G52CPP
C++ Programming
Lecture 2

Dr Jason Atkin

E-Mail: jaa@cs.nott.ac.uk

The “Hello World” Program

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  printf("Hello world!\n");
  return 0;
}
```

C version

```cpp
#include <cstdio>

int main(int argc, char* argv[])
{
  printf("Hello world!\n");
  return 0;
}
```

C++ version

This lecture

- Assuming that you know some basics from C and Java… you need to know these:
- C++ data types
  - Sizes of types
  - C types
  - Two C++ new types
- Type casting
- Operators
- **Pointers**

Data types

Sizes of types…

- **The size of types (in bits/bytes) can vary in C/C++**
  - For different compilers/operating systems
  - In Java, sizes are standardised, across O/Ss
- **Some guarantees are given**:
  - **A minimum size (bits): char 8, short 16, long 32**
  - **Relative sizes: `char` ≤ `short` ≤ `int` ≤ `long`**
- An `int` changes size more than other types!
  - Used for speed (not portability), but VERY popular! (fast)
  - Uses the most efficient size for the platform
  - 16 bit operating systems usually use 16 bit `int`
  - 32 bit operating systems usually use 32 bit `int`
  - 64 bit operating systems usually use 64 bit `int`
- `sizeof()` **operator** exists to tell us the size (later lecture)

Basic Data Types - Summary

Type	Minimum size (bits)	Minimum range of values (Depends upon the size on your platform)
`char`	8	-128 to 127 (WARNING: Java char is 16 bit!)
`short`	16	-32768 to 32767
`long`	32	-2147483648 to 2147483647
`float`	Often 32	Single precision (implementation defined) e.g. 23 bit mantissa, 8 bit exponent
`double`	Often 64	Double precision (implementation defined) e.g. 52 bit mantissa, 11 bit exponent
`long double`	≥ double	Extended precision, implementation defined
`int`	≥ short	varies

`bool` type (C++ only, not C)

- **`bool`** : **`true`**/**`false`**
- Similar to java's boolean type
- Boolean expressions have results of type '**`bool`**' in C++
  - But type **`int`** in C – a difference
- **IMPORTANT: `bool`** and **`int`** can be converted **implicitly / automatically** to each other
  - i.e. C++ is backward compatible
  - **`true`** defined to be **`1`** when converted to **`int`**
  - **`false`** defined to be **`0`** when converted to **`int`**
  - **`0`** is defined to be **`false`**, non-zero as **`true`**

`int`s, `bool`s and `boolean`s

- **In both C and C++ any integer types (i.e. char, short, long, int) can be used in conditions**
  - In C++ the value is ***silently*** converted to a C++ `bool` type
- When using integer types:
  - true is equivalent to non-zero (or 1), false is equivalent to zero
- Example:

```
int x = 6;
while ( x )
{
   printf( "X is %d\n", x );
   x -= 2;
}
```

- In Java this would be an error : “x not boolean”
- In C/C++ this is valid (it means ‘`while( x != 0 )`’)

`wchar_t` type (C++ only, not C)

- **`wchar_t`** : wide character
  - Like a Java '**`char`**'
- ASCII limited to values 0 to 127 (7 bits)
  - Not enough characters for some languages
- **`wchar_t`** is designed to be big enough to hold a character of the : "largest character set supported by the implementation's locale"

  (Bjarne Stroustrup, The C++ Programming Language)

signed/unsigned values

- Signed/unsigned variants of integer types
  - **Unlike in Java where they are all signed**
  - Examples:

```
signed char sc;        unsigned short us;
signed long sl;        unsigned int ui;
```

  - Default is **`signed`**
    - If neither '**`signed`**' nor '**`unsigned`**' stated

Simple C-style casts

Converting between types

- **Data can be converted between types**
- **Sometimes done implicitly**
  - If compiler knows how to safely change the type
  - e.g. **`char`** to a **`short`**, **`short`** to a **`long`**, **`float`** to a **`double`**, **`int`** to a **`double`** (same rules as Java)
- **Sometimes it has to be done explicitly**
  - If conversion may lose data
  - e.g. **`long`** to a **`short`**, **`short`** to a **`char`**, **`double`** to a **`float`**, **`float`** to an **`int`** (same rules as Java)
  - Or compiler needs to confirm that it isn't an error:

    Warnings mean "Are you sure?"

Type casts

- **Can explicitly change the type via a cast**
  - C version is exactly the same as Java, and works in C++
  - Put the new type inside brackets **()**, **e.g.:**

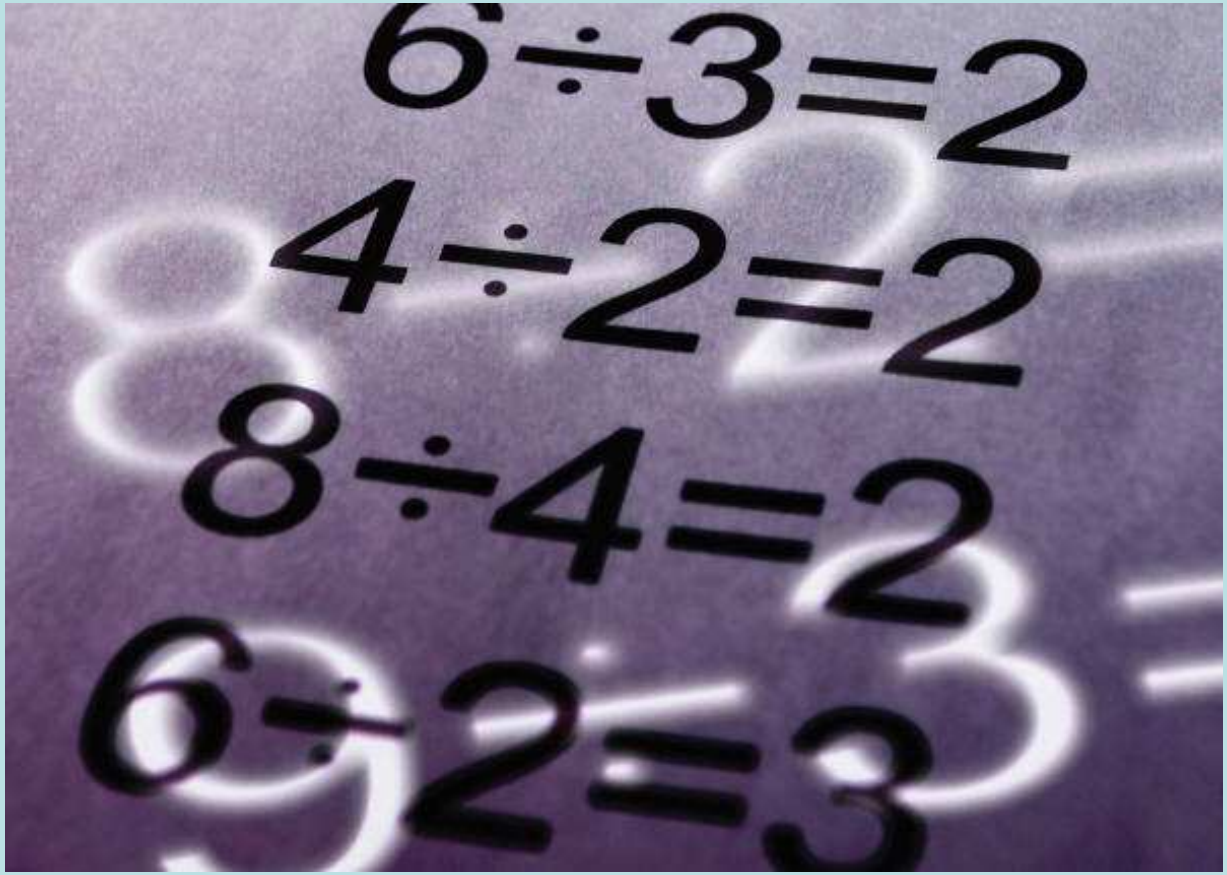
```
long l = 100L;
short s = (short)l;
```

  - Includes signed **<->** unsigned conversion

```
unsigned int ui = (unsigned int)i;
```

- **C++ also adds new types of casts**
  - `… = static_cast<NEWTYPE>(VARIABLE);`
  - `… = dynamic_cast<NEWTYPE>(VARIABLE);`
  - `… = const_cast<NEWTYPE>(VARIABLE);`
  - `… = reinterpret_cast<NEWTYPE>(VARIABLE);`
  - **E.g.** `int i = static_cast<int>(longValue);`
  - Safer and better, see later lecture

Operators (same as Java)



Sample Operator Precedence List

- Operators are evaluated in a specific order
  - Highest operator precedence applies first
- Examples (highest to lowest, not complete)

Increasing precedence ↑

(), [], ++, --	Grouping, array access, post increment/decrement
++, --, *, &	Pre-increment, dereference, address of (right to left)
*, /, %	Multiplication, division, modulus
+ -	Addition, subtraction
<, <=, >, >=	Comparison
==, !=	Comparison: equal to, not equal to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Ternary conditional
=, +=, -= etc	Assignment and '… and assign' (right to left)

Operator precedence matters

&& has higher precedence than **||**

