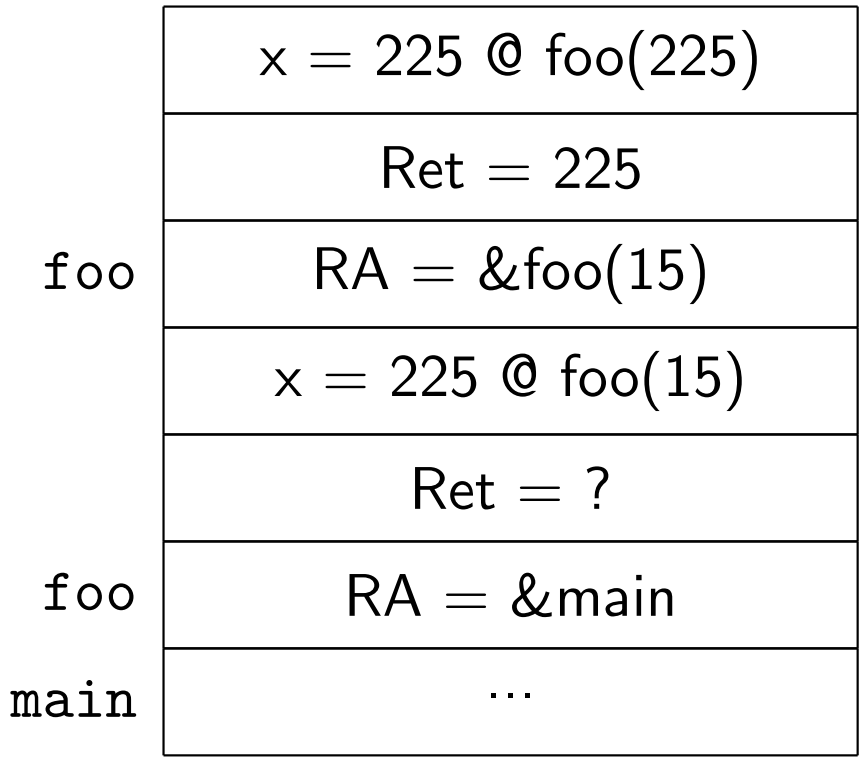
--> code/rc5pz2/a.cpp

```
#include <iostream>
using namespace std;
int foo(int x) {
    if (x < 100) { x = x * x; foo(x);}
    else return x;
}
int main() { cout << foo(15);}
```

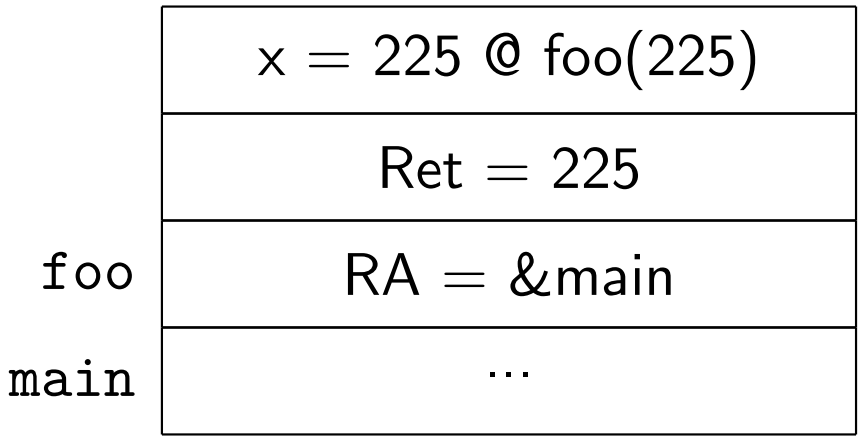
This code is supposed to keeping squaring a number until it's greater than 100.

- Program output random number if c/with `g++ a.cpp`
- Program always return 225 if c/with `g++ -O1 a.cpp`
- Program spits random number if c/with `g++ -O1 a.cpp`, if we change "`foo(x);`" to "`y = foo(x);`".

Solution 2



Without optimization



After optimization

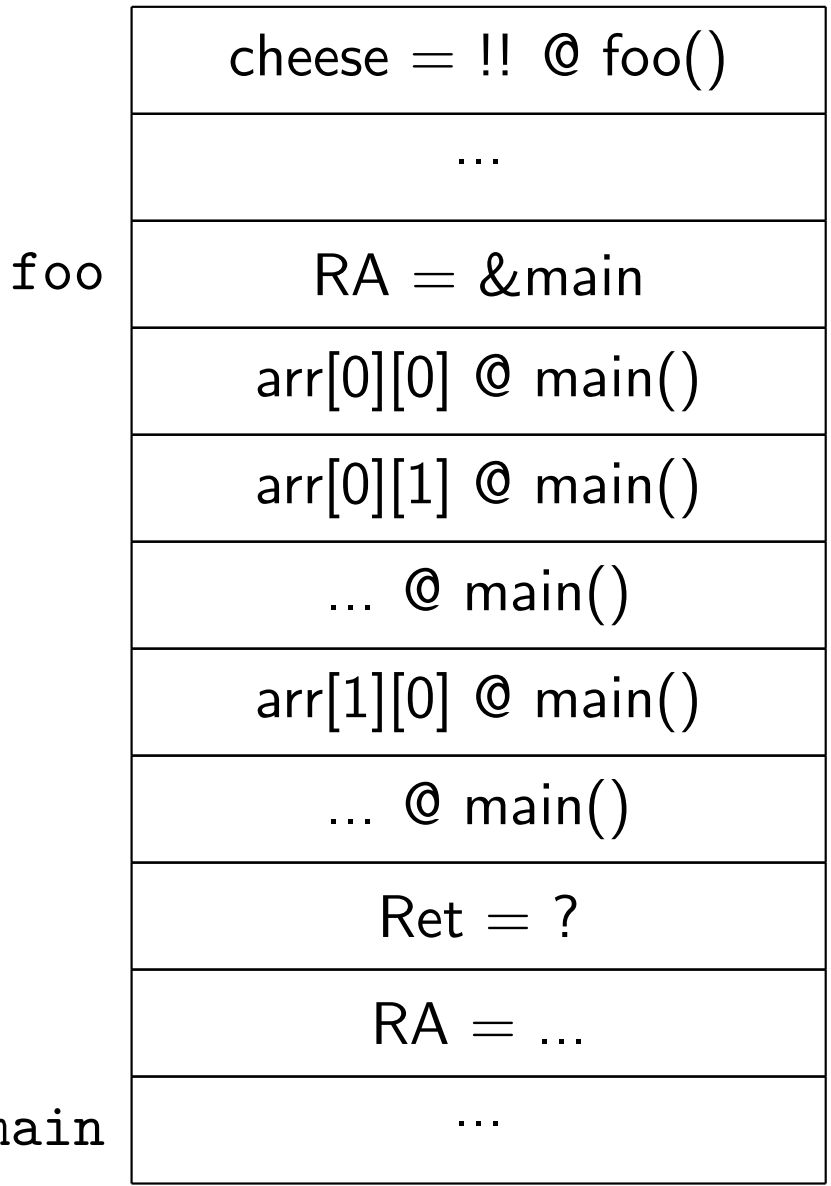
Puzzle 2.5: Who moved my cheese?

--> code/rc5pz25/a.cpp

```
#include <iostream>
using namespace std;
int setFirst(x[][5], int size) {
    int cheese = 0;
    while (size >= 0) x[--size][0] = size;
    cout << cheese << endl;
}
int main() {
    int arr[10][5] = {0};
    setFirst(arr, 10);
}
```

We observe the output to be -1. However we haven't changed the variable cheese. Who mov-ed my cheese.

Solution 2.5


cheese = !! @ foo()
...
foo
RA = &main
arr[0][0] @ main()
arr[0][1] @ main()
... @ main()
arr[1][0] @ main()
... @ main()
Ret = ?
RA = ...
main
...

Puzzle 3: A hijack

For this puzzle to work you might need to turn off *Address Space Layout Randomization* and set `-fno-stack-protector` in g++.

-- > code/rc5pz3/a.cpp

```
#include <iostream>
using namespace std;
void secretFunction() {
    cout << "You shouldn't be here..." << endl;
}
void echo() { char buffer[20]; cin >> buffer; }
int main() { echo(); return 0; }
```

The trick is a carefully constructed input:

```
voidsecretFunction2e34!2&&Q.%x;"]
```

We observe the output is `You shouldn't be here...`

The question is how does this happen, since `secretFunction` is not called at all?

Stack unwinding / unrolling

Hope you still remember *constructors* and *destructors*!
When the function returns, it not only simply discards the stack space, it actually *DESTROY*s the local objects, essentially calling their deconstructors.
This is a very useful feature! We can use this feature to automatically release resources. Consider the following `File` class.
`--> code/rc5unroll/file.h`

```
class File {
    string name;
public:
    File(string file) : name(file) {
        cout << "Opended : " << name << endl; }
    ~File() {
        cout << "Closed " << name << endl; }
};
```

Stack unwinding / unrolling

The following code executes utilizes the above code: $--> $ code/rc5unroll/a.cpp

```
#include <iostream>
using namespace std;
#include "file.h"
int foo() {
    File f1("file1.in"); File f2("file2.in");
    cout << "I Returned!" << endl;
}
int main() { foo(); }
```

We paste the output:

```
Opended : file1.in // Opended : file2.in
I Returned! // Closed file2.in // Closed file1.in
```

The files clean themselves up after returned.

Stack overflow

A final point is that in general your stack is rather small, compared to the heap.
We refer to an empirical experiment done by *Bruno Haible* in 2009.

```
- glibc i386, x86_64      7.4 MB
- Cygwin                  1.8 MB
- Solaris 7..10             1 MB
- MacOS X 10.5            460 KB
- OpenBSD 4.0              64 KB
```

Usually heap size is hundreds of MB, if not GB on a modern computer. It could happen that you ran out of stack space. In such situation we say you encountered a *stack overflow*, because:

- Maybe you recurse too deep (Why?).
- Maybe you declare large arrays in the stack.

RC Week 5

Recurse Recursively

Recursion is the art of abstraction

A typical processes of designing a recursive function goes as follows:

- Be very clear first about the abstraction of the function you needed to input.
- Specify a base case, a set of input where the answer is immediately known (or can be calculated in a few steps).
- Assume that your abstraction actually works for input "simpler" than the current input (closer to the base case). Use this assumption to build your program.

You might find these steps surprisingly similar to a mathematical induction. That's true. They are similar and that's very useful.

Recursion is the art of abstraction

```
// REQUIRES: list is not empty
// EFFECTS:  returns largest element in the list
int largest(list_t list) {
    int first = list_first(list);
    list_t rest = list_rest(list);
    if (list_isEmpty(rest)) return first;
    return max(first, largest(rest));
}
```

- The abstraction is specified in the header.
- The base base is where list contains just one element.
- In the last line, assume abstraction works in simpler case (hope you see why `rest` is "simpler" than `list`). Thus `largest(rest)` returns the largest of the remaining list.
- The largest number can either be the largest of the remaining number, or the first number.

Example: Quick sort algorithm

A quick sort algorithm sorts a array in a quick way (I know that sound like bullshxx!). But list basic steps as following:

- Select an element in the array. We simply use the first element. We call this element `pivot`
- We need to *partition* the array. We need to move the elements less than the pivot to the left and elements larger than pivot to the right. Note we assume on the order of the elements left of the pivot (or elements on the right of the pivot).

```
6 5 3 8 -1 7 3 9 11 -4    | 5 3 -1 3 -4 6 8 7 9 11
|<- pivot                 |              |<- pivot
```

- Call `QuickSort` on the both sides of the pivot.

The majority of the work lies in the partition step.

A usual Quick Sort

--> code/rc5qsort/nonrec.cpp

```
void quickSort(int *data, int left, int right) {
    int len = right - left; if (len <= 1) return;
    int pivotIndex = left; int pivot = data[pivotIndex];
    int *pData = new int[len];
    int top = 0; int bottom = len - 1;
    for (int i = left; i < right; ++i) {
        if (i == pivotIndex) continue;
        if (data[i] <= pivot) pData[top++] = data[i];
        if (data[i] > pivot) pData[bottom--] = data[i];
    } pData[top] = pivot;
    for (int j=0; j<len; ++j) data[left+j] = pData[j];
    quickSortHelperExtra(data, left, left + top);
    quickSortHelperExtra(data, left + top + 1, right);
}
```

Our Quick Sort

Suppose we are using our list interface (in the project!).

- Suppose `QuickSort` is a function that takes in a list and returns a sorted list. Remember this is abstraction. Base case is when the list is empty.
  ```
  list_t qSort(list_t lst); // Returns sorted lst
  ```
- Our computation goes as follows, we first acquire a the partion-left part and partition-right part. We call `qSort` on both parts and concatenate left, pivot and right part:
  ```
  list_t sorted = cat(qSort(left), pivot, qSort(right))
  ```
- The left part are simply numbers less than pivot, the right part are simply numbers greater than pivot.
  ```
  list_t left = filterLess(lst, pivot);
  list_t right = filterGreater(lst, pivot);
  ```
- And we are done. Now we simply copy everything into one place.

Our Quick Sort

And here is the famous (almost) one-line quick sort

```
list_t qSort(list_t lst) {
    if (isEmpty(lst)) return lst;
    int pivot = list_first(lst);
    return concatenate(
        qSort(filterLess(list, pivot)),
        pivot,
        qSort(filterGreater(list, pivot))
    );
}
```

Now it just leaves us to implement the filter function and the concatenate function. But these two functions should be very easy. You have implemented the filter function in your project right?

Why not references and pointers?

Think about it, why this seems much easier (clearer, hopefully you do feel that way)?
In the traditional code, all functions works on the same array. We must manually control the process of copying, moving, etc. We are thinking in terms of *operations*, detailed step to be performed.
But the the new code, we can now begin think of data. We stop focusing on the concrete steps, but simply

What should I do with the data?
What is the expected input and the expected output?

It is the computation we needed to focus.
This is not easy. The immutability of the data and the fact that all functions are pure allows us to do such thing. Every function does calculation on its own and does not impact the outside world.

Bridging the old perspective

Consider the following problem:

Write a function `isMoreOdd` that takes a list $(a_0, a_1, a_2, ..., a_n)$ and returns $\Sigma_{i=0}^{n} i^2 a_i$ (we call this an s-sum) Assuming the list is non empty.

An example as follows:

```
a_i: 6  5  3  8  -1  7  3
 i : 0  1  2  3  4   5  6
```

We follow our usual steps. The abstraction is self-explanatory. The base case is also easy (a single element list). But the problem lies in the third step.

It seems that knowing s-Sum for for the rest of the list doesn't help on the reducing the problem to a simpler point.

It would still be most desirable to have some sort of "accumulator", something that registers a partial sum, a piece of information that "sums" up the elements before a certain point

Accumulator passing style

This is still possible. Such construct is so common that gets its own name. We call this *Accumulator passing style* (APS).

```
int helper(list_t remain, int index, int acc) {
    if (isEmpty(remain)) return acc;
    acc += index * index * list_first(remain);
    return helper(list_rest(remain), index + 1, acc);
}
list_t strangeSum(list_t list) {
    return helper(list, 0, 0);
}
```

How does this work?

The essential idea is to sum up the necessary information into two (could be more) accumulators. We essentially created a sort of running sum.

Accumulator passing style

We then further note the abstraction of the helper function. The function `helper` takes the index of the first element in the remainder list and a partial sum of the elements before, and returns the s-sum of the entire list.
In this way we transform our original problem of finding out `helper(list, 0, 0)`. On each recurse call, we extract the first element, and use it to update our accumulators, and passed that on to the next call.

```
helper([6  5  3  8  -1  7  3], 0, 0)
helper([5  3  8  -1  7  3], 1, 0)
helper([3  8  -1  7  3], 2, 5)
helper([8  -1  7  3], 3, 17)
helper([-1  7  3], 4, 89)
helper([7  3], 5, 73)
helper([3], 6, 248) = helper([], 7, 356) :=356
```

Remarks

Understand APS in a broad sense

APS is extremely useful. In many sense this technique is very similar to loops, which you might feel more comfortable to deal with. On the other hand an accumulator can be more than just an sum of numbers. It could be any information you need to keep track of (for example, if the number before forms an arithmetic sequence, and if they do, what is the increment).

Recursion and correctness

It's extremely difficult to write correct code! It would be nice if we can formally prove that our code is correct. Since the usual procedural code involves state, this proof can be very complex. But if you express the idea using abstraction, proving correctness is very simple and forward. A good recursion construction is almost always correct, and you can prove it! Write once and be free of testing and bug. What a nice thing!

RC Week 6

Engineering correctness: Testing

The definitive correctness myth

We quote the following words from one of yours (@LukeXuan)

> *While testing did indeed help your code to behave correctly in most cases. It never gives full assurance. I think you should recommend the technology of formal methods, especially verification, to introduce the possibility of complete correctness of program to students.*

Despite the obvious taunt in the words, these comment DOES speaks some, truth, that is:

> It is fundamentally impossible, proved in theory, to guarantee the absolute *correctness* of the program by simply testing it.

After all testing is an attempt to engineer correctness. It is an engineer method that aims at decreasing chances of software malfunction in the field, i.e. reliability.

Two general strategies in testing

Black box testing

Treat your program under testing as a "blackbox". The tester cares only about input and output. Essentially our OJ does black-box testing.

Glass box testing

Tester designs the test case according to the case. Test-cases are designed in such way that attempts to

- Achieve full coverage (Activate every branch once).
- Touch boundaries, base cases, or data-type boundaries.
- Stress the implementation, or exploit it for security reasons.

Testing is always an active activity, even for black box testing. Test cases are always designed with the technicals in heart.

Input Partitioning

The purpose of the testing, in most common cases, is to reveal possible defects, be trying to pick representative inputs.
The basic logic in designing test-cases is

If this program works on X, so it should work on Y.

The job of the tester is often to categorize all possible inputs into different *equivalent classes* (if you still remember that term from VE203). For each equivalent class we pick a few inputs and assume if the function works for those input, it should work for all inputs within that class.

Remember, again, testing is an creative process that relies on your understanding of both the problem and implementation at hand.

It is never an easy job to partition the input right.

Program under testing

The following program takes a string (of less than 100 characters) and decides whether it is palindromic recursively:

```
bool isPalindrome(const char* str, int size) {
    if (size == 0) return true;
    if (size == 1) return true;
    if (str[0] != str[size - 1]) return false;
    return isPalindrome(++str, size - 2);
}
int main() {
    char str[100]; cin >> str;
    int size = strlen(str);
    cout << isPalindrome(str, size);
}
```

Test Cases Designing: "Normal Input"

The most common kind of test cases are "Normal Inputs". Normal inputs are considered normal in the following sense:

- They are normal in range.
- They are normally constructed, i.e. are not delicated constructed to sabatotage / overload the program.
- They cover most normal outputs.

For our previous example some good test cases will be:

- `"12321"` Odd size palindrome
- `"1221"` Even size palindrome
- `"1222"` Even size non-palindrome
- `"1234345"` Odd size non-palindrome

Test Cases Designing: "Boundary Input"

Boundary cases are those input that pushes the program to the "boundary":

- They are base case in recursion.
- They pushes program to the edge of used datatype range.
- Any point that is "tricky". For example in a gcd program you should remember to test the input where one argument is a multiple of another. Any input that will trigger a special treatment.

For our previous example some good test cases will be:

- `""` Empty string
- `"1"` Single character string
- `123...321` 99 characters string, why 99?
- `"11"` Odd number (possibly) base case

Test Cases Designing: "Random / Malicious input"

Those are inputs that does not make sense. You normally don't expect your users to supply such arguments, but often technically they can.
For example you won't expect the user to input `I love lemonade` in a text box labeled *What is your social security number?*. But technically your program can receive such input.
It is often these situation that brings about the most trouble.
These input can potentially crash the program. Or worse, construct special input that corrupts / extracts confidential data.
The conclusion is: Never trust your user, not a single byte!

- Random input, random bytes...
- Malicious input.
- Assume that your user will not listen to your warnings. E.g. input more than 10 characters when prompted `Input a string of less than 10 characters :`.

Test Cases Designing: Design with abstraction

It is very important to keep the abstraction, or specification in general in mind when trying to design test cases.

