COMP281
Principles of C and memory management

Lecture 3

Dr. Frans Oliehoek
Department of Computer Science,
University of Liverpool.

Last week

- Got acquainted with C
  - fast, efficient language…
  - …which is not going to stop you from making horrible mistakes!
- First look at C Memory organization
  - static data, call stack
- A variety of language elements
  - input/output, functions, variables, standard headers, etc.

Overview: today

- More about the call stack and *how* functions are called
- The lab: using 'online judge'
    - How it works
    - First approaches to debugging...
    - Submitting

Function Calls & The Call Stack

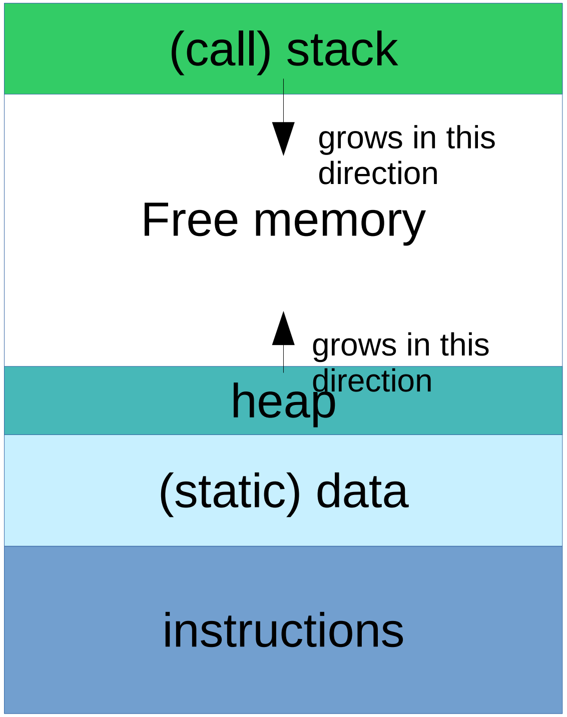
Refresher: The Call Stack

- Remember: the call stack stores
  - return address
  - local variables
  - (and some more stuff...)

  for function calls

- Let's look at this in more detail...

- First: some terminology!



Functions: arguments, parameters

- **arguments** are passed in, **parameters** are received

```
int main()
{
    int v1=42, v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}

int add(int a, int b)
{
    int res = a+b;
    return res;
}
```

Functions: arguments, parameters

- **arguments** are passed in, **parameters** are received

```
int main()
{
    int v1=42, v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d", result);
    return(0);
}
```

v1, v2 are...

- local variables stored in `main`'s stack frame
- the **arguments** that are passed to the `add` function

```
int add(int a, int b)
{
    int res = a+b;
    return res;
}
```

a,b are...

- the **parameters** that are received by `add`
- they are stored in the frame of `add` → have a different location in memory
- initialized with a **copy** of v1, v2

Functions: arguments, parameters

- **arguments** are passed in, **parameters** are received

```
int main()
{
    int v1=42, v2=3;
    int result;

    result = add(v1, v2);
    printf("result: %d", result);
    return(0);
}


int add(int a, int b)
{
    int res = a+b;
    return res;
}
```

v1, v2 are...

- local variables stored in `main`'s stack frame
- the **arguments** that are passed to the `add` function

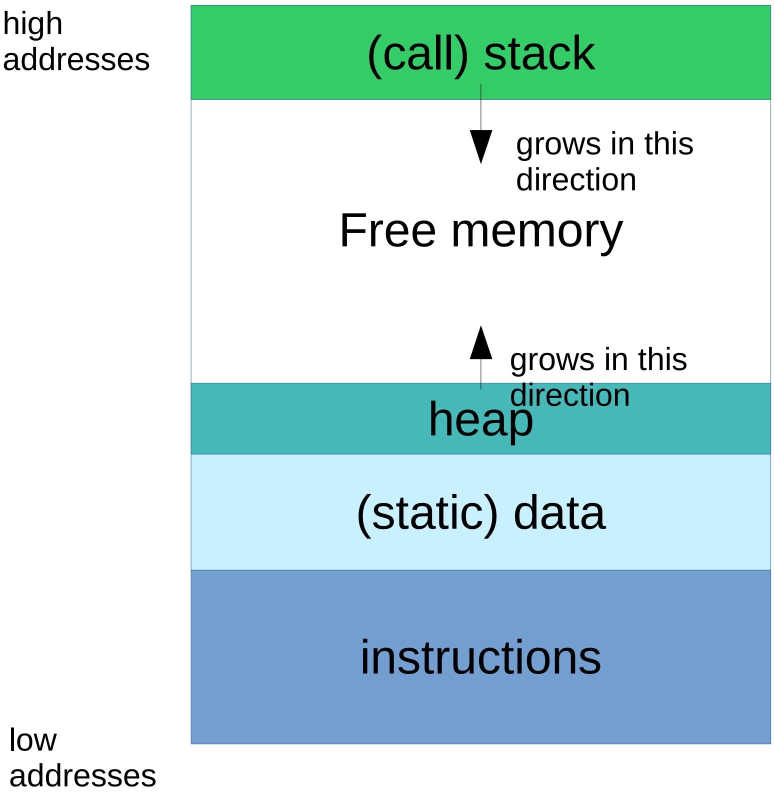
It's simple: 'a' comes before 'p'...

