

VE280 Recitation Class Notes

VE280 SU19 TA Group

UM-SJTU-JI

July 1, 2019

Source code available at <https://github.com/ve280/VE280-Notes>
Slides modified from the version of YAO Yue's, with permission

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?

Technique 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

Many of them are related to *Abstraction*.

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).

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 Procedural 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 TANKs in the game, or modify the how tanks fire by simply changing this single function.

Head First *Abstraction*

With just procedural 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.

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 manage your hardware. These include 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/dnf*.

Flavors of Linux: *Distributions*

Uncommon choices

Open SUSE

Maintained by german company SUSE. Focus on stability.

Arch Linux / Manjaro

ROLLING UPDATE! Absolutely cutting edge. Great community and wiki. User is assumed to be experienced. Installation of Arch Linux is done on command line while Manjaro makes easier.

Package management system is *pacman*.

Linux From Scratch (LFS) / Gentoo

Try LFS if you are absolutely interested in figuring out how everything works. Think LFS is crazy enough? Well Gentoo 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.

Virtual Machines vs. Containers

Virtual Machines

A virtual machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide functionality of a physical computer. The CPUs, memory, kernel, file systems and devices are all virtualized and need translation, which makes it slower than a physical computer.

Containers

A container shares the CPUs, memory, kernel, file systems and devices with the host machine, which makes it almost as fast as the host machine when executing machine code (no virtualization means no translation of machine code). Everything in a container is something mapped from the host system. [Docker](#) is a lightweight, standalone and secure container daemon which is very popular nowadays.

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.

Permission 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:

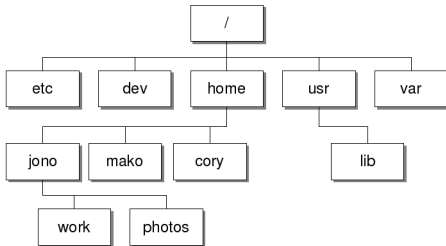
Aspects of Linux: Filesystem

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 **B**inaries

/dev **D**evelop

/boot **B**ootstrap

/sbin **S**ecure **b**inaries

/mnt **M**ounted filesystem

/etc **C**onfiguration files

/usr **U**nix **s**ystem **r**esources

/home **P**laces 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.

Different choices of shell

The “standard” one on most linux is `bash`, as in “**B**ourne’s **A**gain **S**hell”. But choices like `zsh`, `fish`, `cs` 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 `test.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 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 **long** format. Each line for a single file.

mkdir **Make** **directory**, self-explanatory.

rm **Remove** 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 **Remove** **directory**, only **empty** ones can be removed.

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** **C**opy 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** **M**ove 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** **C**on**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

Long Format of File Information

```
root@liu-lenovo-M41-70:/# ls -l
```

permission	link	user	group	file size	modified time	file name
drwxr-xr-x	2	root	root	4096	Apr 21 06:13	bin
drwxr-xr-x	3	root	root	4096	Apr 21 06:13	boot
drwxr-xr-x	2	root	root	4096	May 14 2018	cdrom
drwxr-xr-x	20	root	root	4060	May 22 14:31	dev
drwxr-xr-x	155	root	root	12288	May 14 06:21	etc
drwxr-xr-x	3	root	root	4096	May 14 2018	home
lrwxrwxrwx	1	root	root	33	Apr 5 06:29	initrd.img -> boot/initrd.img-4.15.0-47-generic
lrwxrwxrwx	1	root	root	33	Apr 5 06:29	initrd.img.old -> boot/initrd.img-4.15.0-46-generic
drwxr-xr-x	22	root	root	4096	Jun 30 2018	lib
drwxr-xr-x	2	root	root	4096	Jun 25 2018	lib32
drwxr-xr-x	2	root	root	4096	Apr 27 2018	lib64
drwxr-xr-x	2	root	root	4096	Jun 25 2018	libx32
drwx-----	2	root	root	16384	May 14 2018	lost+found
drwxr-xr-x	3	root	root	4096	May 15 2018	media
drwxr-xr-x	2	root	root	4096	Apr 27 2018	mnt
drwxr-xr-x	10	root	root	4096	Dec 4 20:45	opt
dr-xr-xr-x	247	root	root	0	May 22 14:30	proc
drwx-----	9	root	root	4096	Nov 22 15:06	root
drwxr-xr-x	32	root	root	940	May 22 14:33	run
drwxr-xr-x	2	root	root	12288	May 14 06:20	sbin
drwxr-xr-x	2	root	root	4096	May 14 2018	snap
drwxr-xr-x	3	root	root	4096	Aug 14 2018	srv
-rw-----	1	root	root	2147483648	May 14 2018	swapfile
dr-xr-xr-x	13	root	root	0	May 22 14:41	sys
drwxrwxrwt	18	root	root	4096	May 22 14:41	tmp
drwxr-xr-x	13	root	root	4096	Oct 3 2018	usr
drwxr-xr-x	14	root	root	4096	Apr 27 2018	var
lrwxrwxrwx	1	root	root	30	Apr 5 06:29	vmlinuz -> boot/vmlinuz-4.15.0-47-generic
lrwxrwxrwx	1	root	root	30	Apr 5 06:29	vmlinuz.old -> boot/vmlinuz-4.15.0-46-generic