- The specification specifies the what inputs are "normal", i.e. what are the inputs that fits "REQUIRES" clause.
- The specification specifies the expected output.
- The specification often defines boundaries, the one value that divides valid input with invalid inputs.
- The specification often suggests program load in real world.
- The specification tells what inputs are considered "invalid".

There is one more thing, the "MODIFIES" clause. Often the code under testing produce (or relies on) side-effects. The "MODIFIES" clause specify these things.

Again we emphasize the importance of creating a clear abstraction!

A QA (Quality Assessment) engineer walks into a bar

The following is adapted from the work of *Bill Sempf*'s twitter. A

QA engineer walks into a bar

- Then he orders 0 beers.
- Then he orders 2 beers.
- Then he orders 999999999999 beers.
- Then he orders an `aardvark`.
- Then he orders `nullptr`.
- Then he orders -1 beers.
- Then he orders `3.14` beers.
- Then he orders `asnwikfjsdf`.
- Then he orders `><script>giveMeCreditCard()</script>`.
- Finally, the QA engineer leaves without paying, comes back, and asks for the tab.

Unit test, *Regression test* and *Integration Test*

Rome is not built over night. So are softwares.

We now introduce you to some software engineering terms.

- Often software are first designed by an architect, partitioned into *moduless*.
- Programmers code each module independently. They write `Unit Tests` for each module.
- When the modules are put together, architect team creates *Integration tests*
- The collection of test cases are called a *Test Suite*
- The team will keep updating the software. After each change, a test suite a will be run to ensure the change didn't happen to break anything. This is called a *Regression Test*

Test automation

From you own experience:

- Testing is actually pretty common task.
- It takes time to create driver programs to run the tests.
- It takes time (and code) to create test cases. You often need to manually calculate the expected output.
- It takes time (and code) to analyze test results. To keep track what goes wrong, especially you.

This calls for *Test Automation* and *Testing Frameworks*, a testing framework is (often) a piece of library that does the following:

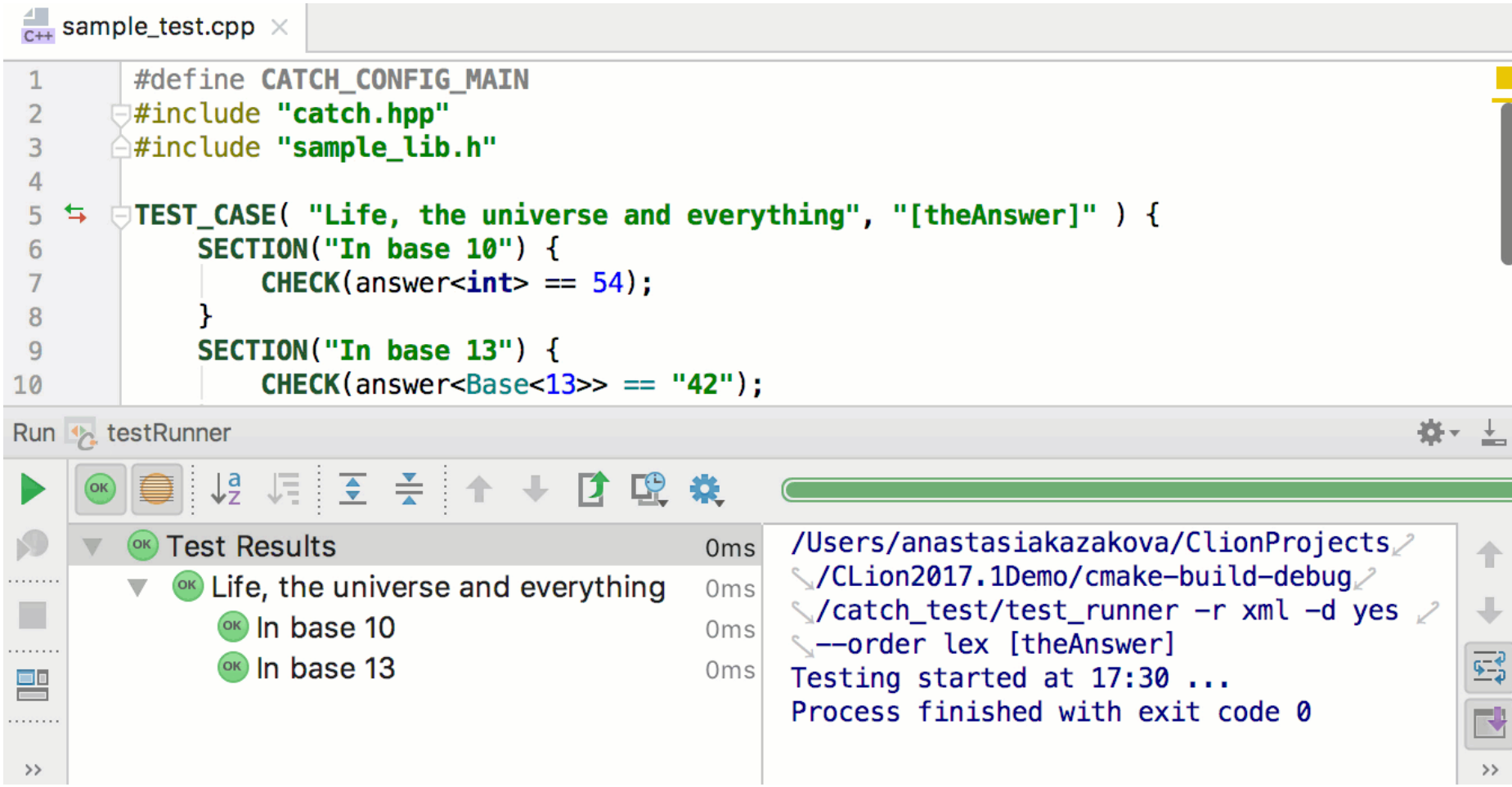
- Runs test cases automatically.
- Manages the test cases, selects what to run and what not.
- Automatically sets-up the test environment (test fixtures).
- Automatically keeps track of what's OK and what goes wrong.
- And much more...

Google Test framework

Project Homepage: `https://github.com/google/googletest`
Introduction is available if you click here
Here is a portion of code I used to test my VE281 project:

```
TEST_P(SelectionTest, DSelection) {
    int *d= dataset->getCopied();
    int size = dataset->getSize();
    for (int i = 0; i < size; ++i) {
        int val = deterministicSelection(d, size, i);
        int answer = dataset->select(i);
        ASSERT_EQ(val, answer);
        delete[] d;
        d = dataset->getCopied();
    }
    delete[] d;
}
```

Google Test in CLion



RC Week 6

Engineering robustness: Exceptions

Breaking the abstraction

The central question that goes around with exceptions are:

What if assumptions of an abstraction is broken?

Note that this should be understand in a broader sense. Any program runs under some assumption. For example,

- There is enough system memory for your program.
- There is enough space when you need to create a file.
- Input outside `REQUIRES` clause never happens.
- Your computer have an available Internet connections.
- ...

All these things constitutes the assumption you made about your abstractions. But it certainly could happen that one (or more) of them are broken. This could due to hardware limitations, or more likely due to an error in programming.

Fail-fast & "I give up."

There is absolutely no point to save a flawed processess. This is called the fail-fast fast.

- The program is already in an non-recoverable state
- It is probably due to a programming error
- Error might propagate, crash site far from source.
- May corrupt data. Programs can be fixed, data can't.
- Best strategy is to quit gracefully.

A typical situation:

```
void foo() {
        int *p = malloc(sizeof(int) * 10);
        // This should always success unless lacking memor
        assert(!p);
        // Do something with p
        free(p);
}
```

"It's my problem."

The attempt is to make the function essentially a "total" function. Essentially this strategy says

"Invalid inputs are part of my abstraction"

Which also implies testing for invalid inputs! An (not so good) example:

```
// Checks if all letters ina string are capital
bool isAllCapital(char* str, int size) {
    int len = strlen(str);
    if (size != len + 1) len = size; // Input validation
    for (int i = 0; i <= len; i++)
        if (*str < 'A' || *str > 'Z')
            return false;
    return true;
}
```

"It's my problem."

There are some serious problem with this approach:

- Function might not have well defined "default" value.
- It might hard to guess a "default" value.

Regardless above problem, much more serious problem comes with error propagation. There is probably a reason why this input is valid. It's most likely because you code is incorrect (in some sense) or your running environment is problematic (stack overflowed, for example).
You need to understand whenever you took this approach, you are trying to fix an already "broken" program, going against the fail-fast principal.

- You could end up crashing somewhere far from the root cause.
- Your program could behave wired. Since your abstraction includes treatment of "special cases"

"It's not my problem"

The problem of above approaches is significant because they are trying to deal with errors that does not come from themselves.
The root cause of these invalid inputs are somewhere else, probably only known by their caller.
The natural idea would be to find a method to pass the information up the chain of calling, a natural way of doing so is by return *Error Codes*.
There are in general two ways to do so:

```
// Returns negative value if error
int fact1(int n);
// Returns value signifies error, zero indicates no error
// Actual result is put into *rst
int fact2(int n, int* rst);
```

In some cases libraries use a global variable to store the error code of last function call.

Problem with error codes

Both methods have severe draw back:

```
double tan(double rad);
```

Function `fun` could return anything in `double`, now what should you use for error code?

```
int foo(double rad) {
    double rst = 0.0;int err = tan(rad, &rst);
}
```

The second makes calling "unnatural. Also performance drawbacks.

- Error code can be ignored. The worse thing than crashing on errors is an unattended error
- Error code needs to be passed up the stream. Sometimes the direct caller also don't have a clue, it needs to pass it on.
- Breaks abstraction.

Structural Error handling: `try`, `catch` and `throw`

We know introduce you to *Structural Error Handling*. The following is a try block

```
// (1)
try {
    // (2) (may throw;)
} catch (const T& e) {
    // (3) ...
}
// (4)
```

Just like usually `if`, `while` blocks, a `try` block marks a special control flow, featuring (usually) 2 different code execution paths.

- Path 1. Exception raised in (2), we go (1) (2, partial) (3) (4).
- Path 2. Normally we go (1) (2) (4).

Path 1 assumes the exception is caught. Further discussion on next slide.

Structural Error handling: `try`, `catch` and `throw`

An concrete example.

```
try {
    cout << "hello!" << endl; throw 123;
    cout << "Goodbye" << endl;
} catch (int x) {
   cout << "Integer Exception";
}
```

Prints `"Hello//Interger Exception"`

- A `throw` clause raise an exception (of arbitrary type).
- When an exception is raise, immediate stop the following execution and go for smallest enclosing `try` block.
- If there isn't one, terminates program.
- If there is one, begin matching `catch` clauses.
- If found, we say the error is *handled*, begin executing after `try`. If not, terminates program.

Structural Error handling: try, catch and throw

A try block can be matched no matter how deep in the function. The following example is just for demonstration! It is NOT a good practice!

```
int foo(int n, int prod) {
    if (n == 0) throw prod;
    foo (n - 1, n * prod);
}
int realFact(int n) {
    try {
        foo(n , 1); cout << "LaLaLa";
    } catch (const int& x) {
        cout << "Aloha!" << endl;
        return x;
    }
} // Prints Only "Aloha!".
```

Structural Error handling: `try`, `catch` and `throw`

An exception cannnot be overlooked! Unhandeled exception terminates the program.

```
int fact(int n) {
    if (n < 0) throw n;
        if (n == 0) return 1;
        return n * fact(n - 1);
}
int main() { int x = fact(-50); }
```

Note we use the word "Teriminate". There is an difference in *crashing* and *terminating*.
The first term indicates the program is stopped by force (by the operating system), immediately.
The second one indicates a seriers events will happen (for example, clearing the buffer of `cout`). The program terminates (somewhat) gracefully.

Structural Error handling: try, catch and throw

The smallest enclosing try block gets to handle the exception

```
void foo() {throw -1;}
void bar() {
    try{ foo(); }catch(double e){cout << "bar!";}
    cout << "Slotted Aloha!";
}
void rua() {
    try{ bar(); }catch(int e){cout << "rua!";}
    cout << "Alright!";
}
void baz() {
    try{ rua(); }catch(int e){cout << "baz!";}}
int main() {baz();}
```

Above programs prints rua!Alright!. It terminates normally.

Structural Error handling: `try`, `catch` and `throw`

An exception can be rethrown after being caught in a `catch` block. This is useful since you might need clean up, although you don't known how to deal with the error. It is also possible to throw a different exception.

```
void foo() {throw -1;}
void bar() {
    int *p = new int(10);
    try{ foo(); }
    catch(...){delete p; cout << "bar!"; throw; }}
void rua() {
    try{ bar(); }catch(int e){cout << "rua!"; throw 1.0;}}
int main() {rua();}
```

Above programs prints `bar!rua!`. It terminates due to an unhandled exception `1.0`.

Rules for matching for `catch`

The rules for matching `catch` is given as follows:

1. First found first match.
2. Match only with exact same type. If the exception is an class instance, also matches with it's base catch.
3. `catch(...)` matches everything.

```
try{int x = 1; throw x;}
catch(long int li) {
    // No match
} catch (int x) {
    // Match here
} catch (...) {
    // Already matched
}
```

```
try{int x = 1; throw x;}
catch(long int li) {
    // No match
} catch (double x) {
    // No match
} catch (...) {
    // Matched
}
```

Exception Hierarchy

The fact that exception instances can be caught by their base class type is actually very useful in reality. You can catch a "class" of exceptions while leaving others un touched.

```
class Exception {};
class IntegerExecption : public Exception {};
class FileExecption    : public Exception {};
class FileNotFound     : public FileExecption {};
class PermissionDenied : public FileExecption {};
void doSomethingToFile(string filename);
void foo(int n) {
    string file = "str" + to_string(n);
    try { doSomethingToFile(file) }
    catch (const FileExecption& e) {
            cout << "Something wrong with File";
} }
```

Exception Hierarchy in standard library

std::exception

Defined in header <exception>

```
class exception;
```

Provides consistent interface to handle errors through the throw expression.

All exceptions generated by the standard library inherit from **std::exception**

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error(C++11)
  - bad_optional_access(C++17)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error(C++11)
  - tx_exception(TM TS)
  - system_error(C++11)
    - ios_base::failure(C++11)
    - filesystem::filesystem_error(C++17)
- bad_typeid

Tips

1. Throw by value and catch by const reference.
2. `throw` only on real exceptions

Throw by value

When an exception is raised, the programming is doing sort of recovery. The function that throws the exceptions most likely is going to terminate. Thus throwing reference (or address) to local objects won't make sense at the handling site.

Catch by const reference

The very reason that you need to catch exceptions by reference is simply because exceptions can contain virtual functions.

RC Week 7

IO

Buffered Input streams

The only input stream is `cin`, the standard input stream. The input stream is buffered.

1. When you hit keyboard, the OS receives the keyboard key stroke and buffers what you type.
2. You can always modify your input at this point.
3. When you hit enter (or CTRL-C, etc.), OS sends the inputs to your program.
4. Your program itself again buffers the input it receives.
5. Read from your programs' buffer when you extract from `cin`.

It is somewhat useful to know that, when you hit enter, not only the your previous inputs are sent to the program, but also the "enter" keystroke it self. What is the ASCII character for the "enter" key?

If you are to write interactive programs, e.g. games that responds to direction keys, you will have to work with the underlying operating systems.

How does input stream work?

If we are to understand the behavior of input streams, we must turn to its internal workings. The input stream follows a 3 step procedure

1. Read the input stream character by character.
2. Based on the read characters, and target stop at some point.
3. Convert what you read into required type.

We demonstrate on an example. Suppose you try to execute

```
string stt; cin >> str;  // cin contains "123abc hello wor
```

- Read the input stream until it hits a blank character.
- The blank character remains in the stream.
- Trim the string, remove all leading blanks.
- Copies the result into `str`.
- The stream is left with `hello world`, 1 leading space.

How does input stream work? Cont'

The following is a summary of usually behaviors of common behavior extraction operators. **based on my memory**. Suppose:

`T var; cin >> var;`

- If `T` is numeric types, such as `int`. 1) First character might be the sign. 2) Reads until the first non-digit character 3) The non-digit character is left in the stream.
- If `T` is `char`. It extracts exactly one character from the stream, no matter what that character is. Equivalent to `getch()`.
- if `T` is `char*`. It do exactly as the string. In the final step it copies the characters read to where the pointer points. *It is dangerous because of buffer overflow.*

```
istream& getline(istream& is, std::string& str);
```

This function 1) Reads from the stream `is` 2) Stops until a newline character. 3) The encountered new line character is **extracted out, but ditched** (not copied into `str`)

Avoid parsing whenever possible!

Specifically, avoid using `getline()` and `getch()` whenever possible. The C++ extractors has a very nice property: if you try to extract numbers or strings, the extractor automatically ignores blank characters. This means you only need to focus on the question of

WHAT is the next thing I need?

instead of thinking

- WHERE is the next thing I need?
- Is what I need on the next line?
- Is there a space or a tab before what I need?
- How do I get rid of that space / tab / newline?

The failed state

If you try to extract invalid things from the stream, the stream enters a so called *Failed State*.
Two typical situation where a input stream will enter a fail state is when you try to extract things from an empty stream (all things in the stream has be extracted). Or you extracted the wrong type.
In general the second situation should be avoided. If you are not sure what next thing is, use a string to accept it. Transform the string into expected type later.
You can test whether a stream is in a valid state (very often if there is still thing remains in the stream) by simply `if(stream)`.
A very commonly used idiom to extract everything is:

```
int num; while(cin >> num) /* Do something here */;
```

Please note that extractor operator returns an reference of the left hand stream.

Buffered Output Streams

There are 2 output streams `cout` and `cerr`

- `cout` is the standard output stream. It is buffered.
- `cerrr` is the standard error stream. It is NOT buffered.

File outputs are almost always buffered. If an output stream is buffered, it usually works as follows.

- Your program stores the output in a buffer.
- When the buffer is full, or explicitly flushed, your program sends the buffer content to OS.
- The OS puts the content into a file, or onto the screen (which is also a "file", remember?).
- Inserting a `std::flush`, or `std::endl` flushes the buffer.

Standard output streams accepts normal program outputs. On the other hand, `cerr` accepts error messages, **warnings** and **diagnostic messages**.

Buffer, flush and performance

Buffering have a few implications, or more likely problems.

- If your program crashed at some point, information in the buffer will be lost and thus not print out (for this reason `cerr` is not buffered).
- There is an extra overhead of maintaining the consistence of the C++ style streamed IO with C style `printf`, `puts`...

And there's more to it. Although `cout` and `cerr` are two different streams, they usually both goes to the stream. If you output to both of them, for example

```
cout << "Hello "; cerr << " Gotcha "; cout << "World!"
```

It is possible you see any one the following three (why?): 1) `Hello Gotcha World!` 2) `Gotcha Hello World?` 3) `Hello World! Gotcha`. What to do if I want to ensure the order?

Buffer, flush and performance, Cont'

Why do we need such buffer? If it causes so many problems?
The reason is actually performance, flushing the buffer, or printing things onto the screen requires an interaction with the operating system, which is very expensive!
Buffer is an attempt reduce the number of such interaction by trying to send as much text as possible out at once. Compare:

```
for (int i=0; i<10000; i++) cout << "Hello" << endl;
for (int i=0; i<10000; i++) cout << "Hello" << "\n";
```

Generally the first one could be at least 3 times slower than the second one. This will be significant in the runtime of your program since your projects are often IO heavy (outputs a lot).

Type signature of extraction / insertion operators

The reason why C++ iostream can work seamlessly with various types, is because it uses operator overloading and inheritance. The type signature of the extraction operator (often) has the following type (general) signature.

```
istream& operator>>(istream& is, T& var);
ostream& operator<<(ostream& os, const T& var);
```

The standard library provides the following for `std::strings`.

```
istream& operator>>(istream&, std::string&);
ostream& operator<<(ostream&, const std::string&);
```

For smaller types, insertion operator may accept pass-by-values, not const references.

```
istream& operator>>(istream& is, int& var);
ostream& operator<<(ostream& os, int var);
```

Type signature of extraction / insertion operators, cont'

```
istream& operator>>(istream& is, T& var);
ostream& operator<<(ostream& os, const T& var);
```

- Each type that supports extraction has its own overloading of the function.
- All sources that supports character-by-character reading inherits from `istream`. For example, the input file stream `ifstream`.
- For extraction operator, variable is passed by reference, allows putting the value "into" the variable.
- For insertion operator, variable is passed often by **const reference**. Why?
- The operator returns first argument as its return value. This allows "chaining" the extraction / insertion.
- Functions like `getline()` also follows this convention.