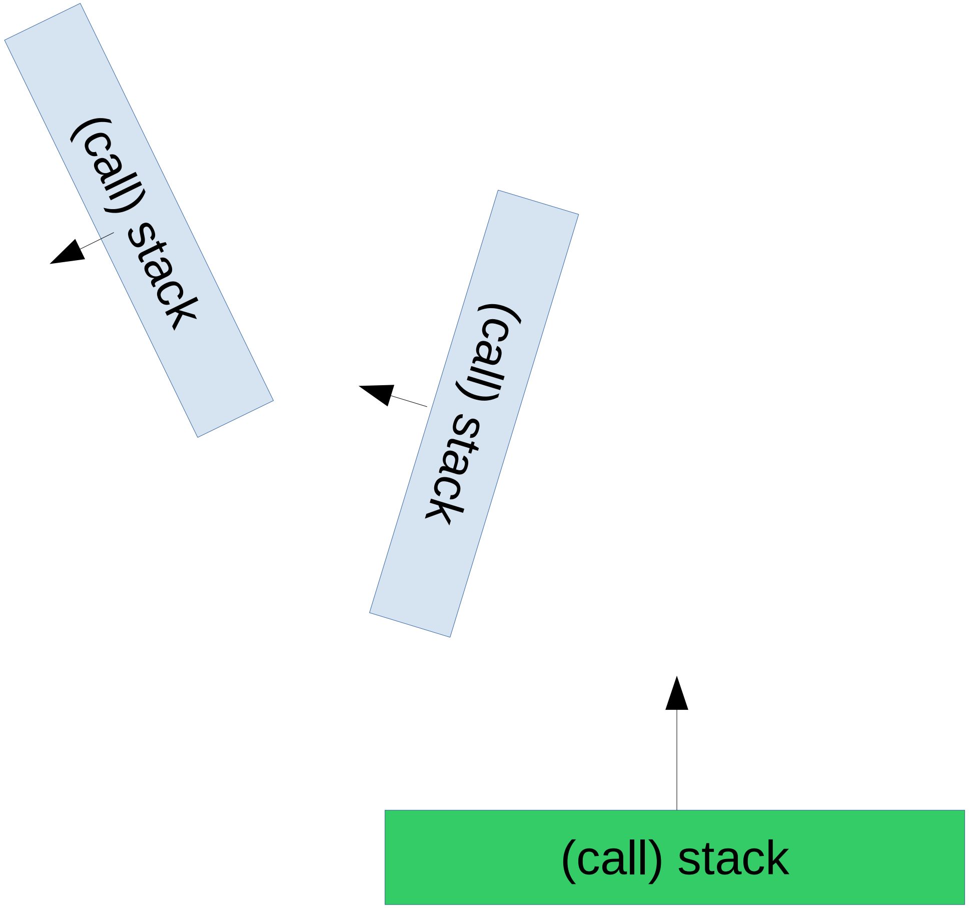
a,b are...

- the **parameters** that are received by `add`
- they are stored in the frame of `add` → have a different location in memory
- initialized with a **copy** of v1, v2