Aspects of Linux: Shell

Long Format of File Information, Cont'd

File permission

- first character: '-' regular file; 'd' directory
- read, write, execution permission of the owner
- read, write, execution permission of the group
- read, write, execution permission of everyone else
- last character: '-' regular file; 'x' executable

Link

- the number of hard links of the file
- a regular file has only one hard link (of itself)
- a directory has $2 + n$ hard links where n is the number of its subdirectories because there are two extra links to `.` and `..`

Owner/Group File Size

- act with file permissions
- in bytes

Aspects of Linux: Shell

CLI Utilities

nano Command line file editor.

diff Compare the **difference** 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 **vi improved**, 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 output.

`prog1 | prog2` Called a “pipe”. Connects the stdout of `prog1` to stdin of `prog2`

Aspects of Linux: Shell

Globbering

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

```
xmxxmxd↔xmxxmxdtql↔
xmxxmzdnxd↔xmxxmd↔
xmxxmxdtql↔xmxxmzdnxd↔
xmxxmd↔xmxxmxdtql↔
xmxxmzdnxd↔xmxxmxd↔
xmxxmxdtql↔xmxxmzdnxd↔
xmxxmd↔xmxxmxdtql↔
xmxxmzdnxd↔xmxxmd↔
xmxxmxdtql↔xmxxmzdnxd↔
```

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?

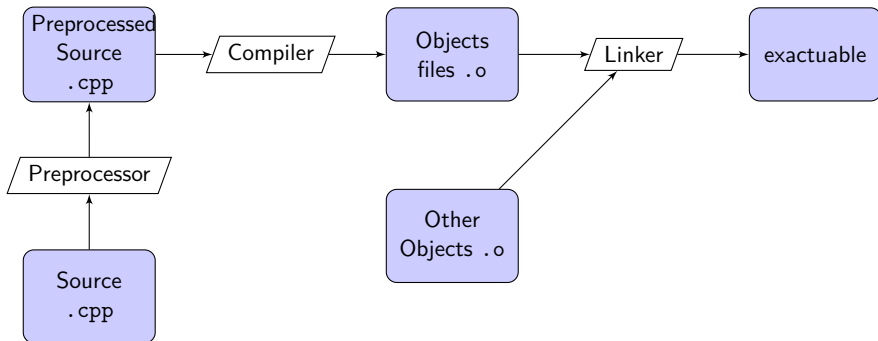
- 37,626 – The number of files
- 15,004,006 – The number of lines of code

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 weird 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/

Search: Go

Not logged in [register](#) [log in](#)

C++

Information
Tutorials
Reference
Articles
Forum

Reference

- C library:
- Containers:
- Input/Output:
- Multi-threading:
- Other:
 - <algorithm>
 - <bitset>
 - <chrono>
 - <codecvt>
 - <complex>
 - <exception>

function

std::to_string C++11 <string>

```
string to_string (int val);
string to_string (long val);
string to_string (long long val);
string to_string (unsigned val);
string to_string (unsigned long val);
string to_string (unsigned long long val);
string to_string (float val);
string to_string (double val);
string to_string (long double val);
```

Convert numerical value to string

Returns a string with the representation of *val*.

The format used is the same that `printf` would print for the corresponding type:

type of <i>val</i>	<code>printf</code> equivalent	description
int	<code>"%d"</code>	Decimal-base representation of <i>val</i> .

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.40. `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 its *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 introduces its 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

Syntactically 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.

	0	4	8	12
0x90	Random data...			
0xA0	x[0]	x[1]	x[2]	x[3]
0xB0	x[4]	x[5]	Unknown	
0xC0	Random data...			