Chaining extraction / insertion operator

A C++ beginner always quickly get used to write things like:

```
int x, y; char c1, c2; std::string str;
getline(cin >> x >> c1, str) >> c2 >> y;
```

The syntax is very concise compared to C (`fscanf()`). The question is, how does it work, without specifying the format string? The secret is mainly lies in the type signature of extraction operators, specifically:

- Extraction (insertion) operators are left associative, meaning parentheses are grouped left to right.
- Compiler chooses overloads according to the second argument. This avoids the `scanf()` style format string.
- Operators return a reference to the stream.

Remark: this example demonstrates what is known as *static polymorphism*, where the identical code is resolved to different function calls at compile time.

Chaining extraction / insertion operator, Cont'

We apply parentheses to previous example:

```
(getline(((cin >> x) >> c1), str) >> c2 )>> y;
```

The first thing to evaluate the is the argument of `getline(...)`. Note that `cin` is of type `istream`, `x` is of type int. The compiler chooses overload:

```
istream& operator>>(istream& is, int& var);
```

It takes `cin` as `is`, `x` as `var`. After extracting one integer from `cin` it again returns the reference of `cin`.

```
((...) /* Evaluates to cin */ >> c1)
```

This allows the compiler to deduce the second overload for `c1`.

```
istream& operator>>(istream& is, char& var);
```

This goes recursively on and on. It should now also be clear why `getline()` should return a reference to its first argument.

Writing a tree to standard output

This is a code snippet from the VE280 Online judge code that write a tree to standard output. A tree is can be written out in the form `[ tree_elt [{left_tree}] [{right_tree}] ]`. For example `[ 2 [ 1 $ [ 3 $ $ ] ] [ 1 [ 4 $ $ ] [ 6 $ $ ] ] ]`. The `$` represents empty tree.
File: `code/rc7writetree/writetree.cpp`

```
std::ostream&
operator<<(std::ostream &stream, const tree_t &tree) {
    if (tree_isEmpty(tree)) {
        stream << "$ ";
        return stream; }
    stream << "[ " << tree_elt(tree) << " ";
    stream << tree_left(tree) << "";
    stream << tree_right(tree) << "] ";
    return stream; }
```

Tips on working with file streams

File streams are very much like `cin` or `cout`. Please note the difference of `ifstream` and `ofstream` and `fstream`.
File streams can be opened on construction:

```
ifstream inputFileStream("input.txt");
```

Above code opens `input.txt` with a `ifstream`. Please note if the file doesn't exist or failed to open, your `inputFileStream` will be in a failed state. Remember to check!
Although you can use `inputFileSteam.close()` to close the stream (and clear its buffer), but you don't actually need to.
Remember the stack unwinding example we have before? Standard libraries' file stream are exactly such objects. Its deconstructor will take care of the closing and freeing the file.

Use `istringstream` as parser

A `stringstream` can either act be an input stream, or an output stream. They are very different!
`istringstream` needs to be "attached" to a string. This can be down by either passing the string in on construction or use its `.str()` method. The stringstream, unlike `cin`, doesn't consume the string, instead it attach a "pointer" to the string. Ideally:

```
123 abc def Hello world  |  123 abc def Hello world
|<-sstream               |     |<-sstream
```

Right side is after extracting 123 through `sstream >> intVar;`
`istringstream` is especially useful in treating data acquired from `getline`.
If you need to use the same `istringstream` object for multiple strings, you need to clear it when changing string.

Use ostringstream as string builder

From time to time we need to convert different type of data into a string. For example, information about a student is described by:

```
strcut Student {
string name; int grade; Date birthday; double GPA;
};
```

And you need to turn such stuctures into below format

```
%name, %grade grade, %YYYY-MM-DD, gpa = %gpa"
```

You can setup an ostringstream. You simply push into the stream as if it is cout, finally you call ostringstream.str() to get the result.

```
oss << name << "," << grade << " grade" << ...
```

The design of the stream makes it very efficient in dealing with large amount of text. For example you can use it to prepare data to send over the Internet.

RC Week 7

Data Abstraction with `classes`

Once another look on data types: Duck typing

Consider the problem:

Define what a "Duck" is?

The very answer from *Alex Martelli* is:

> *In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.*

The whole purpose of having data type is to model things. We use `struct` to model what something consists. But things are only useful when we interact with them. The *observable* effects are all we care. The whole point of data abstraction is to model the behavior of objects.

Abstract Data type

The central idea is:

Data type IS Abstraction.

Always define a datatype in terms of what it can do, what kind of operations are supported for this datatype, and there should be nothing more. Also define what you CANNOT do with such data.

Information Hiding Actual implementation of the object is hidden away, the outside can't see, and shouldn't see what's inside.

Encapsulation Operations become part of the type.

Locality Other components only depend on exposed operations (i.e. interfaces), nothing more. This is different for `struct`.

Substitutable Change the implementable preserves program correctness.

Designing ADTs with classes

We would like specify an Integer class:

```
class Natural {
// OVERVIEW: An N
public:
void  set(int v);
// EFFECT:  ...
// MODIFIES: this
Natural  add(Natural v);
Natural  mul(Natural v);
int get();
private:
int value;
};
Natrual num;
```

- `class` key word begins the specification of an abstract datatype. CamelCase style recommand you to capitalize first character.
- `public` key words begins specifying the *operations*, the *abstaction*, or we say the *interface*. Pay attention to the `MODIFIES: this`.
- `private` key words calls for actual implementation.
- Note what's not there: a `div` method. Dividing natural numbers usually don't give you natural numbers.
- `num` is called an *object*. `num` is an *instance* of `Natural`.

Implementing classes

We now need to try to implement a class. We focus especially implementing class methods (member functions are usually called *methods*.).

The declaration of a class is usually written in a *header file*. For the example of last page we usually implement it in `natural.h`. The name of the file usually is the same as the class you are implement.

Class methods can be implemented either when you declare the class, or in a separate `.cpp` file. For our above example we usually implement it in the `natural.cpp`.

Whenever you need to use a class, you include its header file. Usually each header file only contains definition of one single class, unless you have a couple of tightly related class.

Implementing classes

```
// Natural.h
#ifndef __NATURAL_H__
#defein __NATURAL_H__
class Natural {
public:
void  set(int v) {
this->value = v;
}
Natural  add(Natural v);
Natural  mul(Natural v);
int get() {return value;}
private:
int value;
};
#endif __NATURAL_H__
```

```
// Natural.cpp
#include "natural.h"
Natural
Natural::add(Natural v) {
Natural n;
n.set(value + v.get());
return n;
}

Natural
Natural::mul(Natural v) {
Natural n;
n.set(value * v.get());
return n;
}
```

A remark on private

`private` means private to class, not to instance.

```
class Human {
int leg;
public:
int swapLeg(Human& h) {
swap(this->leg, h.leg);
}
int setLeg(int n) {
this->leg = n;
}
};
Human h1; h1.setLeg(10);
Human h2; h2.setLeg(3);
h1.swapLeg(h2);
```

- Code on the left compiles and runs without problem.
- It's sometimes called "Since we are both human why can't I swap my leg to yours" effect.
- This indicates the design choice that `private` is a technique to hide abstraction implementation, other than to prevent outside modification.
- Members are `private` by defualt in a `class`. Recall that `struct` are just `class` whose member are by default `public`

this keyword

```
class Natural {
int value;
public:
int get() {
return value;
}
void set(int value) {
this->value = value;
}
void printThis() {
cout << this << endl;
}
// Omit other method
};
```

```
int main() {
Natrual n1;
n1.printThis();
cout << &n1 << endl;
Natrual n2;
n2.printThis();
cout << &n2 << endl;
}
```

One possible output:

```
0x7ffff1fd2820
0x7ffff1fd2820
0x7ffff1fd2830
0x7ffff1fd2830
```

Source under `code/rc8this`

this keyword

```
class Natural {
int value;
public:
int get() {
return value;
}
void set(int value) {
this->value = value;
}
void printThis() {
cout << this << endl;
}
// Omit other methods
};
```

- Non-static methods always have a `this` pointer available.
- For non-const member methods `this` keyword is of type `ClassType*`.
- Members (including methods) are always accesses through `this` keyword, either explicitly (in `set()` method), or implicitly (`get()` method).
- If there is an ambiguity you can use `this` to resolve ambiguity. In the `set` method, LHS refers to member and RHS refers to the argument.

const member function and ripple effect

A member function can be declared as const to signify it will not change any member (in any possible way). The declaration must be done both in the declaration and in the implementation:

```
// useless_class.h
class UselessClass {
int value;
public:
int get() const;
};
```

```
// useless_class.cpp
int UselessClass::get() const {
return value;
}
```

- const methods cannot modify any member
- const methods may only invoke other const methods.

You already know the basic rules of const methods. But unfortunately the situation (you will be facing) is much complicated (when you get to STLs). We now make a further discussion.

const member function and propagation

To illustrate the problem, we here uses a functionality called *Runtime Type Info (RTTI)* to let the compiler spit out the types.

* code/rc8const/main.cpp

```
#include <typeinfo>
#include <iostream>
using namespace std; // BAD PRACTICE!
class Foo {
  int value;
public:
  void bar() const {
    cout << typeid(this).name() << endl;
    cout << typeid(&value).name() << endl;
  }
  void baz() {
    cout << typeid(this).name() << endl;
    cout << typeid(&value).name() << endl;
  }
};
int main() { Foo x; x.bar(); x.baz(); }
```

const member function and ripple effect

We take down (and translate for you) the output of the program

`bar()::this` const pointer to `Foo`

`bar()::value` const pointer to `int`

`baz()::this` pointer to `Foo`

`baz()::value` pointer to `int`

Essentially we have the following 3 rules:

- In a `const` method, `this` have type `const Class*`
- Non `const` member functions are not allowed to be accessed from a `const` object.
- Any member variable accessed through a `consted` `this` keyword is automatically const quantified.

We provide some examples to explain them.

const member function and ripple effect

We take down (and translate for you) the output of the program

`bar()::this` pointer to `const Foo`

`bar()::value` pointer to `const int`

`baz()::this` pointer to `Foo`

`baz()::value` pointer to `int`

Essentially we have the following 3 rules:

- In a `const` method, `this` have type `const Class*`
- Non `const` member functions are not allowed to be accessed from a `const` object.
- Any member variable accessed through a `consted this` keyword is automatically const quantified.

They seems very abstract. We provide some examples to explain them.

const ripple effect: Example

Consider the following 2 classes

```
class Integer {
int value;
public:
int get();
void set(int i);
};
```

```
class Bar {
Integer i;  Complicated c;
void doComp(); // uses c
public:
int getValue() {
doComp(); return i.get();
} };
```

Suppose there exists a function that originally looks like:

```
void baz(Bar bar); // Do complicated work
```

The function passes `Bar` by value. It was alright, until bar becomes really large. A quick fix is proposed. Somehow you know function does not modify the argument, you suggest to change it to

```
void baz(const Bar& bar); // Do complicated work
```

Pretty harmless change? What could be possibly go wrong?

const ripple effect: Example

Well it isn't! Let's see what happens:

- We know `baz` calls `getValue` from `Bar` class. Since we change the argument to const reference. We can no long do that, since you cannot call a non-const member function on a const object.
- You change `Bar::getValue()` to const member function. But this gives you 2 more problems (what are they?).
- You tackle the easier one. Since `i` in `Bar::getValue()` is now const object, you have to change `Integer::get()` also to const member function.
- Another problem is `doComp()` is not const. This function does really complicated calculation with `i` and `c`.
- You have no choice but to mark `Bar::doComp()` as const. This forces you inspect every member function of `Complicated` class, and change them to `const`.
- Now you realize this will never end! There are simply more and more things to change. You ends up rewriting the entire code base!

const ripple effect: Example

Things you observed is sometimes called the "ripple effect" of const keyword: when you change X, you ends up changing all things related to X, and this goes on and const ends up "infecting" everything.

There are in general 2 ways to stop this effect, both of them are dangerous and far from satisfaction.

- A const_cast. The programmer explicitly asks the compiler to stop checking constness and tell the compiler to trust him.
- Make a non-const copy of the original object. Pass that object along. This has a obvious performance drawback.

The lesson behind this is you constness is a all-or-nothing deal. You either do it from the begging till the end, and never at all. Constness is considered good practice. It provides very powerful assurance on correctness. So our advice is simple:

Do const, and do it in full scale!

Building source file containing classes

We now turn to ask the question how classes are compiled. This is important for the following two reasons:

- How classes are compiled are related to the memory representation of classes. Many design choices on classes are (or were) have something to do with this.
- You could encounter various strange problem in your project or in a real life situation. Knowing them helps to grasp the situation.

Historically C++ was derived from a language called *C With Classes*. One of the first C++ compiler will first translate C++ code into C code, then a C compiler compiles it into machine code.

We will notice that classes are (once were) merely C structures with some syntax sugar. We will also begin to notice how much trouble does the C linking strategy introduce into the language, and becomes prominent problem in its course of development.

On compiling classes

The code on the left will be compiled, as if, it was written in the form of the right hand side

```
class Clock {
int hour;
public:
void setHour(int h) {
hour = h;
}
int getHour() const {
return hour;
}
};
void foo() {
Clock clock;
clock.setHour(10);
clock.getHour();
}
```

```
typedef struct Clock_t {
int hour;
} Clock;
void clock_setHour
(Clock* this, int h) {
this->hour = h;
}
int clock_getHour
(const Clock* this) {
return this->hour;
}
void foo() {
Clock clock;
clock_setHour(&clock, 10);
clock_getHour(&clock);
}
```

On compiling classes

Now you should find the pattern:

- Classes are arranged in memory exactly like structures. We used to say "structures are just classes whose members are all public". On a memory point of view, it's the other way around.
- Methods are compiled into usual functions, with one extra argument, the `this` keyword.
- Dot operator (e.g. `clock.setHour`) are translated into usual function calls.
- All members are accessed through the this parameter. The reason is now obvious.
- `const` methods results in a `const this`.
- `static` methods (if you still remember what they are), are just methods that does not take the extra argument (thus they are shared among all instances).

On compiling classes: Linking

We first go over a few things we already know about the building process (and see which of them are violated!):

- Building process is mainly about compiling and linking
- Each source file compiles independently.
- Each entity must have one definition and one only (ODR).

We have looked at look at the linking stage.

Remember information about classes are "thrown away" in the compiling stage. When it comes to linking, we are only linking function calls together. Functions are all that we care.

Every class method will be turned into a function. We keep that in mind and see the following slides.

On linking classes: the setup

[code/rc8class/integer.h]

```
#ifndef _INTEGER_H_
#define _INTEGER_H_
class Integer {
    int value;
public:
    int get() const {
        return value;
    }
    void set(int v);
};
#endif
```

[code/rc8class/integer.cpp]

```
#include "integer.h"
void Integer::set(int v) {
    value = v;
}
```

[code/rc8class/inc.cpp]

```
#include "integer.h"
void inc(Integer& i) {
    int old = i.get();
    i.set(old + 1);
}
```

[code/rc8class/dec.cpp]

```
#include "integer.h"
void dec(Integer& i) {
    int old = i.get();
    i.set(old - 1);
}
```

On linking classes: the setup

We accompany above with the following driving program [code/rc8class/main.cpp]:

```
#include <iostream>
#include "integer.h"
using namespace std;
void dec(Integer&); void inc(Integer&);
int main() {
    Integer i; i.set(10); cout << i.get() << " ";
    inc(i); cout << i.get() << " ";
    dec(i); cout << i.get() << endl;
}
```

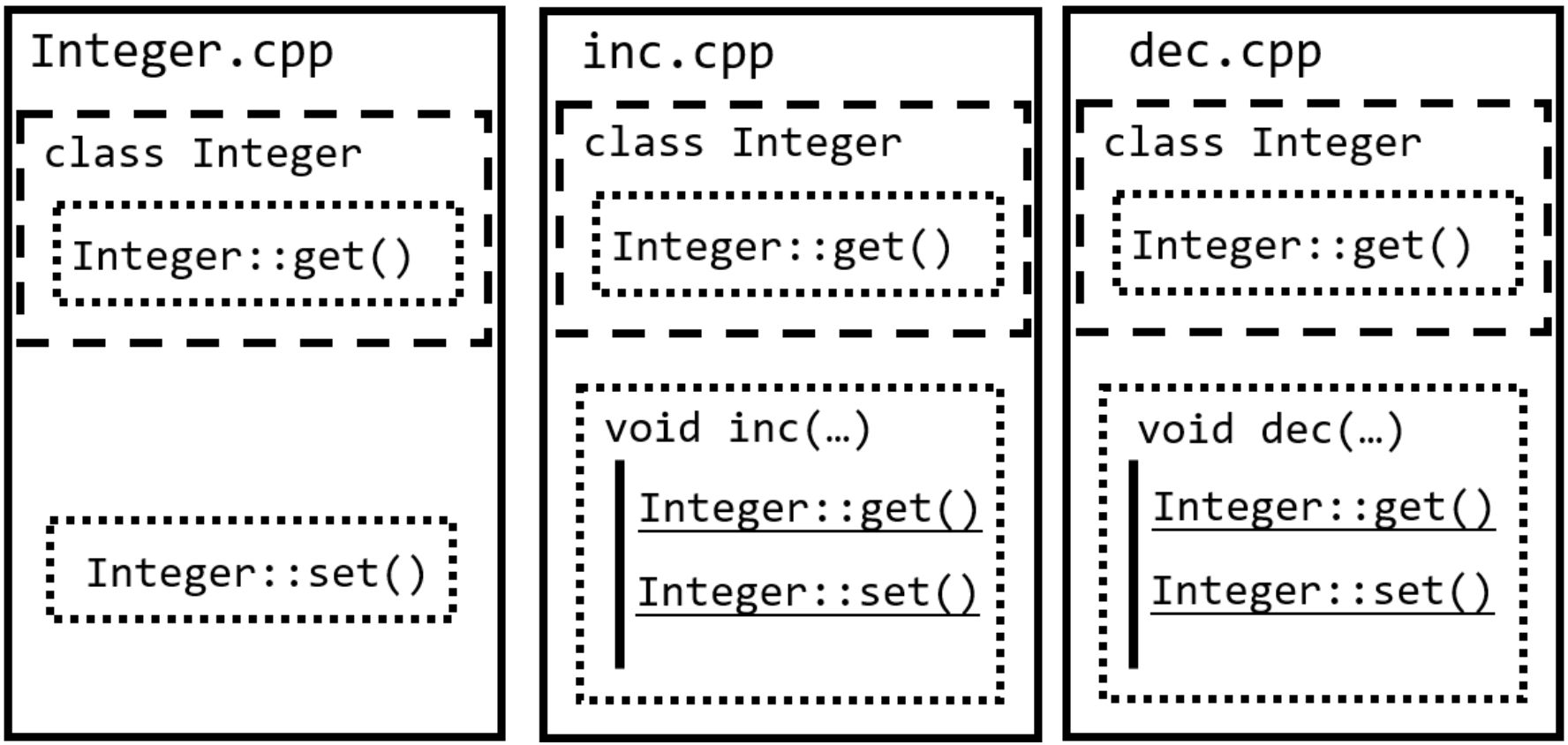
Compile the code and execute it:

```
g++ main.cpp dec.cpp inc.cpp integer.cpp && ./a.out
```