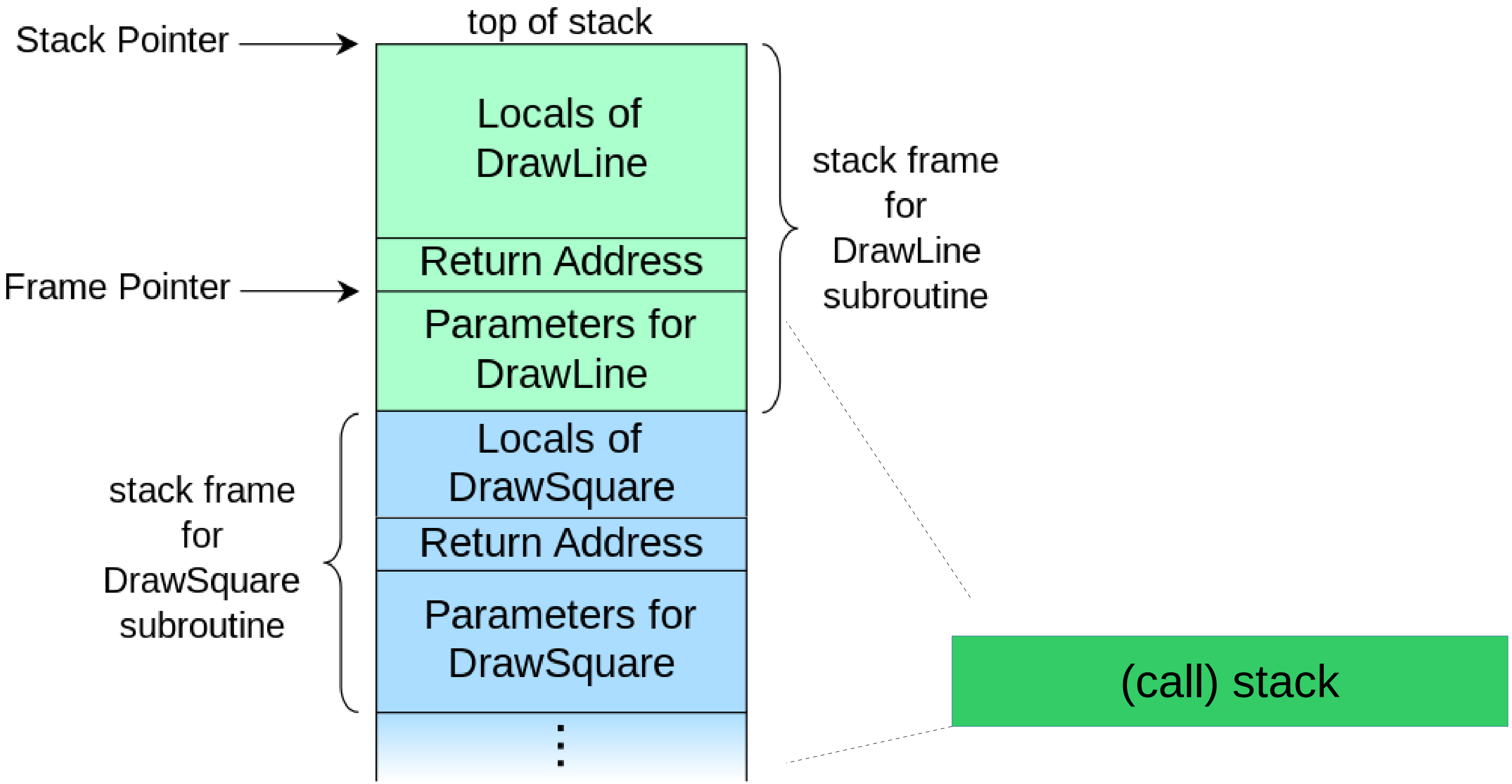
So how does a function call work?

- Note: We will interpret the stack as a regular stack which it top pointing upwards:





Stack contains a 'frame' for each function



Example

- Same example

```
int main()
{
      int v1=42, v2=3;
      int result;

      result = add(v1, v2);
      printf("result = %d",
          result);
      return(0);
}

int add(int a, int b)
{
      int res;
      res = a+b;
      return res;
}
```

current position; right before call to add

frame	symbol	address	value
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

- Same example

```
int main()
{
      int v1=42, v2=3;
      int result;

      result = add(v1, v2);
      printf("result = %d",
            result);
      return(0);
}

int add(int a, int b)
{
      int res;
      res = a+b;
      return res;
}
```

current position; right before call to add

hypothetical memory addresses

frame	symbol	address	value
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

- Same example

```
01. int main()
02. {
03.     int v1=42, v2=3;
04.     int result;
05.
06.     result = add(v1, v2);
07.     printf("result = %d",
08.         result);
09.     return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.     int res;
15.     res = a+b;
16.     return res;
17. }
```

'add' is called: a frame is added to the stack

frame	symbol	address	value
add	res	126	<garbage>
	return addr.	122	line 06
	return value	118	<garbage>
	b	116	3
	a	112	42
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

"Call by value"

- Same example

```
01. int main()
02. {
03.     int v1=42, v2=3;
04.     int result;
05.
06.     result = add(v1, v2);
07.     printf("result = %d",
08.         result);
09.     return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.     int res;
15.     res = a+b;
16.     return res;
17. }
```

Note:

- arguments and params are different: own address!
- Upon function call, the **values** of the arguments is **copied** into the params.
  - Q: How do you think...?