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;
};
```

	0	4	8	12
0xA0	x	y	z	-
0xB0	l1		ptr	t[0]
0xC0	t[1]	-	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 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 prefer?

- 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 Programming Language*

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

Declaration

Interpretation

- | | |
|------------------------------|--|
| ■ <code>int a;</code> | ■ <code>a</code> is an <code>int</code> |
| ■ <code>int arr[4];</code> | ■ <code>arr[]</code> → <code>int</code> , <code>arr</code> → array of <code>int</code> |
| ■ <code>int *p;</code> | ■ <code>*p</code> → <code>int</code> , <code>p</code> → pointer to <code>int</code> |
| ■ <code>int *(pa[4]);</code> | ■ <code>*(pa[4])</code> → <code>int</code> , <code>pa[4]</code> → pointer to <code>int</code> , <code>pa</code> → array of pointer to <code>int</code> . |
| ■ <code>int (*pp)[4];</code> | ■ <code>*pp</code> → array of <code>int</code> , <code>pp</code> → pointer to array of <code>int</code> . |

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

int q = 10; int& rq = q;             ^^I
const int& crq = q;                 bar(q);    bar(*pq); // OK
int* pq = &q;                       bar(10);    // ERROR
const int* cpq = &q;                bar(*cpq); // ERROR
                                     bar(crq)    // ERROR

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

```

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 propagation

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 error  
}
```

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 referring 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 sometimes 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.

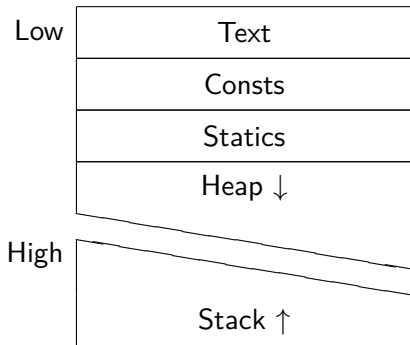
RC Week 5

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

Explanation



* This graph is up-side-down.

- “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. Arguments, return values, return addresses ...

Element in a stack

foo	<code>z = 4 @ foo(3)</code>
	<code>x = 3 @ foo(3)</code>
	<code>Ret = ?</code>
	<code>RA = &CALLS + 1</code>
	<code>q = ? @ main()</code>
main	<code>p = 3 @ main()</code>
	<code>...</code>

```

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 FFFUUUNNNN!

Understanding 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 inputted Hello, symptoms are:

- Program crashes after printing 4 times of Hello.
- Program runs fine 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 keep 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

	x = 225 @ foo(225)
	Ret = 225
foo	RA = &foo(15)
	x = 225 @ foo(15)
	Ret = ?
foo	RA = &main
main	...

Without optimization

	x = 225 @ foo(225)
	Ret = 225
foo	RA = &main
main	...

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

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

- The loop out-of-bound.
- How 2D arrays are stored.
- Understand how indexing works.
- We omit the arguments and return values of foo on the graph

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 *DESTROYs* the local objects, essentially calling their destructors.

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 << "Opened : " << 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:

```
Opened : 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 case 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 partition-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 $(a_0, a_1, a_2, \dots, a_n)$ and returns $\sum_{i=0}^n i^2 a_i$ (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

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, by 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 delicately constructed to sabotage / 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!

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

QA engineer walks into a bar

- 140 / 203

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 *modules*.
- 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 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

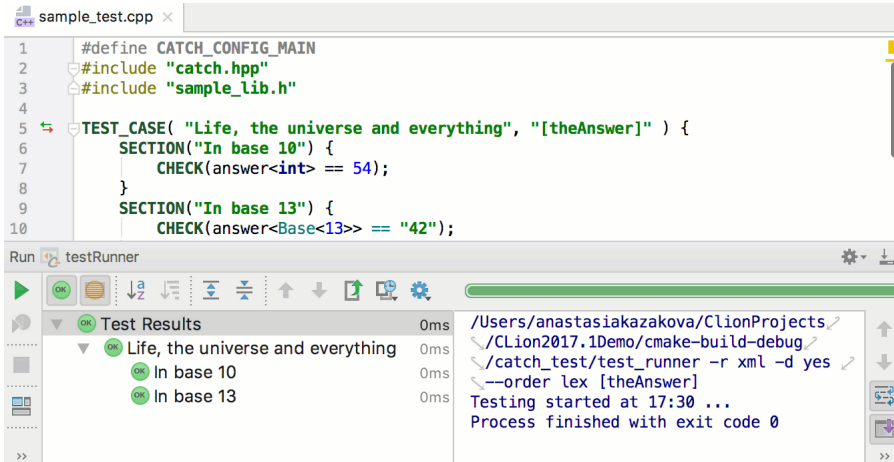
Project Homepage: <https://github.com/google/googletest>

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



The screenshot shows the CLion IDE interface. The top editor window displays the file `sample_test.cpp` with the following C++ code:

```
1  #define CATCH_CONFIG_MAIN
2  #include "catch.hpp"
3  #include "sample_lib.h"
4
5  TEST_CASE( "Life, the universe and everything", "[theAnswer]" ) {
6      SECTION("In base 10") {
7          CHECK(answer<int> == 54);
8      }
9      SECTION("In base 13") {
10         CHECK(answer<Base<13>> == "42");
11     }
```

Below the editor is the **Run** toolbar and the **testRunner** output window. The test results are as follows:

Test Case	Duration
Test Results	0ms
Life, the universe and everything	0ms
In base 10	0ms
In base 13	0ms

The test runner output window shows the following command and results:

```
/Users/anastasiakazakova/CLionProjects
/CLion2017.1Demo/cmake-build-debug
/catch_test/test_runner -r xml -d yes
--order lex [theAnswer]
Testing started at 17:30 ...
Process finished with exit code 0
```

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 process. This is called the fail-fast fast.

- The program is already in a 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 memory  
    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 in a 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 `tan` 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 now 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 cannot be overlooked! Unhandled 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 “Terminate”. There is a 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 series of 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 know 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 IntegerException : public Exception {};  
class FileException    : public Exception {};  
class FileNotFoundException : public FileException {};  
class PermissionDenied : public FileException {};  
void doSomethingToFile(string filename);  
void foo(int n) {  
    string file = "str" + to_string(n);  
    try { doSomethingToFile(file) }  
    catch (const FileException& e) {  
        ^^Icout << "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

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.
- `cerr` 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::string`.

```
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 `inputFileStream.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 destructor 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 ostream 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:

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

And you need to turn such structures into below format

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

You can setup an ostream. You simply push into the stream as if it is cout, finally you call ostream.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;  
};  
Natural num;
```

- class key word begins the specification of an abstract datatype. CamelCase style recommend you to capitalize first character.
- public key words begins specifying the *operations*, the *abstraction*, 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__
#define __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 default** 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 longer 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 a really complicated calculation with `i` and `c`.
- You have no choice but to mark `Bar::doComp()` as `const`. This forces you to 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 end 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;
}
```

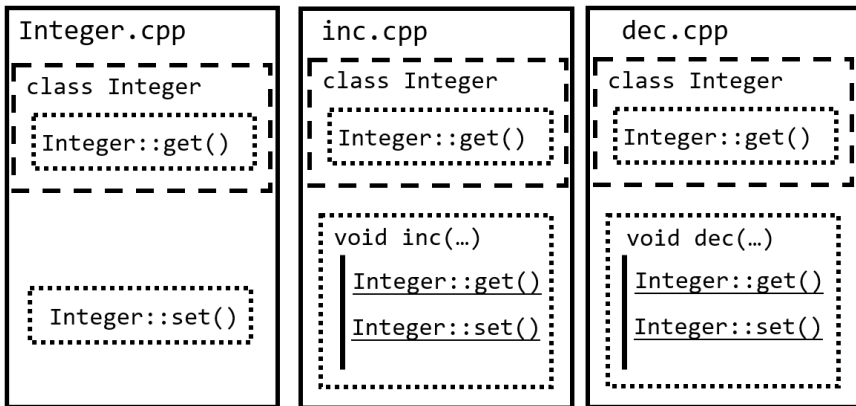
Compile the code and execute it:

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

We observe the (not surprising) result 10 11 10

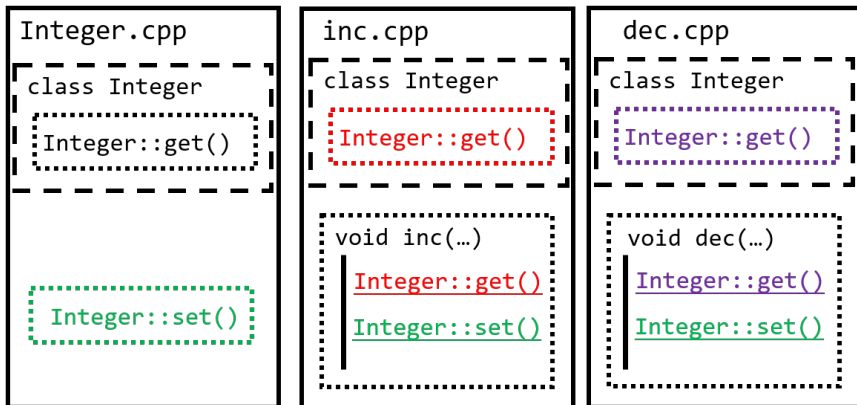
On linking classes

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 indexOf(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::indexOf(int v) const {
3      // Omit implementing
4      // Not found return MAXELTS
5  }
6  void IntSet::insert(int v) {
7      if (indexOf(v) != MAXELTS) {
8          assert(numElts < 100);
9          elts[numElts++] = v;
10     }
11 }
12 void IntSet::remove(int v) {
13     int vic = indexOf(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 `numElt`s 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) {}  
    // Initialize 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)); // Anonymous 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 a 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 go 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 make 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
```

We omit the very similar version for EvenIntSet.

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";
}
```

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 output:

```
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::IntSet() = 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. The 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 observe 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.