We observer the (not surprising) result `10 11 10`

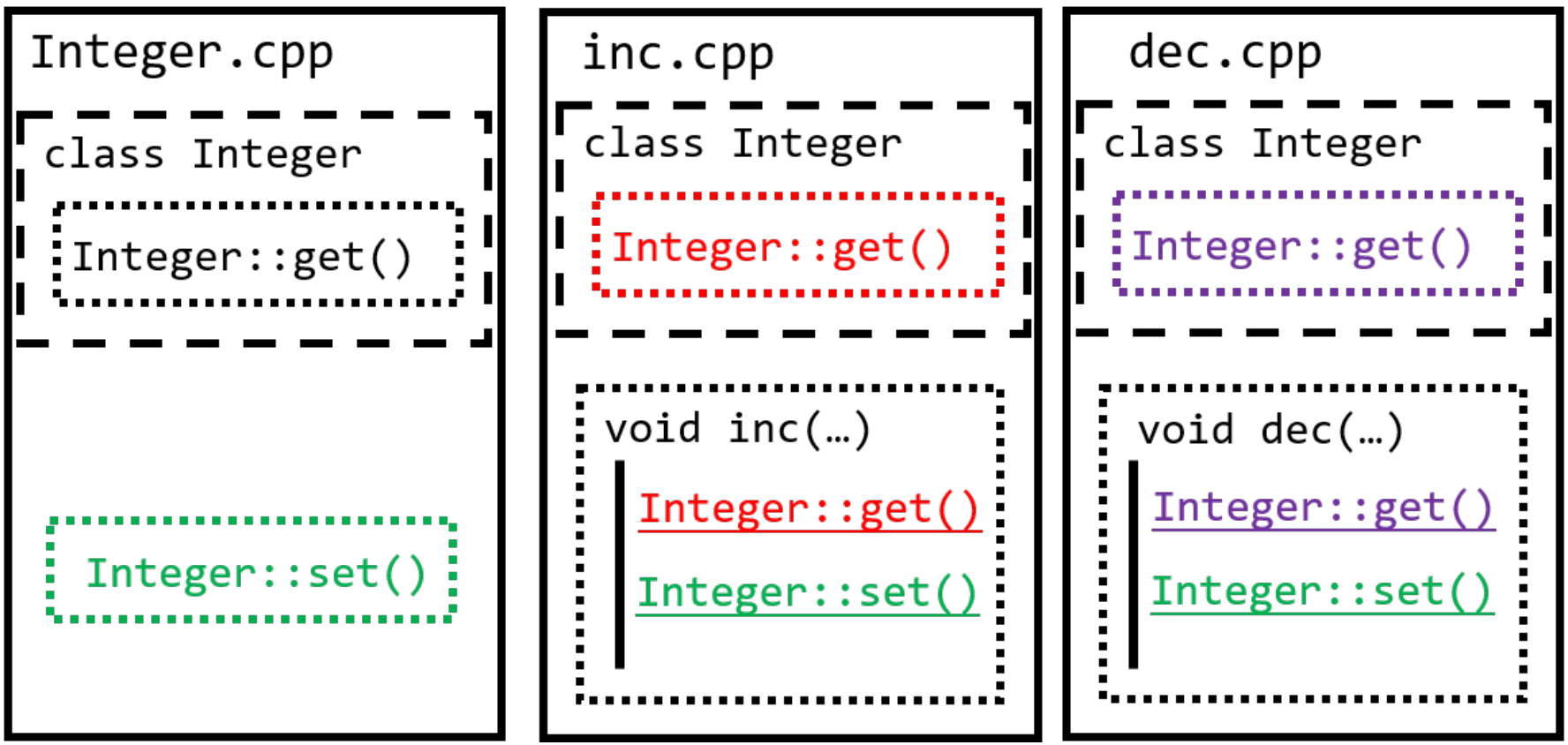
On linking classes

We now take a step back and think. In the preprocessing stage we already knows that files included by `#include` are copied into the file, this is (unfortunately) also true for `integer.h`:



On linking classes

Pairs of the same color are linked together:



On linking classes

Thus the guideline is very simple:

- Methods defined inside the class decls are linked internally.
- Methods defined in a separate `.cpp` source file are linked as if they are usual functions.

From a performance point of view, writing a definition of method inside the class declaration implies that the method is very simple. So instead of generating a function call whenever used the compiler will try to copy the code to each place it is used, avoiding the overhead of calling a function. This is called *inlining*.

A remark we haven't made yet is *abstraction always have an overhead*, a cost that we must endure to enable better flexibility. For classes, function calling mechanism is the overhead. Naturally people would want them both, i.e. "zero cost abstraction". The way C++ achieves it is through allowing compiler to heavily optimize your code, and inlining methods is one major technique.

RC Week 8

Working with class invariants

Invariants of a datatype

We now take a look at the implementation side. From a implementation point of view, what's the difference between a list of integers, and a set of integers?

- Both of them probably are represented using an array.
- Both of them might need to keep track of current number of elements.

The major difference lies in the fact that when inserting an element, integer set must check whether the element already exists. It is the assumptions on the member that makes them different.

Integer set has one more assumption than integer list. Each number in the array must be unique. Every operation of an integer set must start with this assumption, might broke it in the middle, but ends up with it.

We will use your `IntSet` example in class for our discussion.

Example of invariants

```
const int MAXELTS = 100;
class IntSet {
  int elts[MAXELTS], numElts;
  int indOf(int v) const;
public:
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int  size() const;
};
```

Invariants:
1) Numbers are arranged in `elts` sequentially from index zero.
2) Number of elements in array equals to the value of `numElts`.
3) Numbers are unique.

Guideline is:

- Methods assume inv.s on entry.
- (Public) Methods must preserve inv.s on the moment of exit.
- Nobody cares what happens in the middle!

This way we safely claim invariants are always preserved for an class instance from an external point of view.
Analyze the methods:

- `insert` might broke all three.
- `remove` might broke 1) and 2).
- `const` methods will not break anything! One more advantage!

Example of invariants: Implementing methods

```
1  int
2  IntSet::indOf(int v) const {
3    // Omit implementating
4    // Not found return MAXELTS
5  }
6  void IntSet::insert(int v) {
7    if (indOf(v) != MAXELTS) {
8      assert(numElts < 100);
9      elts[numElts++] = v;
10   }
11 }
12 void IntSet::remove(int v) {
13   int vic = indOf(v);
14   if (vic != MAXELTS) {
15     elts[vic]=elts[numElts-1];
16     numElts--;
17   }
18 }
```

Analyze the methods:

- L.9 checks inv.3 for uniqueness.
- L.7 checks inv.2 for capacity. If violation of inv cannot be recovered from, depending on how serious situation is you can `throw` or fail-fast.
- L.9 uses inv.1. We insert at the end, temporarily breaks invariance.
- L.9 operator ++ restores inv.
- Inv.3 ensures `remove` works.
- L.15 restores inv.1. L.16 restores inv.2. A significant amount of code is dedicated to preserving invs.

Remarks on invariants

We would like to note the following things:

The centric problem in choosing an implementation for an ADT, is choosing the invariants.

There are multiple ways to write an implementation an `IntSet` of course. The major difference of them, is they need to keep different invariants, and thus have different performance.
For example, you have already seen `IntSet` implemented using a sorted array. The one extra invariant you need to keep is that elements are sorted. Based on that extra invariant:

- Queries are much quicker.
- Deletion and insertion are slower (by a constant factor).

In VE281 you will see (many) other ways of implementing this IntSet, for example using hash tables, or balance trees. It is always helpful to understand what the invariant for an implementation.

Invariants and bad practices

Thinking about invariants immediately gives us hints on what are seemingly good, some even sounds terrific, but are actually pretty bad ideas. We now take a look at some of them.

Alice I'd like to add a public method to `IntSet`, let's say `find()`, which takes an element, found it in the set, and returns an reference to that element.

Bob Why do you want to do that?

Alice Well my code needs to modify elements in the array, a lot. This would significantly accelerate the process.

Bob ...

Alice Oh, I get it, it's a pretty bad idea. OK I have another idea, I want to add a method, `data()`, which returns the pointer to array, since my code needs to iterate through the set.

Bob ...

Alice What if I return pointer to const?

Bob ...

Alice Well, I give up. It's not as easy as I thought.

Invariants and bad practices

Well Alice does have her point. Her requirements are very real and the solutions does solutions could work, but remember the words from *Donald Knuth*:

Premature optimization is the root of all evil.

Never trade correctness for performance in designing phase.

- 1st and 2nd change Alice proposed have the problem that the client could change the content of the internal array, without notifying the instance. There is by no means the instance can guarantee the new value won't violate uniqueness invariance.
- Now Alice proposes the third one. The `const` key word could prevent the element from being modified. But it exposes the implementation to outside. Now the fact that elements are stored in a linear array becomes part of the abstraction, you can no longer change that. The problem is, iterating through a set seems to be an important feature, we DO need that. Well, the correct solution is using iterators, you will know them later in this course.

Invariants and bad practices

Well Alice does have her point. Her requirements are very real and the solutions does solutions could work, but remember the words from *Donald Knuth*:

Premature optimization is the root of all evil.

Never trade correctness for performance in designing phase.

- 1st and 2nd change Alice proposed have the problem that the client could change the content of the internal array, without notifying the instance. There is by no means the instance can guarantee the new value won't violate uniqueness invariance.
- Now Alice proposes the third one. The `const` key word could prevent the element from being modified. But it exposes the implementation to outside. Now the fact that elements are stored in a linear array becomes part of the abstraction, you can no longer change that. The problem is, iterating through a set seems to be an important feature, we DO need that. Well, the correct solution is using iterators, you will know them later in this course.

Invariants and the constructor

Invariants are there as long as the object is there. The earliest moment the object is created, is when it is defined. Classes, are just structures, in terms of memory. Structures contains undefined values when created, so naturally does the classes.

Think about `IntSet`, when the object is created, chances are `numElts` contains non-zero value, while your set should be empty.

We need a mechanism, a function, that is invoked whenever the an instance is created. This function is called, and called always, to initialize the object, or more strictly, to setup the invariance.

Such special functions are constructors, or ctors for short. From this point on it's important to think the initialization, not as assigning special values to member variables, but as invoking the constructor to setup invariants (together with the initial state, e.g. initialize the `IntSet` with one default element in it for whatever reason).

Constructors: Syntax

The construct for `IntSet`:

```
class IntSet {
  // ....
public:
  // ....
  IntSet() : numElts(0) {}
  // Intialize an empty set
  IntSet(int v);
  // Init. a set with 1 elt
  // provided in the arg
};
IntSet::IntSet(int v)
    : numElts(1) {
    elts[0] = v;
}
```

- Ctors have same name as class.
- Ctors don't return anything.
- Usually `public`, but could be `private` for special need.
- A class could have more than one ctor, depending on what the initial state should be. We say the constructor is overloaded.
- Ctors could take initialization list.
- Ctor is the first function being called when the object is created. Unfortunately "created" is not as simple as it sounds.
- A ctor that does not take argument is called a `default ctor`. Every class must always have a ctor. If you don't specify any ctor, a ctor is synthesized for you (under some conditions).

* We left out copy/move constructors on purpose. You will learn them later.

Constructors: Explanations

We need to break down the words from previous slide. We first look at the first four lines. The first three seems direct. For the fourth one:

```
IntSet set1; // set1 is init to an empty set;
IntSet set2(10); // set2 contains single elem 10
```

```
int foo(const IntSet& s1, const IntSet& s2);
foo(IntSet(), IntSet(10)); // Anymous construction
```

Further more, latest C++ standard guarantees the following:

```
IntSet setx = IntSet();     // Same as set1
IntSet sety = IntSet(10);   // Same as set2
IntSet setz = 10;           // Same as set2
```

This is non-trivial, for reasons you will see one or two weeks later. This is called *guaranteed copy elision*. Now some food for thought: Why `set1` is not initialized as `IntSet set1();` for consistency?

Constructors: Explanations

We now question: Is constructor really the first function being called when created? Well that depends on how you understand "created"! One should argue the ctor IS part of object creation.

In fact, there are 4 steps (roughly) in creating an class instance.

1. Allocation of space, we (in general) don't have control over.
2. Construction of the base class object.
3. Construction of the members of current class.
4. Calling the constructor.

These 4 steps goes recursively. In fact the constructor is the last function being called. This is natural.
When the constructor is called, we should be able to use all member variables and base class. So their invariants must have been setup in advance! This is the only way things makes sense.

Initializer list: Setup

The *initializer list*, the things after colon controls the second and third step.

```
ClsName::ClsName() : base(..), m1(..), m2(..) {
    // Code for the constructor goes here
}
```

To better illustrate our ideas, we make some changes to our original `IntSet`, specifically 2 changes:

- We add a line of output to the ctors to indicate with constructor is being called.
- We make 2 more "version" of our original `IntSet`, one `OddIntSet` and `EvenIntSet`. They only allow odd and even numbers, respectively.

We now propose a slightly faster `IntSet`. This `IntSet` is composed of an `OddIntSet` and `EvenIntSet`. It dispatches the operation on integers to those two sub-`IntSet`.

Initializer list: Setup

The *initializer list*, the things after colon controls the second and third step.

```
ClsName::ClsName() : base(..), m1(..), m2(..) {
    // Code for the constructor goes here
}
```

To better illustrate our ideas, we make some changes to our original `IntSet`, specifically 2 changes:

- We add a line of output to the ctors to indicate with constructor is being called.
- We make 2 more "version" of our original `IntSet`, one `OddIntSet` and `EvenIntSet`. They only allow odd and even numbers, respectively.

We now propose a slightly faster `IntSet`. This `IntSet` is composed of an `OddIntSet` and `EvenIntSet`. It dispatches the operation on integers to those two sub-`IntSet`.

Initializer list: Setup II

Decl. of OddIntSet

```
#ifndef _ODD_INT_SET_H_
#define _ODD_INT_SET_H_
const int MAXELTS = 100;
class OddIntSet {
  int elts[MAXELTS], numElts;
  int indOf(int v) const;
public:
  OddIntSet();
  OddIntSet(int v);
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int  size() const;
};
#endif
```

Partial implementation:

```
#include "oddintset.h"
#include <iostream>
using namespace std;

OddIntSet::OddIntSet()
    : numElts(0) {
cout << "OddIntSet dft ctor\n";
}

OddIntSet::OddIntSet(int v)
    : numElts(0) {
this->insert(v)
cout << "OddIntSet elt ctor\n";
}
```

We omit the very similar version for EvenIntSet.

Initializer list: Setup III

Declaration of `OddIntSet`

```
#ifndef _FAST_INT_SET_H_
#define _FAST_INT_SET_H_
#include "oddintset.h"
#include "evenintset.h"
class FastIntSet {
  OddIntSet oddSet;
  EvenIntSet evenSet;
public:
  FastIntSet();
  void insert(int v);
  void remove(int v);
  bool query(int v) const;
  int  size() const;
};
#endif
```

Partial implementation:

```
#include "fastintset.h"
#include <iostream>
using namespace std;

FastIntSet::FastIntSet()
    : oddSet(), evenSet(10) {
  cout << "fIntSet ctor\n";
}

FastIntSet::insert(int v) {
    if (isOdd(v))
        oddSet.insert(v);
    else
        evenSet.insert(v);
}
```

Initializer list: Example

We create a main function and instantiate an instance of `FastIntSet`. We observe the following ouput:

```
OddIntSet dft ctor
EvenIntSet elt ctor
fIntSet ctor
```

This implies an initialization order of `oddSet-evenSet-ctor`.
This follows from our expectation.

A (not so important) remark we would like to make is

- The order of initialization of the members is guaranteed by standard. It's NOT a UB!
- Contrary to intuition *it is not specified by initializer list*, but by the order they appear in the declaration.
- Base objects are guaranteed to be initialized before members.
- We recommend you to specify the init-list in the same order as the declaration, for clearance.

Initializer list: Default initialization

We now make a tiny change to the original setup. We change the ctor for `FastIntSet` to

```
FastIntSet::FastIntSet()
    : oddSet() {cout << "fIntSet ctor\n";}
```

Oops, we forgot to specify initializer for `evenSet`.

```
OddIntSet dft ctor
EvenIntSet dft ctor
fIntSet ctor
```

`evenSet` is still initialized, no surprise, otherwise the invariant for `oddSet` would be broken. Members are default constructed, if they don't appear in the initializer list. We could even simply write

```
FastIntSet::FastIntSet() {cout << "fIntSet ctor\n";}
```

Which will produce the same result. Now what about members like `int`? Well, you can think of them as a class, whose default constructor does nothing at all!

Initializer list: Default initialization

We can even leave out the constructor for `FastIntSet` entirely. In which case we would have the following output.

```
OddIntSet dft ctor
EvenIntSet dft ctor
```

Essentially the compiler would synthesize (create) and constructor for you, whose initializer list initializes everything using their default constructor.

You might wonder, wait a minute, I have never write initializer list before, I don't even write a constructor before this lesson! That's how you "get away" with writing constructors before!

Modern design principal is against leaving things unspoken. If you believe you only need a default constructor, you could do:

```
IntSet::InSet() = default; // In class decl.
```

Next up we will look at something that is not taught in class, but problems you will meet in your own struggle with C++.

Initializer list: Default initialization

We first change the ctor of `FastIntSet` to

```
FastIntSet::FastIntSet()
    : oddSet() {cout << "fIntSet ctor\n";}
```

Then we **remove the default ctor** for `EvenIntSet`.
We compile the program. The question is what SHOULD happen?

Initializer list: Default initialization

We first change the ctor of `FastIntSet` to

```
FastIntSet::FastIntSet()
    : oddSet() {cout << "fIntSet ctor\n";}
```

Then we **remove the default ctor** for `EvenIntSet`.
We compile the program. The question is what SHOULD happen?

From a design point of view, if a class does not provide a default constructor, indicates the class is not (more likely should not be) default constructible, there is no way to setup a invariance without supplying an argument. Think about it, what should be the default skin color for an instance of a human class? None.
In this case, any reasonable design would require an compile error is thrown. This is exactly the case. They compiler will look for `evenIntSet::evenIntSet()` and reports a not-found error.
Situation is similar if you leave out the entire ctor of `FastIntSet`.

Why initializing in initialization list is better?

You are asked this question in the lecture slides but the question is left unanswered in the lecture slides.

Frankly the questions really should be the other way around: on what grounds do initialization in the code is better? I can think of the following reasons you should prefer (in some cases you have to use) initialization list:

- Performance. This will be discussed later on. In some cases the performance could be significant.
- Exception safety. We will discuss this when we you learn the rule of big three (five, actually).
- A member that don't have a default constructor must be initialized in the initialization list.
- `const` members and references can only be initialized in the initialization list.
- Semantic reasons.

Comments on the performance argument

A final comment on the performance argument. Suppose we write

```
FastIntSet::FastIntSet() {
  cout << "fIntSet ctor\n";
  oddSet = OddIntSet(); evenSet = eventSet(10);
}
```

You will observer 5 lines of output:

```
OddIntSet dft ctor // EvenIntSet dft ctor //
fIntSet ctor // OddIntSet dft ctor // EvenIntSet elt ctor //
```

Things that actually happened (with significant cost) are :

- Your member variables are first default constructed.
- Calls your constructor.
- You create 2 temporal objects, by default construction.
- You invoke an assignment operator and uses your temporal object to overwrite the member variables.

RC Week 9

new Operator, Deep Copying, RAII and Resource Management

Overview

This chapter we dealt with a bunch of all related cocepts, for a better under standing we would first clarify the following:

- **Resource Management** is the problem we are tackling. Resources includes "Memory", files, IO Devices (printers, Internet connections) and Locks, Mutexes, Semaphores when you learned VE482/EECS482. Often there is a limited amount of them and programs share them.
- **RAII**, stands for **R**esource **A**cquisition **I**s **I**nstantiation. It's the idea behind all these messy rules. The key problem in RAII is to identify **ownership**.
- **Deep Copying** is the technique is used to implement RAII.
- `new` **and** `delete` simply represents the one most common resource used in programs.

Need for the `new` and `delete`

We once again examine the necessity for the two operators:

- Programs may require an statically unknown number objects. For examples, for a database (think of fancy spreed sheet!) application number of entries in a table is unknown when the program is designed.
- The precise time for the creation (allocation) and release of these objects are unknown in compile-time.

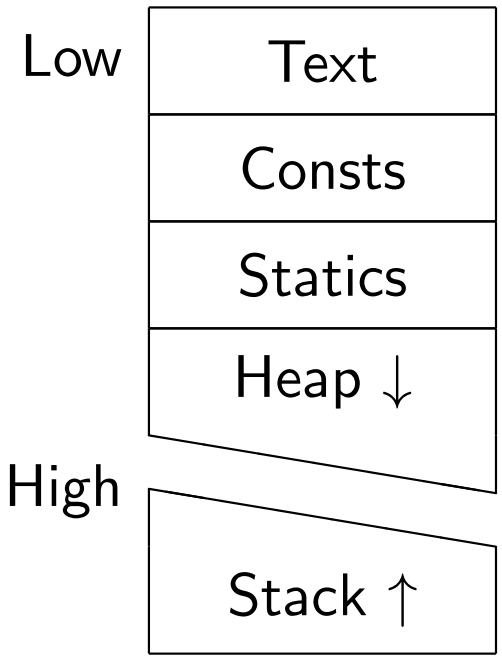
This calls for a mechanism that

- Allocates **the object** on-demand.
- Destroys **the object** on-demand.