'add' is called: a frame is added to the stack

frame	symbol	address	value
add	res return addr. return value b a	126 122 118 116 112	<garbage> line 06 <garbage> 3 42
main	result v2 v1	108 104 100	<garbage> 3 42

Example

Note:

- arguments and params are different: own address!
- Upon function call, the **values** of the arguments is **copied** into the params.
  - Q: How do you think...?
  - A: 'pushed' on the stack!

'add' is called: a frame is added to the stack

- Same example

```
01. int main()
02. {
03.       int v1=42, v2=3;
04.       int result;
05.
06.       result = add(v1, v2);
07.       printf("result = %d",
08.             result);
09.       return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.       int res;
15.       res = a+b;
16.       return res;
17. }
```

frame	symbol	address	value
add	res return addr. return value b a	126 122 118 116 112	<garbage> line 06 <garbage> 3 42
main	result v2 v1	108 104 100	<garbage> 3 42

Example

- Same example

```
01. int main()
02. {
03.       int v1=42, v2=3;
04.       int result;
05.
06.       result = add(v1, v2);
07.       printf("result = %d",
08.             result);
09.       return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.       int res;
15.       res = a+b;
16.       return res;
17. }
```

'res' is computed

frame	symbol	address	value
add	res	126	45
	return addr.	122	line 06
	return value	118	<garbage>
	b	116	3
	a	112	42
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

function return is started...

- Same example

```
01. int main()
02. {
03.     int v1=42, v2=3;
04.     int result;
05.
06.     result = add(v1, v2);
07.     printf("result = %d",
08.         result);
09.     return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.     int res;
15.     res = a+b;
16.     return res;
17. }
```

frame	symbol	address	value
add	res	126	45
	return addr.	122	line 06
	return value	118	<garbage>
	b	116	3
	a	112	42
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

function return is started...
- return value is stored

- Same example

```
01. int main()
02. {
03.       int v1=42, v2=3;
04.       int result;
05.
06.       result = add(v1, v2);
07.       printf("result = %d",
08.           result);
09.       return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.       int res;
15.       res = a+b;
16.       return res;
17. }
```

frame	symbol	address	value
add	res	126	45
	return addr.	122	line 06
	return value	118	45
	b	116	3
	a	112	42
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

- Same example

```
01. int main()
02. {
03.       int v1=42, v2=3;
04.       int result;
05.
06.       result = add(v1, v2);
07.       printf("result = %d",
08.             result);
09.       return(0);
10. }
11.
12. int add(int a, int b)
13. {
14.       int res;
15.       res = a+b;
16.       return res;
17. }
```

function return is started...

- return value is stored
- local variables are popped

frame	symbol	address	value
add	return addr.	122	line 06
	return value	118	45
	b	116	3
	a	112	42
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

- Same example

```
int main()
{
    int v1=42, v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d",
        result);
    return(0);
}

int add(int a, int b)
{
    int res;
    res = a+b;
    return res;
}
```

function return is started...

- return value is stored
- local variables are popped
- return address restored and popped

frame	symbol	address	value
add	return value	118	45
	b	116	3
	a	112	42
main	result	108	<garbage>
	v2	104	3
	v1	100	42

Example

- Same example

```
int main()
{
    int v1=42, v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d",
        result);
    return(0);
}

int add(int a, int b)
{
    int res;
    res = a+b;
    return res;
}
```

function return is started...

- return value is stored
- local variables are popped
- return address restored and popped
- return value used to complete instruction, and then popped

frame	symbol	address	value
add	b	116	3
	a	112	42
main	result	108	45
	v2	104	3
	v1	100	42

Example

- Same example

```
int main()
{
    int v1=42, v2=3;
    int result;

    result = add(v1, v2);
    printf("result = %d",
        result);
    return(0);
}

int add(int a, int b)
{
    int res;
    res = a+b;
    return res;
}
```

function return is started...

- return value is stored
- local variables are popped
- return address restored and popped
- return value used to complete instruction, and then popped
- parameters popped; frame is destroyed

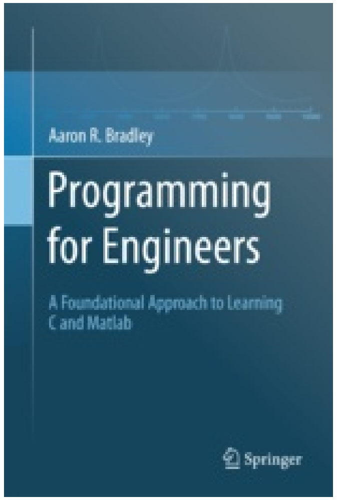
frame	symbol	address	value
main	result	108	45
	v2	104	3
	v1	100	42

The Stack – Summary

- Stack is a mechanism for **dynamic memory allocation**
  - it allocates memory as needed by functions
  - very powerful in combination with recursive functions

- The function call protocol and return protocol take care of all these things; need not to worry mostly
  - and it's fast.