It just happens that these objects requires memory. One should always understand that `new` is different than `malloc` in that it does not just allocate memory. It allocates objects, it just happens that these objects requires memory!.

The Heap

This graph is up-side-down, This is the same graph in page 103.



- Since we cannot decide when to allocate and when to destroy the object we surely CANNOT allocate the objects from the stack. The only way is to allocate them from a separate region, which is the Heap.
- The name is strange in that there is no commonly agreed upon origin. According to TAOCP by Knuth several authors starts to using it and everybody follows through.
- Heap is usually much bigger than stack. You should allocate large amount of memory only on heap. But that's not the decisive reason.

new and new[]

new and new[] does the following:

- Allocates space in heap (for one or a number of objects).
- Constructs object in-place (including, but not limited to ctor).
- Returns the "first" address.

The syntax for new operator are very simple

```
Type* obj0 = new Type;    // Default construction
Type* obj1 = new Type(); // Default construction
Type* obj2 = new Type(arg1, arg2);
Type* objA0 = new Type[size]; // Default cons each elt
Type* objA1 = new Type[size](); // Same as obj A1
```

For array allocation has no easy to use "construct with argument" option. In fact array allocation is seldom useful. You should always consider std::vector instead when you learned STL.

delete and delete[]

Since new-expressions allocates memory from the heap, they essentially requested (and occupies) resources from the system. For long running programs resources must always be returned (or released) when the program is finished with them, otherwise the program will end up draining all system resources, in our case running out of memory.
delete and delete[] releases the objects allocated from new and delete[] respectively. They does the following:

- Destroy the object (each object in the array) being released (by calling the *destructor* of the object).
- Returns the memory to the system.

We must emphasize that delete an object more than once, or delete an array allocated using new[] by delete instead of delete[] cause undefined behavior!

Memory Leaks

Now the problem of memory leak seems obvious. If an object is allocated, but not released after the program is done with it, the system would assume the resource is still being used (since it won't examine the program), but the program will never use it. Thus resource is "leaked", i.e. no longer available for using. In our case the leaked resource is memory.

```
void foo() {int* p = new int(0); /* Code */}
```

The function `foo` causes memory leak each time it is being called. When the function returned the handle of the resource, i.e. the address (stored in `p`), is lost, thus the program is unable to release the memory. Thus we have a memory leak.
Technically memory leak represents all situations where resources are not released once the program is done with it. The following program also "leak" resources in the technical sense.

```
IntPointerSet s;
void foo() {int* p = new int(0); s.add(p); /* Code */}
```

Fixing Memory Leaks

Fixing memory leaks is by no means as simple as you might thought! You might thought the following code fixes the `foo` once and for good.

```
void foo() {int* p = new int(0); /* Code */ delete p;}
```

But what if an exception is thrown in `/* Code */`? The function will return immediately at the point of exception, skipping the final `delete` statement. You could

```
try{/* Code */}catch(...){delete p; throw;}
```

But I trust you know the draw back!. Another problem is what if `foo` has multiple return point? Will you be able to remember to release everything whenever you `return`?. What if above two situation are mixed?
All these problem suggest we need a better way of handling the problem! We need a strategy other than patches!

Digression: Diagnosing memory related problem

`valgrind` is not a tool that only looks for memory leaks. It actually looks for for all sorts of memory related problems, including:

- Memory Leaks.
- Invalid accesses, array out-of-range, use of freed memory, etc.
- Double free problems.

These problem are most likely UBs so bugs related to them are very likely to be elusive and random. Always use `valgrind` to check your program. It significantly improves your success rate on OJ.
Valgrind can also act as a *profiler*. A profiler measures how much time is spent on each portion of your code. It let you to know which portion is taking up the most time. Profiling is always the first step towards performance optimization.
Valgrind does all these by tap into (hijacks) EVERY function call of your program, which comes with a truely significant performance cost.
For more information, RTFM, thank you.

Lifetime of objects

We now take a little digression. The *lifetime of an object*, sometimes "life cycle", is the period starting from creation of object, to the point the object is destroyed. We typically has (roughly) 3 of them

- Global objects. They are created when the program starts and destroyed when the program is terminated. Including global variables and `static` variables.
- Automatic objects. Basically local variables.
- Dynamic objects. Objects created/deleted on-demand through `new` and `delete`.

From the lifetime view resource leaks are simply dynamic objects whose lifetime should have been terminated when they are no longer useful. It should be clear that the lifetime of the objects is tightly related to the *scope* of the variable. This is natural. Since global objects can be access by any portion of the program they have to exist throughout the program. Local objects are only visible in the enclosing brackets thus they are destroyed when they go out of scope.

Binding resource to the scope

The solution to our previous problem is to bind the resource to a scope where it is needed. The detailed way of doing so is:

```
// intset.h
const int MAXELTS = 100;
class IntSet {
    int *elts, sizeElts;
    int numElts;
public:
    IntSet(int size = MAXELTS)
        : elts(new int[size]),
          sizeElts(size),
          numElts(0) {}

    ~IntSet() { delete[] elts;}
    // Other methods unchanged
};
```

```
// driver.cpp
#include "ex.h"
class T {
    IntSet s1, s2;
// ommit other
}
// Nothing leaks
void foo() {
    IntSet s;
    while (...) {
        T t; if (...) throw -1;
    }
    if (...) return;
    else ...;
}
```

RAII

Stands for *Resource Acquisition Is Instantiation*. RAII is perhaps the one most famous rule specific to C++, unfortunately with an extremely terrible name! Recently people are start referring to the rule by *Scope-based Resource Management*. It's a little bit clearer, but by no means summarizes it's full power. RAII requires the following:

Holding an resource is a class invariant!

With a little (?) bit explanation:

- Resource is allocated in ctors and ctors only.
- Resource is released in destrctors (dtors).
- Object "owns" the resources. The resource is managed by the object and the object only.
- The resources would share it's life cycle with object. As long as there is no object leak there is no resource leak. Fortunately we know automatic objects are destroyed by the compiler when they go out of scope, impossible to have object leak. Valgrind is only

Destructors

Often denoted as `dtors` for short. Destructors should:

- Be named as `~ClassName`.
- Takes no argument and returns nothing (Not even `void`).
- If one expect the class to be inherited the dtor should be declared as virtual.
- Release resource allocated only in this class (don't release base class resources!).

The process of destroying an object is as follows:

- It calls the dtor of the class.
- Calls the dtors for each member of current class.
- Calls dtor of the base class.
- Does above recursively until no more dtors to invoke. Finally it releases the memory.

Dtor examples: Memory Leak

The following code causes memory leak problem.

```
//classes.h
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    ~Base() {delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() :
     Base(), q(new int(20)) {}
    ~Derived() {delete q;}
};
```

```
//driver.cpp
#include "classes.h"

void foo() {
    Base* ptrA = new Derived;
    delete ptrA; // Memroy Leak!
}

void foo() {
    Derived* ptrB = new Derived;
    delete ptrB; // Safe
}
```

Dtor examples: Double Free

The following code causes double free problem.

```
//ex2.h
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    virtual ~Base(){delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() :
     Base(), q(new int(20)) {}
    ~Derived()
    { delete q; delete p;}
};
```

```
//ex2.cpp
#include "classes.h"

void foo() {
    Base* ptrA = new Derived;
    delete ptrA; // Double Free!
}

void foo() {
    Derived* ptrB = new Derived;
    delete ptrB; // Double Free!
}
```

Dtor examples: Free only what you "own"

This is our correct solution.

```
//ex3.h
class Base {
protected:
    int *p;
public:
    Base() : p(new int(10)) {}
    virtual ~Base(){delete p;}
};

class Derived : public Base {
    int *q;
public:
    Derived() :
     Base(), q(new int(20)) {}
    ~Derived() {delete q;}
};
```

```
//ex3.cpp
#include "classes.h"

void foo() {
    Base* ptrA = new Derived;
    delete ptrA; // Safe!
}

void foo() {
    Derived* ptrB = new Derived;
    delete ptrB; // Safe!
}
```

Linked List example

To support our future discussion with a concrete example we must take the singly linked list example from the instructor. We would like to make a comment. Objects like the linked list are sometimes called "containers". The major purpose of containers is to container other objects, store and more importantly arrange them in a manner suitable for specific access. For example:

- Fast random access with few or no deletion at all? go for an array.
- Only access front? Frequent insertion at front? linked list.

However there are in general 2 types of containers:

- Container of value. The container only store the "value" of inserted object. The container does not give you back the exact same object that you gives it. It only gives back a object that has the same "value" you provided. It does NOT own the object.
- Container of pointer. Think of a value container whose value are pointers. It gives you back the actual object. When you insert an object into a container of pointer, you are transfering ownership!

Linked List Interface

```
class listIsEmpty {}; // An exception class
struct node{ node *next; int value; };
class IntList {
    node *first;
    void removeAll();
    void copyList(node *list);
public:
    bool isEmpty() const;
    void insert(int v);   // inserts v into the front of the lis
    int remove(); // Pops the first element
    void print() const; // print the int list
    IntList();                          // default constructor
    IntList(const IntList& l);  // copy constructor
    ~IntList();                         // destructor
    IntList &operator=(const IntList &l); // assignment operator
};
```

Linked List Implementation: ctor and dtor

```
bool IntList::isEmpty() const {
    return first == nullptr;
}

IntList::IntList(): first(nullptr) {}

IntList::~IntList() { removeAll(); }

void IntList::removeAll() {
     while(!isEmpty()) remove();
}
```

Resource management: An analysis

Keep in mind that an object "owns" the resources. Keep in mind as well that holding a resource is a class invariant. Whenever class invariant is in play we must question whether each operation breaks / preserves the invariant.

- Ctors and Dtors are checked already :=).
- `insert()` and `remove()` might break invariant.
- `print()` and `isEmpty()` are `const`.

But there are two cases that are more tricky.

- When you pass the object by value, the content of the object will be copied, byte by byte. Now both the "original" object and the "copy" both owns the same actual list.
- When you performs assignment a byte-wise copy will happen. In addition to the previous problem, the old object must give up it's ownership!

insert and remove

A kind note: draw the graph when confused. Remember boundary condition that `first` could be empty!

```
int IntList::remove() {
    if (isEmpty()) throw listIsEmpty();
    int result = first->value; // None empty, always possible
    node *victim = first;      // Whay save to victim?
    first = first->next;
    delete victim;             // Giving up ownership
    return result;             // Value container
}

IntList::IntList(const IntList &l)
    : first (0) {
        copyList(l.first);
    }
```

Resource management: Copying

Keep in mind that an object "owns" the resources. A value container owns the location it used to store the values. A container to pointer owns both the location it used to store the pointers, but also the objects stored by the pointer.

These invariants are part of that objects, and must be preserved for both the copied object and the original object when passed into a function.

When a variable is passed into a function by value, the copy constructor of a function will be called. The copy constructor has the following form, argument type is critical!

```
class Type {
// Omit other methods
public:
    Type(const Type& type); // Copy constructor
}
```

Copy constructor

Just like the default constructor:

- A copy constructor is "synthesized" if not specified.
- For types like `int` copy constructors performs byte-wise copy.
- A synthesized copy constructor by default calls it's element's copy constructor.

All three rules combined provides the same behavior as in C when you pass a struct (class) by value. But the real process happened is listed above. But a copy constructor is after all a constructor and that gives us problems:

- If you only write a custom copy constructor, since there already exists one ctor, the default ctor will not be synthesized.
- If you choose to write a custom copy ctor, you must remember to call the copy-ctor on each of your element in initialize list. If you don't, your members will be default initialized.

This gets much more complicated combined with inheritance.

Implementing the copy ctor

This is very easy to understand recursive algorithm. A comment is that a copy ctor is also a ctor, which means all rules applicable to a regular ctor applies to copy ctor as well, for example the initialization list.

```
IntList::IntList(const IntList &l)
    : first (0) {
        copyList(l.first);
    }

void IntList::copyList(node *list) {
    if (!list) return; // Base case
    copyList(list->next);
    insert(list->value);
}
```

Operator Overloading

The first rule of thumb:

Operators are just functions.

And functions can be re-written. You can rewrite the effect of existing operator on your custom type. This is a very powerful tool of abstraction. For example you could overload $*$ on matracies, so that they looks as if they are regular values. It might surprise you how many operators can be overloaded:

- Arithematic logic operators: $<<$, $+$, $-$, $*$, $==$, $||$, $\&$...
- Unary logic: !,
- Special operators: $[]$ (array), $->$, $*$ (pointers), $()$ (function).
- Memory: `new` and `delete`
- And others ...

A few remark:

- Overloaded operators preserves their old precedence.
- Overloaded operators preserves their old associativity.

Overloaded operator =

It's better that to keep the original "behavior" of an overloaded operator. For example you might want to overload + with a function that preserves interchangeability. You might want to avoid overload operator = with a function that returns a `bool`. For the operator =

```
class Type {
public:
        Type& operator= (const Type& rhs);
};
```

- The return type should be a reference to the left hand side.
- The keyword `operator` indicates opearator overloading. You can think the name of the function is `operator=`.
- The argument type must be a const reference to the right hand side.

Again if you don't overload the assignment operator, one will be synthesized for you, which follows similar rules as the cp-ctor.

Implementing the overloaded assignment operator

```
IntList &IntList::operator= (const IntList &l) {
    if (this != &l) {
        removeAll();
        copyList(l.first);
    }
    return *this;
}
```

In addition to the copying the overloaded assignment operator needs to first release the resources of the left hand side. Because the left hand side is letting go the ownership of old resources, and taking over new resources from the compied from list.
There is a caveat. In the copy constructor the destination is guaranteed to be different from the source (why?). However one might assign a value to itself. What problem would there be if the branch `if (this != & l)` is not there?

Move Semantic: The problem

Consider the following code, suppose we have a function combine

```
IntList combine(const IntList& l1, const IntList& l2);}
```

In a function `foo`

```
void foo() {
        IntList l1; // Populate l1 with much data
        IntLIst l2; // populate l2 again with data
        IntList l3 = combine(l1, l2);
}
```

We examine what will happen on the third line.

- On return of `combine`, a temporary object will be constructed.
- The temporary object will be used to copy construct `l3`.
- The temporary object will be deleted.

You should realize that the second step comes with a significant performance cost.

Move Semantic: An analysis

We now take a step back and think: We don't need to **copy** the content of temporary object. What we really need is to **move** the resources from the temporary object to the target object. We need a mechanism to transfer the ownership of resources from the temporary object to the target. The language features supporting this is called move semantics.

We know that the temporal object is a rvalue, but in order to do what we expected we need to modify the temporal object, i.e. pass the temporal object by reference of some sort:

```
class Type {
public:
    Type(Type&& rhs); // A move constructor
};
```

Here `Type&& rhs` declares a *rvalue-reference*.

Move Semantic: Implementation

```
IntList::IntList(IntList&& rhs)
    : first(rhs.first) {
    rhs.first = nullptr;
}
```

Remember move constructor is also a ctor. In this very simple code it exchanges resources owned by the original object (which is empty) and the temporal object. Note argument is not consted.

Another similar situation is the case of assignment operator:

```
Type& operator=(Type&& rhs); // Move assignment
```

And the implementation

```
IntList& IntList::operator=(IntList&& rhs) {
        swap(this.first, rhs.first);
}
```

It is equally simple. The resources of the original object will be freed during the deconstruction of the temporal object.

The rule of big X

Where $X = 3$ traditionally and $X = 5$ after c++11.

Whenever an object owns resources, any resources, not just memory, it should implement 5 methods:
A ctor and a dtor, A copy ctor, a move ctor, a copy assignment operator, and a move assignment operator.

These are 5 typical situations where resource management and ownership is critical. You should never leave them unsaid. If you want to use the version synthesized by the compiler, use `=default`:

```
Type(const Type& type) = default;
Type& operator=(Type&& type) = default;
```

Traditionally constructor/destructor/copy assignment operator forms a rule of 3, you should know this if being questioned in the exam. Move semantics is a feature available after C++11.

Singleton Pattern

Very often some object is "global". It is used by the entire program (for example `World` class). Probably this object should not be copied, or more likely cannnot be copied. Probably this object can only be validly instantiated once.
Using a global variable is one possible solution, but hardly a good one. For instance if you have multiple global instances the order of their instantiation is not guaranteed.
A clean slate solution is called *the singleton pattern*:

- Clients use a `static` method to access the object.
- Object is created on first access, than return every time.
- Implement `private` constructor.
- Use deleted copy ctor and deleted assignment operator to explicitly signify the client that this object is not copy-able.

Singleton pattern: Implementation

```
// s.h
class S {
        static S* instance;
        S() {}                          // Constructor
    public:
        static S& getInstance() {
            if (instance == nullptr) instance = new S;
            return *s;
        }
        S(S const&)                     = delete;
        S&   operator=(S const&)  = delete;
};
// s.cpp
S* S::instance = nullptr; // initialize
// foo.cpp
void foo() { S& s = S::getInstance(); /* Use s as you wish */}
```

Problems with exception

Quotation from Google C++ Style Guide:

> *We do not use C++ exceptions. ... Our advice against using exceptions is not predicated on philosophical or moral grounds, but practical ones. ... Things would probably be different if we had to do it all over again from scratch.*

The Criticism

Again we quote from the same document:

> *When you add a throw statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result.*

`https://google.github.io/styleguide/cppguide.html#Exceptions`

Exception Safety

There are in general 3 levels of exception safety:

No-throw Guarantee The function will not throw an exception or allow one to propagate.

Strong Guarantee If a function terminates because of an exception, it will not leak memory and no data will not be modified, as if it has never been called from the beginning.

Basic Guarantee If an exception is thrown, no resource (memory) is leaked ... though the data might have been modified.

Understand these guarantees as part of the specification for your function. Exceptions allows the function to interact with the outside world (by throwing exceptions) and it requires the outside world to change accordingly (catch exceptions).

From MSDN 0https://msdn.microsoft.com/en-us/library/hh279653.aspx

No-throw guarantee

No-throw guarantee is very strong guarantee! You might think this is easy. You might thought as long as your function does not throw anything than you are safe. However consider the following example:

```
int g(int); int foo(int i) { return g(i); }
```

In this example, `foo` does not give no-throw guarantee. It g could throw an exception so that `foo` fowards an exception. But can we make `foo` no-throw?

The only way of doing this is to allow `foo` to catch everything. But since `foo` does not have the faintest idea what kinds of exception g might throw (since exception are not part of abstraction specification), it cannot catch them without causing problems.

In this example it is simply impossible to achieve no-throw guarantee.

Strong Exception Safety

Now we take a step back. What about strong exception safety. At first look it seems strong exception safety is easier to achieve since a function need only to look over it's own stuff. But this is very difficult as well. Consider the following code:

```
int divide(IntList& stack) {
    int num = stack.pop();
    int denum  = stack.pop();
    if (denum == 0) throw ExceptionDivZero();
    return num / denum;
}
```

At first look you might think this is perfect. But this code is not strong exception safe. This function has side-effect (still remember what are side effects?), and side effects must be reversed before exiting from `divide`. Unfortunately not all side-effects are reversible (within reasonable performance overhead).

Basic Exception Safety

Now we seek the least. Can we achieve at least basic safety. Well, the good news is we can, but by no means easy!
Note RAII takes most of the trouble away. If you follow RAII and only explicitly use local objects (so they are destroyed automatically by the compiler) you are already on the safe track.
When a exception happens, the compiler performs *stack unrolling* (see page 116 of the slides).
But there is one corner case. We start by asking what happens if a constructor has to report an error.

- If an construction fails, which indicates the invariance of an object cannot be satisfied, so that an object cannot be valid.
- Constructors do not have return value.

It seems we are cornered, but we do have choices.

Error handling in constructors

One way of handling this is to create a "dummy" object, a tombstone. This is used in `fstream` in standard library. If you construct a file string with a invalid file, you would have a `fstream` that is in a failed state. It is not bind to any resource (it owns nothing). You can use a method (`fstream::is_good()` in our case) to determine it's state.
This have the same problem of error code! (What are they?)

Another approach is to throw an exception in a constructor (yes you can do that!). However, since we are still in the constructor, this indicates the object itself has not been valid (i.e. it does not really exist). When a exception is thrown:

- Terminate the function and unroll stack as usual.
- The destructor will NOT be invoked.
- It's members are already valid object, so they will be destructed.

Exceptions in constructors

The rules can easily gets extremely complicated. First of all, since destructor of an object might be called in the process of throwing an exception (the exception is raised, but is not brought to process), it will be extremely bad if another exception is thrown. So that a destructor may not throw any exception.

Secondly we note a few complicated situations.

- If a member of class throws an exception in the process of initialization, the members that have been initialized will be destroyed.
- If one of the objects throw an exception in initializing an array of objects obtained by `new[]`, the objects that already constructed will be destroyed.

And there are more of them. They are so tricky that the only way to safely deal with is to follow RAII strictly.

Smart Pointers

The problem of ownership often comes with pointers. We use pointers for two very different reasons:

- To signify owner ship.
- To store a reference of an object.

```
Type* getNewObject(int param) {
    Type* obj = new Type; /* Calculation */ return obj;
}
```

Above code essentially transfer the ownership of the new object to it's caller. The caller must be aware that it is accepting an ownership, thus it should be in charge of releasing it.

Further more it could happen an object is shared among multiple modules. The ownership is not clear, and we might not know which module finishes using it last.

Smart Pointers

The standard library features 3 smart pointers:

```
std::unique_ptr<Type> ptr;   // Unique ownership
std::shared_ptr<Type> s_ptr; // Shared ownership
std::weak_ptr<Type> s_ptr;   // No Ownership
```

Unique pointer features a complete ownership. This pointer cannont be duplicated by copying. It can only be transfered through `move`.
Shared pointer indicates shared ownership. Multiple owner "shares" the object. Reference counting it used to indicate how many instances is using the object. When reference counting goes to zero, the object is released.
Weak pointer indicates no ownership at all. Smart pointers supports syntax just like pointers. For more information RTFM.

RC Week 10

Sub-types, Code Reuse and Inheritance

Principals of Object Oriented Programming

There are 5 widely accepted principles throughout the object oriented design:

1. Single responsibility principle
2. Open/close principle
3. **Liskov substitution principle**
4. Interface segregation principle
5. Dependency inversion principle

- The idea of "abstract data type" by first proposed by Barbara Liskov and Stephan Zilles (1974) in "Programming with abstract data types".
- Later on in Liskov's 1988 key note "Data Abstraction and Hierarchy", the SOLID principles of object oriented programming was first proposed.

Principals of Object Oriented Programming

Barbara Liskov (1939 to present)

- MIT computer scientist
- Ford Professor of Engineering
- One of the American's first women PH.D. in Computer Science

- 2004 John von Neumann Medal winner for "fundamental contributions to programming languages, programming methodology, and distributed systems"
- 2008 Turing Award winner, for her work in the design of programming languages and software methodology that led to the development of object-oriented programming

Sub types and Liskov Substitution Rule I

The substitution rule is formally described as follows:

> If S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of T (correctness, task performed, etc.)

This is by all means very abstract. We provide a more "concrete" explanation.

Subtype relation is an "IS-A" relationship.

For examples, a `Swan` is a `Bird`, thus a `class Swan` is a subtype of `class Bird`. A bird can fly, can quake and can lay eggs. A swan can also do these. It might do these better, but as far as we are concerned, we don't care. `Bird` is the super-type of the `Swan`.

Pre-conditions & Post-conditions

The term "sub-type" is misguiding because the prefix "sub" suggests the sub-type is "inferior" to the super-type in some sense. But this is actually other way around.

- The assertions in the `REQUIRES` clause and argument types in abstraction specs combined is called a "Precondition".
- The assertions in the `EFFECTS` and `MODIFIES` clause specifies the post conditions.

Sub-types typically perform a combination of belows:

- Supports extra operation. Naturally preserves follows LSR.
- Weakens the pre-conditions. Expands range of input.
- Strengthens the post-condition. Enhances the effect.

In reality sub-types are beefed-up versions of the super-type. They perform more specific jobs or do the same job better.

Inheritance

We now start to look at a specific language feature in C++, namely *inheritance*. The syntax takes the form of the following:

```
class Derived : /* access */ Base1, ... {
    /* Contents of class Derived */
};
```

When a class (usually called *derived*, *child* class or *subclass*) inherits from another class (*base*, *parent* class, or *superclass*), the derived class is automatically populated with *everything* from the base class. *Everything* includes member variables, functions, types, and even static members. The only thing that does not come along is *friend*ship-ness, which is irony.

Note that, it is one thing that the derived class "has" everything from the base class, it is totally another thing whether the base class can access it. Whether the base class can access the member is dependent on the choice of `access`.

Inheritance access specifiers: `public`

There are a three choices of `access`, namely `private`, `public` and `protected`. The most commonly used one is `public`.

- Private member of the base class stays private *to the base class*. Even the derived class CANNOT touch them!
- Public member of the base class stays public. This means they are exposed as part of the interface of the derived class.

```
class Base {
    private: int priv; void privMethod();
    public:  int pub;  void pubMethod(); };
class Derived : public Base {
    void bothError() { priv = 0; privMethod(); /* Error */ }
    void bothOK()    { pub = 0;  pubMethod();  /* OK    */ }
} derived;
void bothError() { derived.priv = 0; derived.privMethod(); }
void bothOK   () { derived.pub = 0;  derived.pubMethod(); }
```

Inheritance access specifiers: `private`

When you omit the access specifier, the access specifier is assumed to be `private`. The `private` specifier follows the following rule.

- Private member of the base class stays private *to the base class*. Same as before.
- Public member of the base class are still accessible to the derived class. However they are no longer part of the interface of derived class. I.e. cannot be access from outside.

```
class Base { ... };
class Derived : private Base {
    void bothError() { priv = 0; privMethod(); /* Error */ }
    void bothOK()    { pub = 0;  pubMethod();  /* OK    */ }
} derived;
void test() {
derived.priv = 0; /* Error */ derived.privMethod(); /* Error */
derived.pub = 0;  /* Error */ derived.pubMethod();  /* Error */
};
```

Member access specifiers: `protected`

The first thing you need to hear about this keyword, is **DO NOT ABUSE IT**. Use it with extra care. The fact that base class private member is not accessible to derived class sometimes causes in convenience. We would like something that:

- Not accessible by the outside world.
- Accessible for derived class, if inherited as public.

The key word that satisfies such need is `protected`.

```
class Base { protected: int i; int prot();};
class Derived : public Base {
    void bothOK() { i = 0;  prot();  /* OK */ }
} derived;
void bothError () { derived.i = 0; derived.prot(); };
```

We would really like to restrain ourself from using the keyword, since allowing other class to access a private comes with the possibility of the other class breaking invariants.

Two aspects of inheritance

There are two aspects of inheritance, namely:

Inheritance of code: reusing code

The derived class now comes with the same set of "code", or contents of the base class. This essentially saves us time and effort of coding same thing again and again.

Inheritance of interface

With public inheritance, the derived class will have the same interface (well, at least same method signatures...) as the base class. If used corrected, this creates a *subtype*.

Private inheritance is sometimes referred as *implementation inheritance*, since it allows the derived class can reuse the code of the base class (as if the base class is a private member variable of derived class), without exposing the base class.

We will assume public inheritance in the rest of this chapter.

Remark: Inheritance & subtyping

There are two remarks that I would like to make. First of all, subtypes does not have to be created from inheritance. In the following code, class B is definitely a subtype of class A.

```
class A {public:void quak(){puts("Hello.");} };
class B {public:void quak(){puts("Hello.");} void nop();};
```

On the other hand, having identical interface (or inheritance) does not guarantee subtyping relation. For example (why?):

```
class A { protected: int a = 0;
          public:    int add(int i){ return i + a;} };
class B : public A { public:  B() : a(10) {} };
```

Although inheritance is neither a sufficient nor a necessary condition of subtyping relation, it IS the only subtyping method supported by C++ (without a hack) in runtime. We will see this shortly after.

Pointer, reference and sub classing

From our previous discussion we see that there is no definite rule between a class being as subclass and a class being a subtype. However, from the language perspective, C++ simply **trusts** the programmer that every subclass is indeed a subtype. This is visible from the following rules:
Suppose we have `class Base` and `class Derived`, with `Derived` being a subclass of `Base`. We have the following rule.

- Derived class pointer compatible to base class.
- Derived class instance compatible to base class (possibly `const`) reference.
- *You can assign a derived class object to a base class object.

Two remarks. 1) The last rule needs further explanation. 2) Those rules are NOT true if reversed. For example, you cannot (normally) assign a base class object to derived class object. Assigning a base class pointer to derived class pointers needs special casting.

Pointer, reference and sub classing: Example

Code in code/rc10compatible

```
// class.h
class Base {
public: string str;
Base(const Base& base)
: str(base.str) {
    cout << "cp Base\n"; }
void print() {
    cout << a << endl; }
};
class Derived : public Base {
public: // ...
Derived(const Derived& d)
: Base(d) {
    cout << "cp Derived\n"; }
};
```

```
// class.cpp
void test() {
    Derived d; d.a = "hello";
    d.print(); /* hello */
    Base* bp = &d; bp->a = "ha";
    d.print(); /* ha */
    bad(&d); d.print(); /* bad */
    good(d); d.print(); /* good *
    pbase(d); /* good */
}
void bad(Base* base) {
    base->a += "bad"; }
void good(Base& base) {
    base.a += "good"; }
void pbase(const Base& base) {
    base.print();}
```

Pointer, reference and sub classing: The third rule

Now the previous example demonstrates the first two roles. We know how things worked with reference and pointers. To explain the third rule, we need to understand first how synthesized methods worked with inheritance. Note that here we mainly focus on how copy constructors works. The case for the assignment operator overload is very similar.

Here we will use the following base class as an example:

```
// base.h
class Base {
    string str;
public:
    Base() { cout << "default base\n"; }
    Base(const Base& other) { cout << "copy base\n"; }
};
```

Code in `code/rc10synthesize`

Pointer, reference and sub classing: Synthesized methods

```
// ok.cpp
class Derived1 : public Base {};
class Derived2 : public Base {
public: Derived2() = default;
        Derived2(const Derived2& d2) : Base(d2) {
            cout << "copy derived 2\n"; } };
int main() {
    Derived1 d1; Derived1 d1c(d1); /* cp base */
    Derived2 d2; Derived2 d2c(d2); /* cp base; cp d2 */ }
```

A synthesized copy constructor will do things almost identical to synthesized default constructor.

- **Copy construct** the base class. See `Derived1`.
- **Copy construct** every member, if there is any.
- Call the copy constructor of the class.

The case of `Derived2` shows how we do this manually.

Pointer, reference and sub classing: Synthesized methods II

```
// error.cpp
class Derived3 : public Base {
public: Derived3(const Derived3& d3) : Base(d3) {
            cout << "copy derived 3\n"; } };
class Derived4 : public Base {
public: Derived4() = default;
        Derived4(const Derived4& d4) {
            cout << "copy derived 4\n"; } };
int main() {
        Derived1 d3; Derived1 d3c(d3); /* Compile Error */
        Derived2 d4; Derived4 d2c(d4); /* dft base; cp d4 */}
```

Here are mistakes that people usually make. Without default constructor (`Derived3`), since you already provided a constructor, compiler won't synthesize default constructor for you! Without copy constructing the base (`Derived4`), the compiler will treat it as if the cpctor is a usual constructor, defaulting constructing the base and all members.

Pointer, reference and sub classing: Copying

Code in code/rc10compatible

```
// class.h
class Base {
public: string str;
Base(const Base& base)
: str(base.str) {
    cout << "cp Base\n"; }
void print() {
    cout << a << endl; }
};
class Derived : public Base {
public: // ...
Derived(const Derived& d)
: Base(d) {
    cout << "cp Derived\n"; }
};
```

```
// copy.cpp
void test() {
    Derived d; d.a = "hello";
    Base b = d;    /* cp Base */
    passByVal(d);
    /* cp Base; hellofoo; */
    d.print() /* hello */
    Derived d2 = d;
    /* cp Base; cp Derived; */
    Derived d2 = b; /* Error */
}

void passByVal(Base base) {
    base.str += "foo";
    base.print(); }
```

Digression: Inheritance and memory map

We now digression to the question of how does this work exactly. If we print the address of the objects in our previous example:

```
void addr1() {Derived d;  Base* b = &d; cout << b; }
void addr2() {Derived d;  Base& b = d; cout << &b; }
```

You will see the same address being printed over and over again. Consider a function a function taking an base class pointer as argument. The function will have no idea the address passed to it, is indeed a base class object, or a derived class object "disguised" as a base class object.

The derived class object always "embeds" a base class object at the begging of its corresponding memory region. Memory layout of Derived object looks like below.

```
struct Derived { Base base; /* Derived members */ };
```

This also explains why base class objects are initialized as if they are member variables, because to a degree they are!

Introducing new methods to create subtypes

Now we go back to the question of subtypes. Recall the 3 three ways of creating subtypes. Two involves modifying existing methods. The last one involves adding extra operations. Adding extra operations is simply adding methods to the derived class.

The term "sub-type" is misleading?

Adding new methods usually enhances the class, yet the enhanced version is called a "sub-type", as if it is inferior in functionality. However, the derived class is not only *enhanced*, but also *specialized*. It is less general than its superclass, which is "sub-".

Should I use "protected"?

If there is no shared data between derived class and base class, e.g. global variables, protected member, static members etc, adding a new method always results in a sub type. Though sometimes you want share some data structure.... Be careful!

First (failed) attempt towards new sub-typing

We now make an attempt of the the other two ways of subtyping. Both relaxing preconditions or tightening postconditions requires modifying existing methods. Since we cannot modify existing functions, our naive attempt is to define a function with the same name as the function being modified, hoping that this function would somehow "replace" the original function.
Consider the following example from your lecture slides. Suppose we have an `IntSet` and `SortedIntSet`.

```
class IntSet {
protected: int count, *data;
public: /* Other methods ommited */
  void insert(int i) {cout << "IntSet\n"; ... } };
class SortedIntSet : public IntSet { public:
  int  max() { /* Get maximum (last) element */ }
  void insert(int i) {cout << "SortedIntSet\n"; ...} };
```

First (failed) attempt of sub-typing, Con't

```
void test1() {
    SortedIntSet set;
    /* A number of insert / del / max */ }
void insert100(IntSet& set) { set.Insert(100); }
void test2() {
    SortedIntSet set; set.insert(10);
    insert100(set);  /* Will print IntSet */
    cout << set.max() << endl; /* very likely be 10 */ }
```

When you run functions like `test1()`, everything will be fine. You cannot break things if you just stick with functions like `test1()`. However, when you do function `insert100()`, a function that assumes sub-typing, you might break your instance's invariance! Breaking the invariance is bad. We know that very well. But we need to look closer. We must ask, how did we fail? And what exactly are the damage?

Static binding causes the problem

Recall how C++ links member function calls?

- All member functions are just usual functions.
- When a member function call happens, the compiler links the function according to the type of the class instance, passing the object as the first argument.

Now in `insert100()`, the method `insert()` is called on object `set`. `set` is an instance of `IntSet`. In this case, the compiler will choose the function `IntSet::Insert()`. Remember that the compiler have no idea what is actually referenced by `set`. When it compiles `insert100`, all it knows is that `set` refer to an object of `IntSet`. It doesn't care if this object is part of a larger object. In fact, up till this point, when you make a function call in the code, the actual function being called is always know at compile time. **The process of binding a function call to the actual definition is static.**

Name hiding

On the other hand, you could always:

```
void test3() {
  SortedIntSet set; set.insert(10; /* SortedIntSet */
  IntSet& is = set; set.insert(10); /* IntSet */)}
```

Using a simple reference, you can always access the function you should have replaced. In fact, the `SortedIntSet` actually exposed 2 sets of interfaces:

- `IntSet::insert(int)`, inherited from `IntSet`.
- `SortedIntSet::insert(int)`, which is defines.

Defining a method with same DOES NOT replace the original function. Instead, it only adds a new function! It's just when you say `insert()`, the compiler has two choices. Instead of complaining, it chooses the "closest" match. The second function "hides" the name of the first one. This is called *name hiding*. You can expose the first one with a `using` statement.

Name hiding is not sub-typing (in general)

- `IntSet::insert(int)`, inherited from `IntSet`.
- `SortedIntSet::insert(int)`, which is defines.

We finally examine the damage. How bad is the name hiding?

- First of all, substitution rule still works here. Replacing `IntSet` with `SortedIntSet` will not break any existing code. So `SortedIntSet` is (arguably!) a subtype of `IntSet`.
- We can also think in terms of invariants. All methods exposed by a class must always maintain invariance. In our case `SortedIntSet` assumes a different set of invariance (the ordering), `IntSet::insert` will break the invariance. That's what actually goes wrong.
- On the other hand, in this case `SortedIntSet::insert` will not break the invariance of `IntSet`. However, since two classes are sharing data, this will NOT be the case in general. If the `SortedIntSet::insert` breaks the invariance of `IntSet`, they will be no subtyping.

RC Week 10

Virtual-ness and runtime polymorphism

The need for polymorphism

Why are we spending so many time on the name hiding things. Well, the example motivates the following discussions.

- From the subtyping perspective, we want something that allows us to implement other two ways of subtyping.
- As we have shown, doing this often requires actually "replacing" the method call, not just hiding it.
- For functions using that call, let's say `insert100`, the actual function being called would be dependent on what the reference actually is. The fact that identical code behaves differently for different object is called *polymorphism*.
- We will only know that the actual object being referenced at runtime, which means the binding process must be dynamic, i.e. happens also in the runtime. What we want is a form of *dynamic polymorphism*.

All that boils down to the what is known as *virtual* functions.

Apparent type and *actual type*

Before we head out to define what virtual functions are we first start out to make two definitions:

Apparent Type Apparent type is the type annotated by the type system. It is the static type information. It is the you remarked to the compiler.

Actual Type It is the data type of the actual instance. It is the data type that describes what exactly is in the memory.

In our previous example, in function `insert100()`, the apparent type of the variable `set` is `IntSet`, while what's in the memory is actually a `SortedIntSet` (The actual type).
If we note again we have discussed using these new term, normally C++ resolves function bindings based on apparent type. Since apparent types are always known at compile time, function binding will be able to be resolved at compile time.

Dynamic *polymorphism* through *virtual*-ness

What we want is dynamic function binding, the ability to bind a function call based on an object's actual type, instead of the apparent type. This is done through the `virtual` keyword. Using previous example:

```
class IntSet { /* ... */
public:
    virtual void insert(int i) { /* Impl ommited */ }
};
```

The above syntax marks `insert` as a `virtual function (method)`. Virtual methods are methods replaceable by subclasses. When a method call is made, if the method you are calling is a virtual function (based on the apparent type), the language bind the call according to the actual type. In this way, the function `insert100` achieves dynamic polymorphism, the ability to change its behavior based on the actual type of the argument.

Inheritance and keyword `override`

Virtual functions are functions that allows being replaced by subclasses. This is done by the subclass defining a function with identical name as the base class.

```
class SortedIntSet : public IntSet {
public: void insert(int i) { ... } ...  };
```

The act of replacing a function is called *overriding* a base class method. As you can imagine, if you somehow forgot the `virtual` keyword in the base class, overriding would become name hiding, which is subtle but dangerous. C++ provides a keyword to allows make clear your intent:

```
void insert(int i) override { ... }
```

Would cause the compiler to verify if a function is indeed overriding a base class method. If the base class method is not a virtual function, compiler will complain. The keyword is introduced in C++11. **It is considered a best practice always mark override whenever possible**.