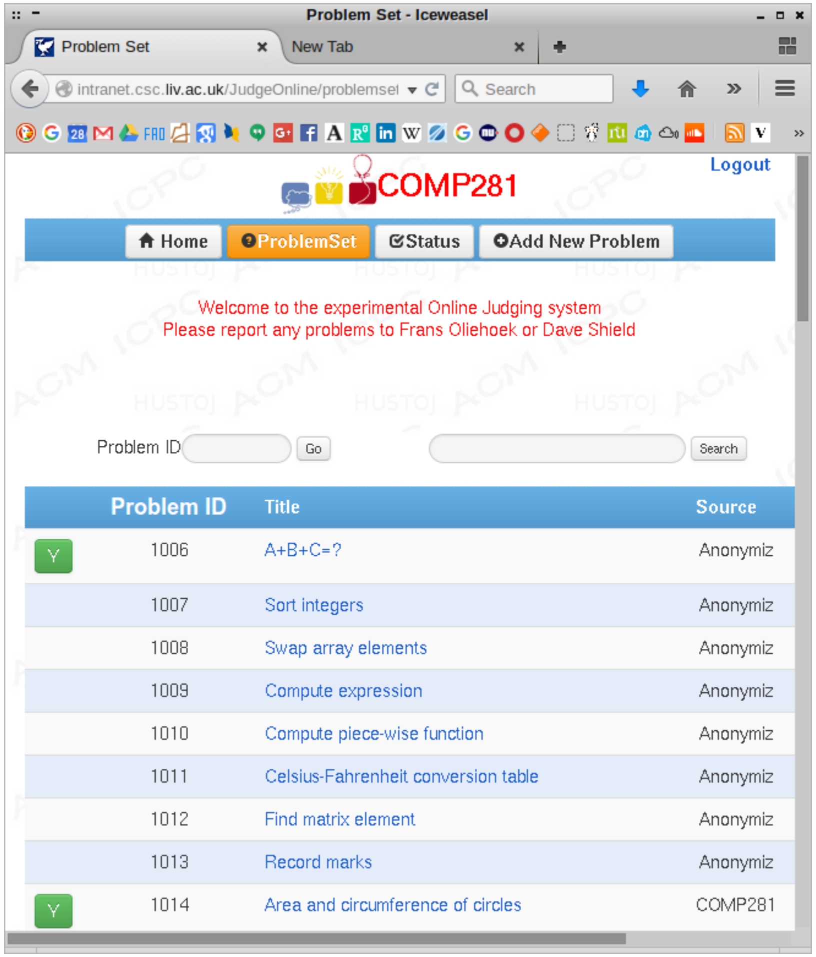
- However, remember:
  Arguments are **copied** into functions by pushing them onto the stack in the callee's frame
  - i.e. C by default implements **call by value**


Aaron R. Bradley
Programming for Engineers
A Foundational Approach to Learning C and Matlab
Springer

Online Judge & Submitting Assignments

Online Judge

- The online judge practice system…
  - http://intranet.csc.liv.ac.uk/JudgeOnline/
  - practice programming exercises
  - submit your assignments here
- It will perform 'black box' tests on the program
  - test input to expected output
- Make sure you read the assignments carefully!



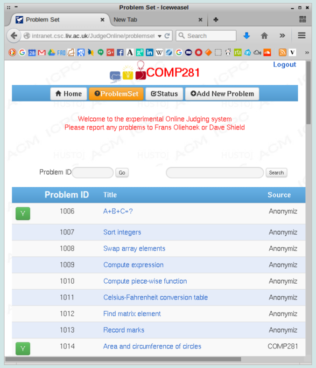
Welcome to the experimental Online Judging system
Please report any problems to Frans Oliehoek or Dave Shield

	Problem ID	Title	Source
Y	1006	A+B+C=?	Anonymiz
	1007	Sort integers	Anonymiz
	1008	Swap array elements	Anonymiz
	1009	Compute expression	Anonymiz
	1010	Compute piece-wise function	Anonymiz
	1011	Celsius-Fahrenheit conversion table	Anonymiz
	1012	Find matrix element	Anonymiz
	1013	Record marks	Anonymiz
Y	1014	Area and circumference of circles	COMP281