Fix our example with `virtual`

With the changes we have noted, now this is fine

```
void insert100(IntSet& set) { set.Insert(100); } /*same*/
IntSet is; insert100(is);              /* print 'IntSet' */
SortedIntSet sis; sis.insert(10);
insert100(sis); cout << sis.max(); /* SortedIntSet; 100 */
```

Now when `set.Insert()` is called inside `insert100`, the apparent type is still `IntSet`. however, the compiler finds out that `insert` is a virtual method. Thus it generates code that makes the call based on the actual time at runtime.
At runtime, when `IntSet` instance is passed, the actual type is just `IntSet`, so it binds it `IntSet::insert()`. The other situation, the actual type is `SortedIntSet`, so it binds it to `SortedIntSet::insert()`. Now what if modify `insert100` to?

```
void insert100(IntSet /*byVal*/ set) {set.Insert(100);}
```

Inheritance and keyword `final`

On the other hand, *virtualness is automatically inherited*. For instance, if have class and another function:

```
class QuickSortedIntSet : public SortedIntSet { ...
public: void insert(int i) { ... } };
void insert200(SortedIntSet& set) {set.Insert(200);}
```

You DO NOT need to change `SortIntSet` to achieve below:

```
QuickSortedIntSet qsis;
insert100(qss); insert200(qss); /* QuickISIS*/
```

Sometimes, for instance, `SortedIntSet` would like to stop its methods being override by further subclass (Why?). In this case, you can mark its method using the keyword `final`.

```
/* SortedIntSet */ void insert(int) override final {...}
```

The order of two keyword does not matter. Trying to override a method with `final` causes compilation error.

Remark: Use overriding with extra care

The overriding mechanism provides the programmer with a mechanism to change the **base** class behavior. The emphasis is that you not only able to modify your own class, *you would also be able to modify the behavior of another class, with it knowing it!* It sounds dangerous, and you make make careful use of it.
First of all, overriding not always creates subtypes. For instance if we override `IntSet::insert()` with an the following:

```
/* SortedIntSet */ void insert(int) override {numElts--; }
```

This function will very likely break the invariant of `IntSet`. The substitution would fail, and there goes the sub typing relations.

Reuse of implementation and object composition

Secondly, virtual methods are almost the most misused feature of C++! We have noted inheritance has two affects: reuse of implementation and interface. Many would "accidentally" involve the latter, while what they actually want the first!
If you just want the implementation, there are two ways to go:

- Private inheritance, as we have discussed before,
- Or even better, declare the base class object as a member instead of deriving from it. This is called *composition*.

Let's take a look at an example:
Suppose we have a `class Graph` that abstracts the concept of a graph. It supports `Graph::InsertNode(string name)` and `Graph::addEdge(string node1, string node2)`. Suppose we have a class `Binary` tree that inherits `Graph`. It supports one more operation `Tree::GetRoot()`. The question is whether `Tree` is a sub-type of `Graph`?

Remark: Differentiating `IS-A` and `HAS-A`

Unfortunately the answer is negative. The reason should be obvious. There is essentially no requirement on where the edges and nodes are (precondition is weak). But you can't add arbitrary edges and nodes must connect to at most 2 other nodes (stronger precondition). So a `Tree` is not a `Graph` in the subtype sense.

However we would naturally see that a `Tree` type must have a lot common code as the `Graph`. We would like to avoid rewriting them. There are two ways to do this:

- Through `private` inheritance (implementation inheritance). We would not discuss this.
- By setting `Graph` as a member of a `Tree` type. We forward calls to `Graph`. This is a typical case of a `Has-A` implementation.

Compiling technology supporting virtualness

Note we are now in the field of undefined behavior. We are now going to discuss how virtual functions are implemented internally. Knowing this will help you solve exam questions. However, what we are going to discuss here are not ground truth. I.e. things could change from platform to platform.
Internally, virtual functions are implemented using VTables:

- Virtual methods are grouped together in an array of function pointers, as if they are member variables. This array is called *Virtual Table*, or *vtable*
- When the compiler makes a virtual method call, it calls the function pointer in the corresponding entry of the vtable.
- When derived objects instantiates itself, it replaces entries of function it need to override in the vtable, with its own implementation.

Virtual Tables and performance: No free lunch

Virtual function comes with preformance hit

- The cost of one extra layer of indirectness. There exists one more pointer dereference to find the target function. That's one more memory access.
- Cost of unknown call target. Modern processors will "prefetch", or guess the future instructions and execute them in advance. Since the function call target is unknown, this will not be possible for virtual functions
- Cost of unable to inline methods. For simple methods, compiler will try to inline them. Since the binding happens at runtime for virtual methods, this is no longer possible.

Using the `final` keyword will help. If the compiler is able to determine the actual type, it may choose to preform *de-virtualization*. Those costs could be quite huge if the method is used frequently. In old time the cost is often indurable. Modern computers are more powerful, things get better.

Using VTable to solve problem

```
struct Foo {
    virtual void f() = 0;
    virtual void g() = 0;
};
struct Bar : public Foo {
    void f() {};
    void g() {};
    void h() {};
};
struct Baz : public Bar {
    void    f() {};
    void    g() {};
    virtual void h() {}; };
struct Qux : public Baz {
    void f() {};
    void h() {}; };
```

```
Bar bar; bar.g(); // Bar::g
Qux qux; qux.g(); // Baz::g
Baz baz; baz.h(); // Baz::h
Foo& f1 = qux; f1.g();
// Baz::g
Bar& b1 = qux; b1.h();
// Bar::h
Baz& b2 = qux; b2.h();
// Qux::h
Bar* bar = &Quz; bar->h();
// ?
const Foo& cbar = Baz;
cbar.g(); // ?
```

Using VTable to solve problem

The idea will be to draw out the VTable for each object. E.g. When you run `b1.h()`, you would find that the apparent type of `b1` is `Bar`. And `Bar` does not have an entry for `h()`. So this is static linkage. On the other hand, if you run `b2.h()`, `b2` is `Baz`. We thus follow the vtable to find the function call.

Vtable for `Qux` instance:

```
struct Qux {
 struct Baz {
  struct Bar {
   struct Foo {
     vtbl f() = Qux::f;
     vtbl g() = Baz::g;
   }
  }
  vtbl h() = Qux::h
 }
}
```

Pure virtual functions and classes

It is possible that we do not supply a implementation when defining a base class. In this case, the corresponding entry in the vtable would simply be left unfilled:

```
/* IntSet */ virtual void insert(int i) = 0;
```

In this case we say the method is *pure virtual*. If a class contains **one or more** pure virtual methods, we say the class is a *pure virtual class*. You only need to have one pure virtual function for a class to be "purely virtual".

You **CANNOT** instantiate a *pure virtual class*. This is somewhat obvious. If you ever have an instance of `IntSet`, what should happen when you call `IntSet::insert()`?

Pure virtual class are also called *abstract base classes*, or *interfaces*. We will explain this terminology shortly after. It is often that the name abstract base classes starts with a case letter `I` (/ai/) for *interface*.

Abstract base classes

Why would we ever wanted a pure virtual classes since we cannot instantiate it? Because we would want to model abstract *concepts*. We would like to say things like *Matricies are subclass of summable object*. We would like to increase code reuse. Consider the following class:

```
class ISummable { public:
    /* Add item x to itself */
    virtual void add(ISummable& x) = 0; };
```

This class models the objects that are summable. Based on this modeling, we could write the following very general function:

```
void sum(ISummable elem[],size_t size, ISummable& rst) {
    for (int i = 0; i < size; ++i) rst.add(elem[i]); }
```

This will work for anything that is `Summable` object. When a class derives from an interface and provides an implementation, we say it `implements` the interface.

Programming with interfaces

Abstract base classes provides very strong decoupling technique, in a large project:

- One team of engineers write algorithms using the interface.
- Another, independent team of engineer write classes that implements the interface.
- The only coupling point is the interface itself. Once the interface is agreed upon, two teams can work independently.

This is even better if you ever would like to update your code:

- To support new features, simply write **more** classes. There is no need change any existing code.
- To update existing feature, only rewrite and recompile the part related. I.e. you only recompile the algorithm, or the classes implementing the interface.
- Dynamic linking technology, you could update your software by shipping the end user only **part** of the entire binary.

Programming with interfaces

Modern software engineering technologies usually "abuses" the Liskov Substitution rules by write software that depends only on interfaces. In this way maximum utilization is achieve. For instance, we now provide an example of what is known as *"Factory" design pattern*. Let's say you are asked to write a encryption program that intends to support a number of ciphers. You first abstract out what a cipher is:

```
class EncryptionEngine {
public:
  virtual string name() = 0;
  virtual string generateKey() = 0;
  virtual string encrypt(...) = 0;
  virtual string decrypt(...) = 0;
  virtual ~EncryptionEngine() = default;
};
```

Programming with interfaces, II

Then you write a few engines, and write a *factory method*:

```
EncryptionEngine *
encryptionEngineFactory(std::string engine) {
  engine =string_tolower(engine);
  if (engine == "aes") return new AesEngine;
  if (engine == "rc6") return new Rc6Engine;
  if (engine == "dummy") return new DummyEngine;
  throw UnknownEngine(engine);
}
```

In the main, use a unique_ptr to preform memory management.

```
string algorithm = getEngineName();
unique_ptr<EncryptionEngine> engine(
  encryptionEngineFactory(algorithm)
); cout << engine->enrypt("abc", "DEADBEEF");
```

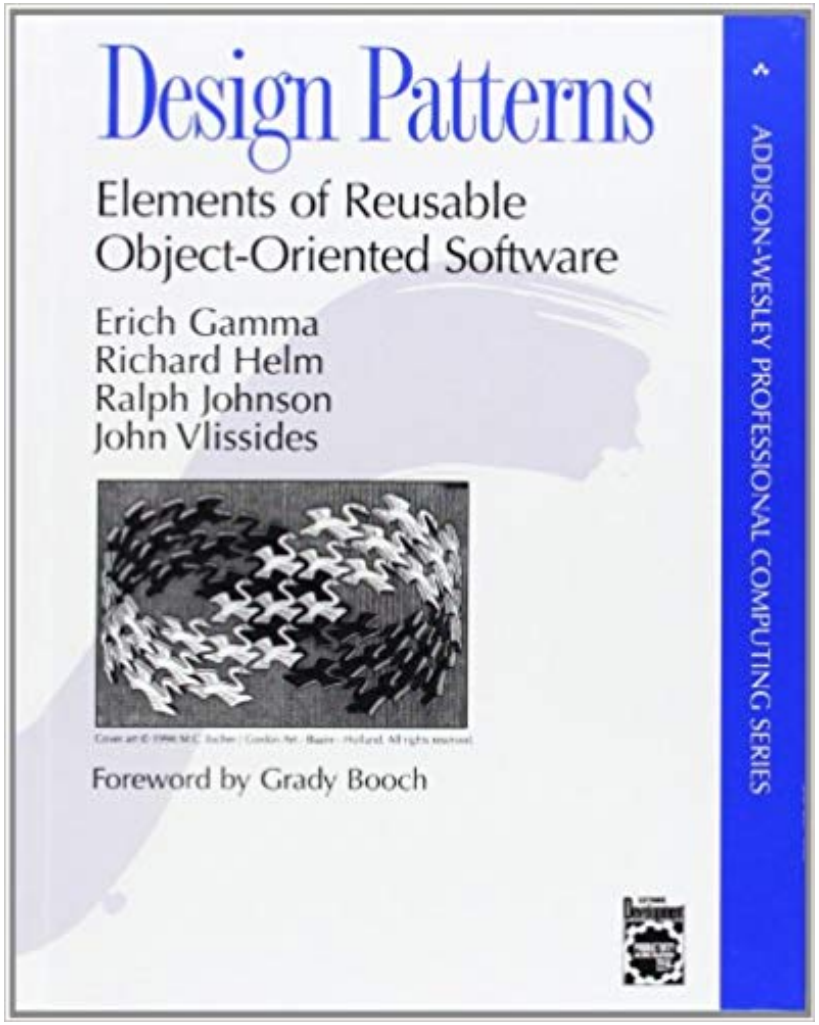
Programming with interfaces, III

This method gives you a number of advantages:

- First of all, it allows you to decouple the encryption algorithm from the code that handles you user input. The user input may come from GUI, input or a file. We simply don't care. The main routine would be exactly the same
- Then, you no longer need to duplicate any code to support more algorithms. This prevents possible mistakes.
- Finally, this gives you the flexibility of dynamically add, remove, disable, enable some algorithms. You could complicate your factory methods, for instance, check if your user have paid for your program. The list of available engines could come from Internet, or from a configuration file.

Remark on *design patterns*

There exists a number (over 20) of different (object oriented) design patterns. The urge for design pattern rise with the popularizing of JAVA. Almost all of them relies on interfaces. There exists a book written by GOF called *Design Patterns* listing all of them. The book is considered one of the most classical text book for software engineering.



Is it a subtype?

Designing whether something is or is not a subtype of a particular super type is probably in the heart of program structure design.

A Remark

The problem of whether two objects form a "IS-A" relation must be understood in a realistic context. Consider two classes `RegularIntSet` and `SortedIntSet`. Suppose both of them supports a `max()` function.

- You can argue that `RegularIntSet` is a sub-type of `SortedIntSet` because they support the same set of operations.
- You can also claim that `RegularIntSet` is NOT a sub-type since the `max()` of `RegularIntSet` is significantly slower than it's counterpart. One could argue the performance is part of the abstraction.

RC Week 11

Generics, polymorphism and templated containers

Preliminary: A container of pointer

As a kind reminder we briefly discuss containers of pointers here.
The key idea in the whole discussion again is the problem of the ownership.
One must be crystal clear that a container of pointer actually owns the object, which means:

- Pass to a container only things you have ownership.
- When you pop from a container, you must take over ownership.
- When an object has been passed to the container, it's the container's responsibility and authority to treat the object. You should not use the object in any way since you no longer owns the object.

With this in mind we take a look a few slides:

Templated Container of Pointers

```
template <class T>
class List {
  public:
    ...
    void insert(T v);
    T    remove();
  private:
    struct node {
        node *next;
        T     o;
    };
    ....
};
```

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T    *remove();
  private:
    struct node {
        node *next;
        T    *o;
    };
    ....
};
```

Container of Pointers

Use

- To avoid the bugs related to container of pointers, one usual "pattern" of using container of pointers has an **invariant**, plus three **rules** of use:
  - **At-most-once invariant**: any object can be linked to at most one container at any time through pointer.
  1. **Existence**: An object must be **dynamically allocated** before a pointer to it is inserted.
  2. **Ownership**: Once a pointer to an object is inserted, that object becomes the property of the container. No one else may use or modify it in any way.
  3. **Conservation**: When a pointer is removed from a container, either the pointer must be inserted into **some** container, or its referent must be **deleted**.

Do not repeat your self

Containers are objects to contain other objects, they do not have an intrinsic meaning. From now on we focus on Containter of pointers (though we use a container of value in this example). Consider the example of a container of `char` and `int`:

```
struct node {
    node *next;
    int   v;
};

class IntList {
    node *first;
  public:
    void insert(int v);
    int remove();
    ...
};
```

```
struct node {
    node *next;
    char  v;
};

class CharList {
    node *first;
  public:
    void insert(char v);
    char remove();
    ...
};
```

Generics and polymorphism

We would imagine they share the same implementation. Now our DRY principal suggest we should find a way to abstract out the almost identical implementation.
Now we must familiar ourself to two widely used terms:

Generics A generic algorithm is an algorithm that does not depend on a specific type. Type can be specified later.

Polymorphism Polymorphism is a property of code. A piece of code is said to be polymorphic, if it works on "a range of" types.

Clearly polymorphism is a technique to achieve generics. What we need to implement here is a generic algorithm (data structure), and we are going to use polymorphic code to achieve our purpose.

Polymorphism tool box

We first start by examining our polymorphism toolbox. In general there are 3 ways achieve (different levels) of polymorphism. Polymorphic code always involve a time where the "real" implementation starts to step in. In general there are two timings:

Compile-time The compiler "injects" the real implementation into our polymorphic code when it is being compile. Sometimes called *static polymorphism*.

Runtime The polymorphic behavior is determined when it is needed at runtime. A typical example is `virtual` functions. It introduces overhead.

We now examine our choices:

Overloading Function / operator overloading. Limited power.

Subtyping Virtual functions. Polymorphism through dynamic dispatch. This is runtime polymorphism.

Parametric Templates in C++. Type as a parameter.

Polymorphic containers

Both of the second and third solution can be used to solve our problem at hand. We first look at the second one. We now introduces `Polymorphic containers`.
In a polymorphic container, the basic idea is to create a universal super type. Every type it it's subtype. And a container is written in terms of the universal super type.
The universal super type looks like the following:

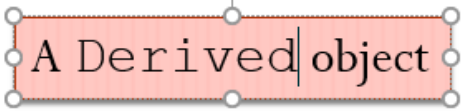
```
struct Object {
        virtual ~Object() {}
        virtual Object* clone() = 0;
}
```

Every class to be pushed into the container should be a subtype of this class. There are two questions: why are both methods virtual? Why do we need the `clone()` method.

Polymorphic containers

Note when we copy construct the constructor we also use copy copy constructor to copy the contained objects.

```
void List::copyList(node *list) {
  if (list != NULL) {
    Object *o;
    copyList(list->next);
    o = new Object(*list->value);
    insert(o);
  }
}
```



But unfortunately this simply cannot work. Remember that constructors cannot be virtual, since it is not possible for a base class object to setup invariants for a derived class object.
Note above code does compile (why?). However what we get from such construction is an empty, useless `Object` class instance.

Polymorphic containers: copying

The solution is to ask the derived class instance to make a copy itself. This gives us the `clone()` method in the previous slides. This methods asks the object to make a copy of itself and returns a ptr to base class. Now our copy method can be written as follows:

```
void List::copyList(node *list){
    if (list != NULL) {
        Object *o; copyList(list->next);
        o = list->value->clone(); insert(o); } }
```

Note this design makes sense from an interface point of view. In order for an object to be put inside a container it must be copyable. For normal container this is checked by the compile for the copy constructor (since the compiler knows the type in question). For polymorphic container since the compiler do not know what type is contained in advance so we must specify the constraint by ourself, namely through adding a `clone()` method.

Type Erasure

We would like to make a final note. Consider when you pop something out of the container. The container only knows that the value is of type `Object*`. But you know it is actually an instance of `Derived*`. And most likely you need to use it as a `Derived` object. You need to transform a base class pointer to a derived class pointer.

This way you will need a cast:

```
Derived* bp = dynamic_cast<Derived *>(list.remove());
```

The `dynamic_cast<>` checks at runtime if the pointer of base class is actually a pointer of derived class. If not so it returns `nullptr`.

Type Erasure

Note all this must happen at run-time. No checks will be done at compile time. This has both up-side and down-side:

- Since actual type is involved only when used this allows for heterogeneous containers. The contained object does not have to of the same type. This is a huge gain.
- However the cost is also significant! Since all checks are done at runtime, there is not type check at compile time either. Type mismatch will be deferred to runtime. This makes writing type safe code much harder!

Since essentially we are "erasing" the actual type information of the objects when we put them into the container, this strategy is often called *type erasure*.
Problematic as it seems, `JAVA` choses to use type erasure when it comes to generics. This is one of the debatable feature of the language.

Poor man's template C++

Now we turn to `template`, which is so called *parametric polymorphism*. Before we introduce that in C++, I would like to familiarize you with a piece of code in C++:

```
#define SWAP(type) swap_##type
#define SWAP_IMPL(type) \
static void SWAP(type) (type &x, type &y) {\
    type t = x; x = y; y = x; \
}
SWAP_IMPL(int);
SWAP_IMPL(double);
int main() {
    int x =10, y =20;          SWAP(int)(x, y);
    double p = 0.0, q = 1.0; SWAP(double)(p, q);
}
```

Poor man's template in C++

The code seems very cryptic at first. We now look at what it does by expanding the macros. `SWAP(int)` would be expanded into `swap_int`. The two sharp signs means concatenating. An `SWAP_IMPL(swap)` will be expanded into.