Black box testing: How OJ works

Step 1: Compiling your source

your code



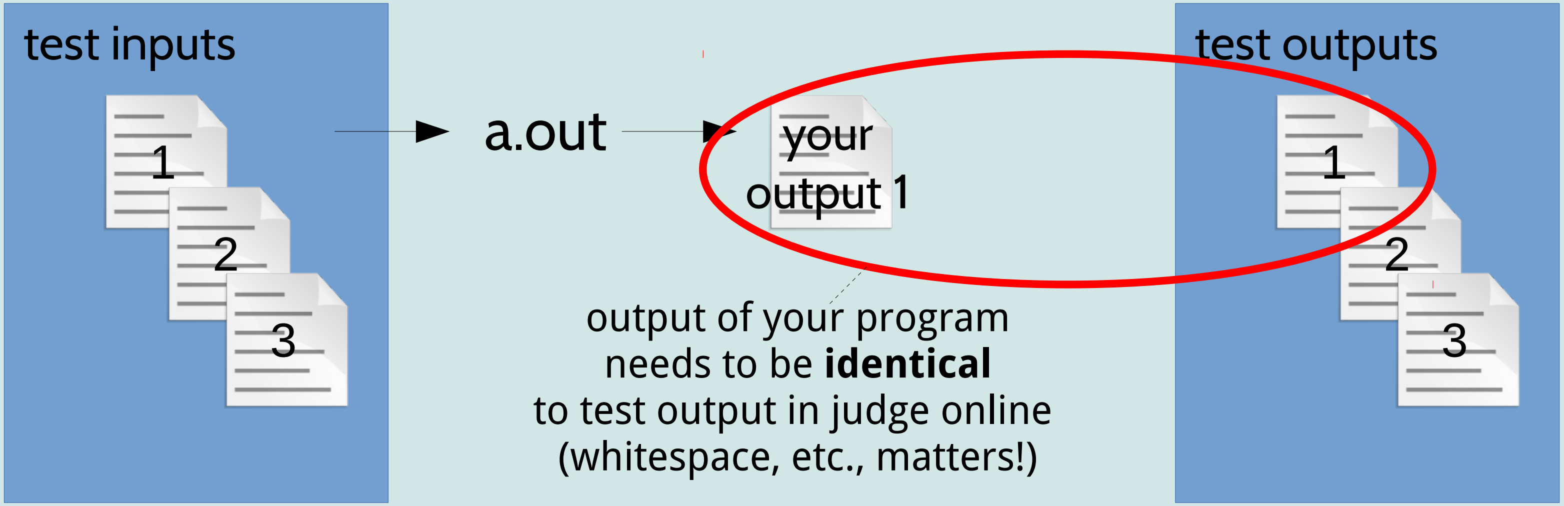
Welcome to the experimental Online Judging system
Please report any problems to Frans Oliehoek or Dave Shield

Problem ID

	Problem ID	Title	Source
Y	1006	A+B+C=?	Anonymiz
	1007	Sort integers	Anonymiz
	1008	Swap array elements	Anonymiz
	1009	Compute expression	Anonymiz
	1010	Compute piece-wise function	Anonymiz
	1011	Celsius-Fahrenheit conversion table	Anonymiz
	1012	Find matrix element	Anonymiz
	1013	Record marks	Anonymiz
Y	1014	Area and circumference of circles	COMP281

→ GCC → a.out

Step 2: Testing your program



test inputs

1

2

3

→ a.out → your output 1

test outputs

1

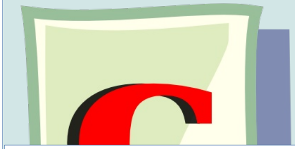
2

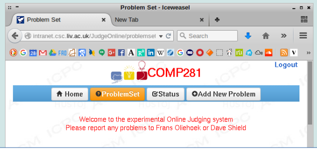
3

output of your program
needs to be **identical**
to test output in judge online
(whitespace, etc., matters!)

Black box testing: How OJ works

Step 1: Compiling your source





Welcome to the experimental Online Judging system
Please report any problems to Frans Oliehoek or Dave Shield

CC → a.out

Reproducing OJ

You can easily reproduce these steps:

1) prepare `test.in`: a text file with input in the format specified in assignment
2) compile:

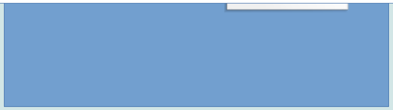
```
$ gcc YOURSOURCE.c
```

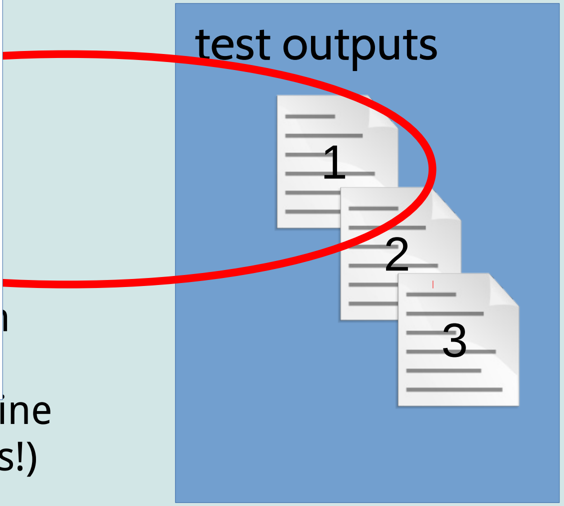
3) run your code 'piping' the input into it:

```
$ a.out < test.in > test.out
```

This generates the output in the file `test.out`. Inspect it to see if there is any unexpected elements in your output.

am





n

to test output in judge online
(whitespace, etc., matters!)