```
static void swap_int (int &x, int &y) {
    int t = x; x = y; y = x;
}
```

This is a very straight forward swapping two integers. As you can see this way we can reuse the `swap` function easily for different types by simpling use `SWAP_IMPL` to create an implementation for different types and call them with `SWAP()`.
Further more we can put the two macro definitions into a header file and include the when we need to.

Templates

A template in C++ does exactly the same (for now. It if far more complicated if you explore deeper). When you write a template, for example:

```
template <class T>
class List {
  public:
    ...
    void insert(T *v);
    T    *remove();
  private:
    struct node {
        node *next;
        T    *o;
    };
    ....
};
```

- `T` is called a template parameter. It is usually a `class`, but it can be a value (e.g. an `int`).
- The traditional way of specifying the type parameter is through `template<class T>`. But after C++11 one could use `template<typename T>`.
- A template is much like a macro. When it comes to using it, the compiler uses an actual type to replace the type argument, and generate a version of implement for that type.

Template instantiation

When you use a templated class, for example:

```
List<int> intList;
```

The compiler will automatically produce a version of the actual code with the `int` as the parameter. This process is called *template instantiation*. Template instantiation is conceptually the compiler writing `SWAP_IMPL(int)` for us.
We would like to make a footnote here. A template is by no means a type. A template is not a type since you cannot have a variable of `List` type clearly. It can be `List<int>` or `List<double>`, but simply is not `List`. A template is incomplete. It becomes a type when you put an actual type argument into it.
In some literature templates are refered as a *dependent type*. You should notice this behavior is very much like a *function*, which takes in a type and spits out another type. This function is "sort of" evaluated in compile type.

A comment on the syntax

Templates and their implementation is almost always written in a header file (I would say always if not for the project). Libraries consists only of template classes are often called header libraries. The reason behind this should be simple from the previous poor-man's template example.
There are in general two ways of implementing a templated class method.

- You can implement it inside the declaration.
- You can implement separately.

For the second method:

```
template <class T> void List<T>::isEmpty() {...}
```

The beginning `templated <class T>` must be there. Note the method is implemented within the namespace `List<T>`. This makes sense since namespaces corresponds to types, and `List<T>` is a type, `List` is not.

A comment on the syntax

We would like to make a comment on the following slide (ch21,30):

- The function header of the constructor is

```
List<T>::List()
List<T>::List(const List &l)
```

Must have <T>! No <T>! No <T>!

- The function header of the destructor is

```
List<T>::~List()
```

Must have <T>! No <T>!

- The function header of the assignment operator is

```
List<T> &List<T>::operator=(const List &l)
```

Must have <T>! No <T>!

30

A comment on the syntax

We would like to make a comment on the following slide (ch21,30):

- The function header of the constructor is

```
List<T>::List()
List<T>::List(const List &l)
```

Must have <T>! No <T>! No <T>!

- The function header of the destructor is

```
List<T>::~List()
```

Must have <T>! No <T>!

- The function header of the assignment operator is

```
List<T> &List<T>::operator=(const List &l)
```

Must have <T>! No <T>!

30

A comment on the syntax

You should find this to be very weired. Take a look at

```
List<T>& List<T>::operator=(const List &l);
```

Now what is the type of `l`? We have commented that `List` does not name a type. But clearly `l` is function argument and it must have a type. The professor commented "No <T>!".
After a little research this is called a injected-class-name. Basically if you write `List` is a templated class, it will be inferred as `List<T>`, so we could use `List` as a shorthand of `List<T>` inside class declaration. We would like to note:

- It is syntactically correct to write `List<T>`.
- It is also the recommended way as long as it does not impact readability too much, since it enhances clarity and makes more sense.

Note the name of the ctor/dtor must be `List`, not `List<T>`.

A comment on the syntax

We would like to emphasize on one thing, consider the following code

```
// Create a static list of integers
List<int> li;
// Create a dynamic list of integers
List<int> *lip = new List<int>;
// Create a dynamic list of doubles.
List<double> *ldp = new List<double>;
```

Essentially forms `List<int>` are no difference to a regular `int`, `IntSet` etc. Anywhere a type can be use, such form can be use. For example it is possible to inherit from a templated type:

```
class Foo : public: List<int> {...}
```

And then override some of its methods.

Compile time polymorphism trade-offs

Now we are at the core of compile time polymorphism. Templates are instantiated at compile time. The instantiation process is essentially the compiler deciding when to plug in the actual implementation. This gives us the following trade-offs:

- It enhance code safety, by a lot. After substituting the type parameter with the actual type the compiler will be able to perform type checks. Type mismatches will be caught so that code correctness is improved.
- It improves performance (compared to polymorphic containers). Since the dispatch is done at compile time there is no run-time overhead. Compile-time dispatch also allows for further (much more) optimizations.

Above two are the major reasons people use templates. This is also why standard libraries choose to use templates to implement generic containers, since C++ is a language that focuses on performance and static typing.

Compile time polymorphism trade-offs

Now we take a look at the down sides

- It prolongs compile time (a lot). The template system of C++ is so powerful that it self is Turing complete: meaning it is possible to write algorithms in terms of templates, or infinite loops that crashes the compiler.
- It increases the executable size. Each template instantiation creates more code. This gets worse combined with the C++'s "one file per object" rule, as we will see next.

The first point is not really a pure down side. Some people heavily exploit the first point and ends up creating a whole new area of programming, called *meta-programming*.
The second down side is generally OK for most desktop applications since people have large memories. But for embedded applications where memories are scares, this basically rules STLs out from the embedded use.
Another problem with templates is it complicates the compiler by a lot. However it is not a bad news for you.

Operator overloading

We now fill in the final piece of puzzle, i.e. operator overloading. We have introduced this in the previous slides. See page 253/254. Again just as a reminder there are two ways to write an overloaded operator:

- As a class method of the first operand (if there are multiple operands)
- Write as a normal method.

The second approach is very common for overloading the extraction / push operator on a I/O stream (pay attention to the return type of the function). For example:

```
ostream& operator<< (ostream &s, const MyClass &r) {...}
```

allows you to print to `cout` and `fstream` (why?) by:

```
Ofstream file(...); MyClass obj(...);
file << obj; cout << obj;
```

friend keyword

In our previous example operator<< might need to access private member of MyClass instance. You could provide an accessing operator for each of the member, but often it is not a good idea (why?).
One workaround is specifically grant operator<<(ostream &s, const MyClass &r) access to the protected members. This can be done by using the friend keyword:

```
class MyClass {
    friend ostream& operator<< (ostream &s, const MyClass &r);}
```

It doesn't matter where this is marked public or private. Note friend can also grant access to regular functions and even classes:

```
class MyClass {
    friend class Bar; friend int foo(double foo);}
```

Pay attention that friend is not mutual. If ClassA declares ClassB as friend. ClassB can access ClassA's private member, but the other way around doesn't work.

RC Week 12

Elementary data structures

Data structures 101

There are always two interleaved concepts when we talk about data structures:

Abstraction This is how they are used, for example, as a queue, or as a stack etc. This includes design choices of supporting random access, supporting enumeration etc.

Implementation This focuses on how the abstraction is implemented. For example a stack can be implemented by a array (if you dynamically resize it), or by a linked list, or by a doubly linked list, or by a hash map or a binary tree.

It is always that you first choose your abstraction base on what you need. Then you decide on the implementation. Of course there sometimes exists sort of "default" implementation, which is reasonably fast on most (if not all) operations. We will learn about them, but remember the that a `stack` is not always a linked list, but it is most likely to be.

Measuring performance

We need a way of measuring how fast an operation is. We do this by examining *time complexity* of an operation. We will study this in VE281, now we just give you a preliminary idea. Essentially we assume there is n element in the container, and we measure how much time it takes to perform such operation once.

Super Fast	Means operation takes $O(1)$ (constant time) no matter how many elements in the container.
Fast	$O(\log n)$. The base of the logarithm is irrelavent.
Reasonable	$O(n \log n)$ or $O(n)$. Acceptable for large dataset.
It depends	$O(n^2)$, (n^3). Depends on scale.
Emmmm	Any polynomial with order larger than 3
Exponential	$O(b^n)$ with arbitrary base. Usual operations taking exponential time is not acceptable unless no other methods are available.
NO!NO!NO!	Super exponential algorithms. For example $O(n!)$.

A stack

A stack is a sequential data structure that:

- (Usually super fast) insert and remove at front.
- Objects first pushed onto the stack comes out last. It is often called First-In-Last-Out. Thus it is often referred as a FILO.

It does not (need to) support

- Any form of random access. i.e. no insert / remove even read in the middle.
- Enumeration in any order
- Fast querying of any sort.

A stack describes an abstraction. Essentially any implementation that guarantees the above two property (abstraction) can be used to implement a stack.

A queue

A stack is a sequential data structure that:

- (Usually super fast) insert at front.
- (Usually super fast) remove at the end.
- Objects first pushed onto the stack comes out first. It is often called First-In-First-Out. Thus it is often referred as a FILO.

It does not (need to) support

- Any form of random access. i.e. no insert / remove even read in the middle.
- Enumeration in any order
- Fast querying of any sort.
- Remove at the front and insert at the end.

A queue describes an abstraction. Essentially any implementation that guarantees the above three property (abstraction) can be used to implement a queue.

A deque

deque stands for *double ended queue*. A deque is a sequential data structure that:

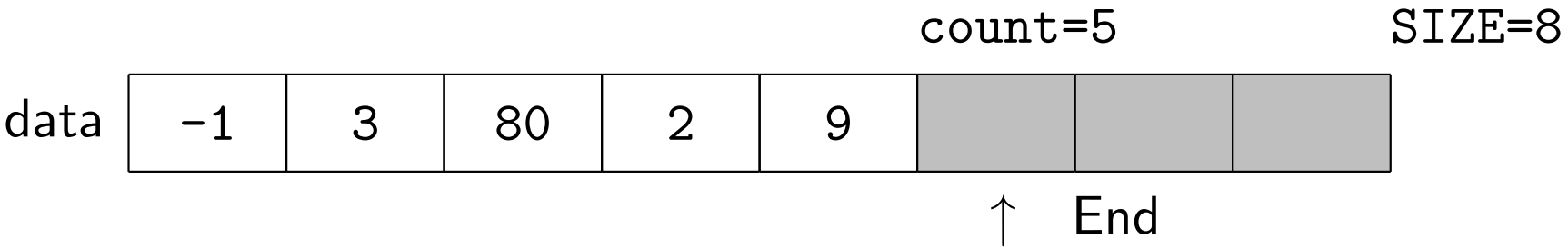
- (Usually super fast) insert / remove at front.
- (Usually super fast) insert / remove at the end.
- As it's name suggests it looks like a double ended queue. It's hard to rigorously (clear in the sense of mathematics) define what a deque does.

It does not (need to) support

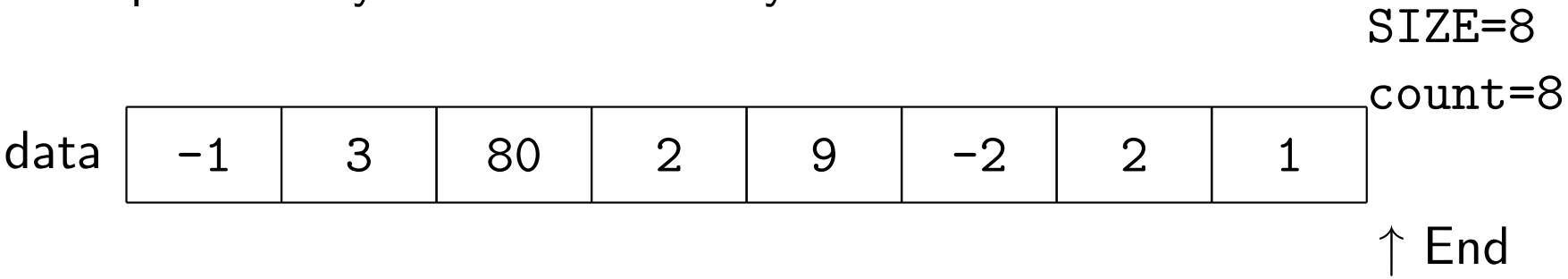
- Any form of random access. i.e. no insert / remove even read in the middle.
- Enumeration in any order
- Fast querying of any sort.

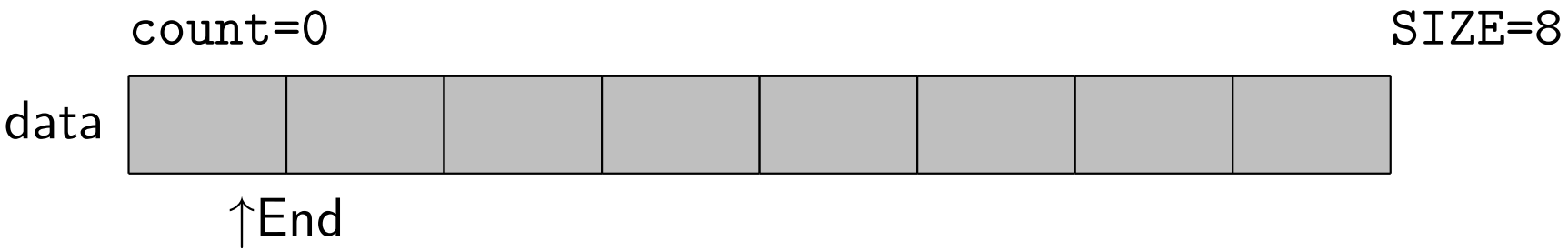
Implementation by Linear List

A *linear list* is simply a fancy name for `array`.



Note particularly the two boundary cases:





Left close right open convention

One convention that is used throughout any sequentially iterable data structure is the *Left-close-right-open*. Remember this is more than a direct array. The rule is essentially the following:

- The end pointer (or anything equivalent) points to one-pass-the-end location.
- The front pointer points to the first element (if there is one).

This rule is so important that the standard is willing to put a syntax hole just for this convention. The standard says:

If you have an array defined in the following manner:

```
int x[10] = {0};
```

- `&(x[11]) == x + 11`, `&(x[11]) - x`, `&(x[11]) > x` : are all undefined behavior.
- `&(x[10]) == x + 10`, `&(x[10]) - x == 10`, , `&(x[10]) > x` are guaranteed to hold.

Left close right open convention

Following the convention is very beneficial for us:

- `end` always puts to the next place to add new elements.
- Thus `data[size]` is always the next place to add new elements.
- `end - data == size` always hold.

These property are very helpful in keeping track invariants within a class. They are especially very helpful in writing loops

```
for (int i = 0; i < size; i++) /* Use data[i] */;
```

Or in terms of pointer

```
for (int* p = front; p < front + count; p++) /* Use *p */;
```

Or more generally, with `c` being an instance of `Container`:

```
for (ptr p = c.begin(); p != c.end(); p = p.next() ) ... ;
```

Performance of linear list

We assume a linear list with one end pointer. Here pointer refers to "logical pointer", meaning any information that allows for obtaining the one-pass-the-end location in constant time. Note in the tables *random access* refers to accessing through index.

Operation	Performance	Note
Insert at front	Super fast	Neglecting stretching
Deletion at front	Super fast+	Update `size`
Insert at end	Reasonable	Move all elements
Deletion at end	Reasonable	Move all elements
Insert at middle	Reasonable	Move following elements
Deletion at middle	Reasonable	Move following elements
Random access	Super fast	Arrays are stored continuously
Forward iteration	Easy	iterate by index
Backward iteration	Easy	iterate by index
Memory consumption	Least	(Arguably, unused space)

Performance of Singly Linked list

We assume a singly linked list with both front and end pointer.

Operation	Performance	Note
Insert at front	Super fast	allocate a node
Deletion at front	Super fast	delete a node
Insert at end	Super fast	allocate a node
Deletion at end	Reasonable*	Why?
Insert at middle	Super fast	play around pointers
Insert at middle	Super fast	play around pointers
Random access	Reasonable*	Need to follow the links
Forward iteration	Easy	Use the link, Luke.
Backward iteration	Slow	Why?
Memory consumption	A little	1 extra ptrs / elem

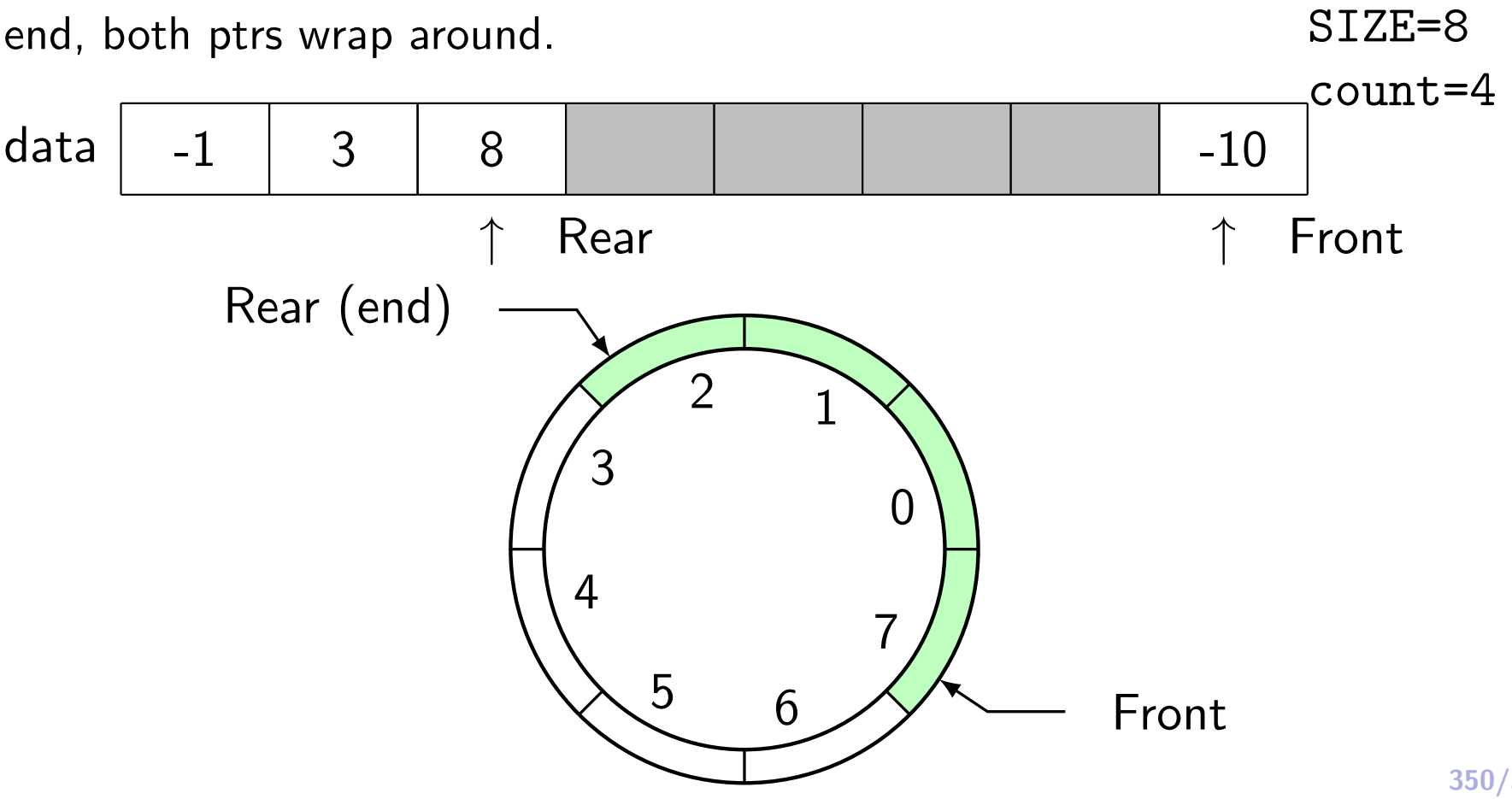
Performance of Doubly Linked list

We assume a doubly linked list with both front and end pointer.

Operation	Performance	Note
Insert at front	Super fast	allocate a node
Deletion at front	Super fast	delete a node
Insert at end	Super fast	allocate a node
Deletion at end	Super fast	Compare with previous
Insert at middle	Super fast	play around pointers
Insert at middle	Super fast	play around pointers
Random access	Reasonable*	Could be improved.
Forward iteration	Easy	Use the link, Luke.
Backward iteration	Easy	Thanks to the reverse link.
Memory consumption	A little*	2 extra ptrs / elem

Implementation of Circular Array

A circular array is sequential data structure. It is based on an array. It uses both front and rear ptrs to indicate begin and end. When a new element is added into the array, front moves to begin and rear moves to end, both ptrs wrap around.



Performance of Circular Array

We assume a circular array with **fixed** size and both front and rear pointer.

Operation	Performance	Note
Insert at front	Super fast	
Deletion at front	Super fast	Move front ptr
Insert at end	Super fast	
Deletion at end	Super fast	Move rear ptr
Insert at middle	Reasonable	How?
Deletion at middle	Reasonable	How?
Random access	Super fast	How?
Forward iteration	Easy	
Backward iteration	Easy	
Memory consumption	Least	Unused space

A note on the performance

RC Week 12

Standard Template Library

History of the standard template library

std::vector

std::list

std::map

std::unordered_map