GCC (Gnu Compiler Collection)

- OJ uses the Gnu Compiler
  - http://gcc.gnu.org/
  - has become a standard for many systems

- Your assignments will be tested using the gcc compiler
  - You can access the gcc compiler using either the Linux or Mac systems
  - If you use Windows you can use gcc…
    - easiest: virtualbox – www.virtualbox.org
    - Cygwin or mingw (see https://gcc.gnu.org/install/binaries.html)

- Alternative:
    - Visual Studio mostly uses the ANSI standard; it should work…
    - …but might “works on your pc” but not on OJ…!

- See also the additional handout on 'using different compilers'

Compiling

- With Visual Studio / Xcode / etc.
  - Click the correct buttons.....

- Linux (and Mac Terminal, Cygwin, ...):

```
gcc -Wall helloworld.c [-o executable filename]
```

  - "-Wall" enables all warnings... this is **useful**!!
  - if you don't specify an output file, you get one called a.out (or possible a.exe on windows/dos)
  - To run the resulting program

```
./a.out
```

Compiling Locally

- You could rely on OJ for compilation…
  …but it is **not** convenient.
  - It is convenient to have a working linux system to do compilation
  - Moreover, it is **required** in order to use some of the programming tools we will cover later…
- Easy way into to linux:
  - e.g., "ubuntu linux"
  - run in a virtual machine (e.g., https://www.virtualbox.org/)

A Variant of Problem 1006

Title	**A+B=?**
Description	Calculate a+b
Input	Two integer a,b (0<=a<=10, 0<=b<=10)
Output	Output a+b
Sample Input	1 2
Sample Output	3

Solution

```
#include <stdio.h>
int main()
{
    // declare variables before using them
    // (required by C89 standard)
    int a,b,answer;

    // read a decimal number as input and store it in a
    scanf("%d",&a);

    // read another decimal number and store it in b
    scanf("%d",&b);

    // compute the solution
    answer = a + b;

    // print out the solution as a decimal number
    printf("%d",answer);
}
```

Solution

```
#include <stdio.h>
int main()
{
    // declare variables before using them
    // (required by C89 standard)
    int a,b,answer;

    // read a decimal number as input and store it in a
    scanf("%d",&a);

    // read another decimal number and store it in b
    scanf("%d",&b);

    // compute the solution
    answer = a + b;

    // print out the solution as a decimal number
    printf("%d",answer);
}
```

alright, so let's try it on online judge (OJ)...

1006-variant.c
1006-variant-fixed.c

Online Judge Says: “No”

- Before you start debugging… **Read the notifications!**
  - There may be useful information there!
  - e.g., is it a 'run-time' error or 'compilation' error?

- Things to do:
  - Read the problem again
    - Are you sure you have interpreted it correctly?

  - Test it yourself: It’s very likely (99%) that the fault is with your program…!
    - Did you try different test cases?
    - What **assumptions** did you make about the test cases?
    - Don't assume the test case(s) give you all of the possibilities
    - Did you try…. large numbers, small numbers, negative numbers, …, etc.?

  - Check the output
    - Does the output match ***exactly***?
    - Whitespace matters!
    - Don’t print anything else to the screen!

Online Judge Says: “Compilation Error”

- “...and it works on my computer”
- That is great, but not sufficient!
  - remember: compilers are different and OJ uses gcc

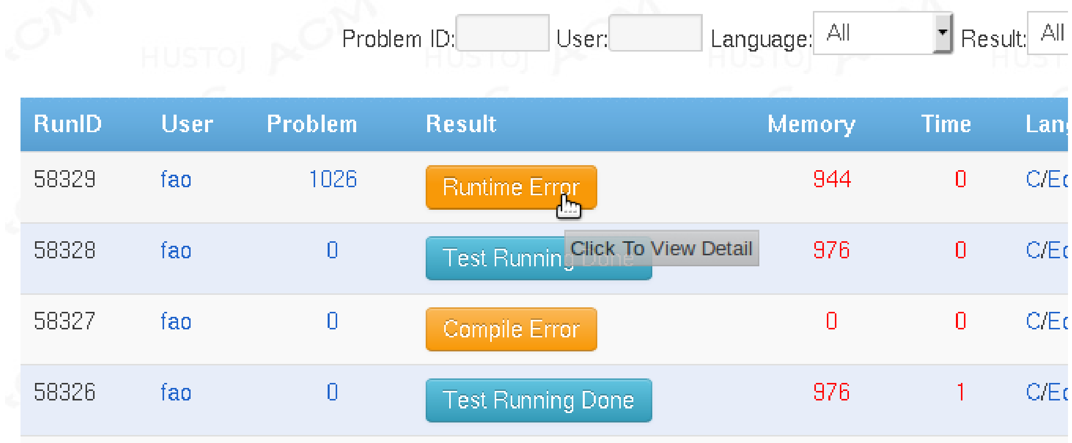
- But compilation errors are easy to fix: compiler tells you what goes wrong...

  ... so try using gcc!
  - do you have linux running?
  - try installing it, or use 'virtualbox'

- Also: **check the output box in OJ**
  - it gives you the error message!

Online Judge Says: “Runtime Error”

- Problem: this typically is a segmentation fault…
  - Click on the button to verify



Problem ID: User: Language: All Result: All

RunID	User	Problem	Result	Memory	Time	Lan
58329	fao	1026	Runtime Error	944	0	C/E
58328	fao	0	Test Running Done	976	0	C/E
58327	fao	0	Compile Error	0	0	C/E
58326	fao	0	Test Running Done	976	1	C/E

Click To View Detail

- Unfortunately, more difficult to resolve…
- …approaches:
  - **Testing:** try different inputs until you can reproduce the error
  - **Print debugging**: use print statements to understand where the problem occurs
  - **Use a debugger tool**: use the debugger gdb (treated next lecture) to find out what causes the error
- If that does not work… resolve at the lab!

OJ: Key Points

- You are free to submit **any** of the problems
  - as often as you like!
  - it's all there to practice, so go for it!
- OJ shows
  - runtime
  - memory usage
  - keep you programs as efficient as possible!
  - (for the simpler programs you will not see these numbers vary)
- When you 'submit' OJ will generate a **RunID**
  - this is important when submitting to the departmental system!

'Final' submission

- In order to hand in your assignments:
  - submit via the **departmental submission system(DSS)**
  - http://www.csc.liv.ac.uk/cgi-bin/submit.pl
- So, yes: there are 2 places to 'submit':
  - OJ: submit whatever you want to experiment, as often as you like
  - DSS: only submit the problems as you want them assessed, once.
- and they are related...
  - need to include the RunID of one of your 'accepted' OJ submissions in the DSS submission.

    → make sure that these submissions are the same! ←

'Final' submission

- In order to hand in your assignments:
  - submit via the **departmental submission system(DSS)**
  - http://www.csc.liv.ac.uk/cgi-bin/submit.pl
- So, yes: there are 2 places to 'submit':
  - OJ: submit whatever you want to experiment, as often as you like
  - DSS: only submit the problems as you'd want them assessed, once.
- and they are related...
  - need to include the RunID of one of your 'accepted' OJ submissions in the DSS submission

All this is spelled out in the assignments so just follow the instructions carefully!

→ make sure that these submissions are the same! ←

Frequent Mistakes

- "OJ accepts it so it is correct"
  - you are expected to give a correct solution to the stated problem
  - (so could be wrong even if OJ accepts!)

- Not following instructions to hand in:
  - no name on report or on code
  - no pdf file (but .docx or whatever...)
  - submitting not as .zip (but as .rar, .7z, ...)
  - not use the standard (pkzip) zip file format

  → if we cannot open it, we cannot grade it ←

- Copying...
  - ok to help each other, but don't copy
  - the **Golden Rule:**

    if you didn't write some part entirely by yourself, then declare this in your report

Lab and Online Judge (OJ) – Summary

- Many exercises to learn from
- OJ performs black box testing
- Read the assignment very well before starting asking,
- read the discussion board before asking,
- and when asking, first post on the discussion board
  - help each other!

- Most convenient way of working:
  - local compilation using gcc
  - advice: try and get linux running (e.g., in virtualbox)

Review

- More about the call stack and *how* functions are called
  - 'by value' arguments pushed on the stack to create the parameters
- The lab and using online judge
  - where to look if OJ says no

- You now know
  - the difference between arguments and parameters (and why that matters)
  - how to replicate OJ's black box testing
  - that you should submit correctly – read carefully!

suggested reading for this week

stack:	Bradley Ch1 / Lu Ch2
debugging:	Bradley Ch4 / Lu Ch3
pointers:	K&R Ch5 / Bradley Ch1
pointers and arrays:	K&R Ch5 / Bradley Ch3