```
if (  a && b  ||  c && d  )
```

means

```
if ( (a && b) || (c && d) )
```

```
if ( a ||  b && c  || d )
```

means

```
if ( a || (b && c) || d )
```

Operators and precedence

- **Operator precedence matters!**
- **Many style guides state that operator precedence should not be relied upon**
  - Makes code less readable
  - Prone to reliability of programmer's memory
- **I will NOT mark you down for adding unnecessary brackets (within reason)**
  - I do it where I think it aids clarity
  - 'Company' coding standards often require them
- **<u>But</u> you need to know the precedence rules**
  - To understand code written by others
  - An exam question may rely on them

Pointers
An introduction/reminder

Variables: size and location

Every variable has:

A name:	In your program only
An address:	Location in memory at runtime
A size:	Number of bytes it takes up
A value:	The number(s) actually stored

Does it matter:

1) Where a variable is stored?
2) How big a variable is?

Variables and memory

- C/C++ let you find out:
  - Where variables are in memory
  - How big they are
- In Java we don't care
  - In C/C++ we **MAY** care
  - We can take advantage of this for faster code
- I am going to use the kind of table on the right (in yellow) throughout these examples (& later lectures)
- Assume all variables are local variables – defined within some function

Example, local variables:

```
short s1, s2;
long l1, l2;
char c1,c2,c3,c4;
```

Address	Name	Type	Size
1000	s1	short	2
1002	s2	short	2
1004	l1	long	4
1008	l2	long	4
1012	c1	char	1
1013	c2	char	1
1014	c3	char	1
1015	c4	char	1

IMPORTANT WARNINGS

- Addresses in diagrams are for illustration only
- Actual positions of data in memory depend upon
  - Compiler
  - Operating system
  - Whether optimisation is turned on
- For example, you cannot assume:
  - That local variables will be in adjacent areas in memory
  - The ordering of the bytes in a multiple byte data type
- **DO NOT RELY ON POSITIONS OF DATA**
  - UNLESS YOU KNOW THEY ARE FIXED
  - There are **some** guarantees (within arrays and structs)

Address of : &

- We can ask for the address of a variable
  - And we can ‘write it down’ somewhere
  - This is like asking where someone lives
- Use the **`&`** operator in C/C++
- E.g.: If we have:

```
long longvalue = 345639L;
```

- Then: **`&longvalue`** is the address where the variable **`longvalue`** is stored in memory
  - Like the address of a person in a street/town
- Now we just have to store the address…

Data type for an address?

- But what type of data is an address?
  - i.e. `&longvalue` is of type ???
  - Is it a **number**?
  - Is it **2 numbers** combined?
    - e.g. segmented memory architecture (Win 3.1)
  - Is it an `int`?
  - Is it a `long`?
  - Is it a `short`?
  - **How are we going to store it?**
- **Question: Any ideas?**

Pointers

- We need pointer types!

- Remember: `*` is used to denote a pointer
  - i.e. a variable which will hold the address of some other variable
- Examples:

  `char*` is a pointer to a `char`

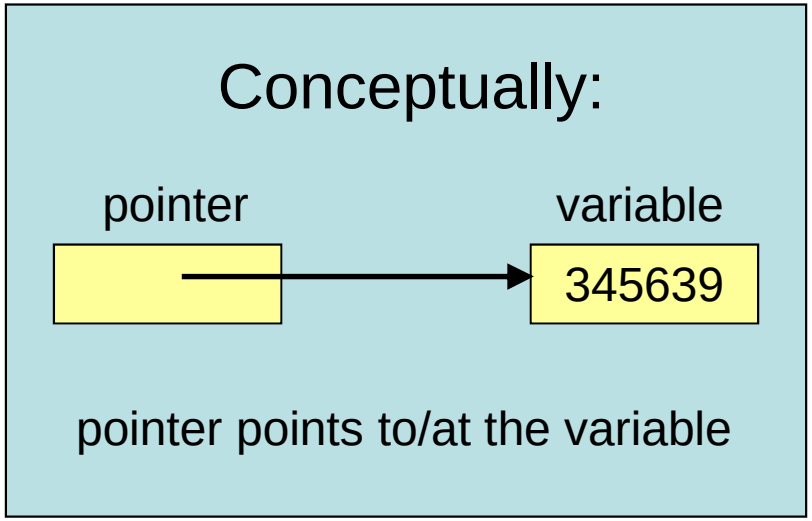
  `int*` is a pointer to an `int`

  `void*` is a generic pointer, an address of some data of *unknown* type (or a 'generic' address)

- Remember two things about pointers:

1. The **value** of the pointer is an **address** in memory
2. The **type** of the pointer says what **type** of data the program should **expect** to find at the address

The concept of a pointer



- You can think of pointers whichever way is easier for you
  1. As an **address** in memory and a **type**
  2. As a way of **pointing** to some other data, and a record of what type of data you think the thing pointed at is

Putting & and * together

- Example:
  - Create a long variable

```
long l = 345639L;
```

  - Take the address and store it in a `long*` variable
    - i.e. in a **pointer** to a **long**

```
long* pl = &l;
```

Conceptually:

pl points to/at l

Actually: (example addresses)

Address	Name	Type	Value
1000	l	long	345639
3056	pl	long*	1000

pl's value is the address of l

Sending a letter

- You can ask for somebody's address and use it to send a letter
- The postman/woman does not need to know who lives there
- He/she can deliver to the address, regardless of who is there

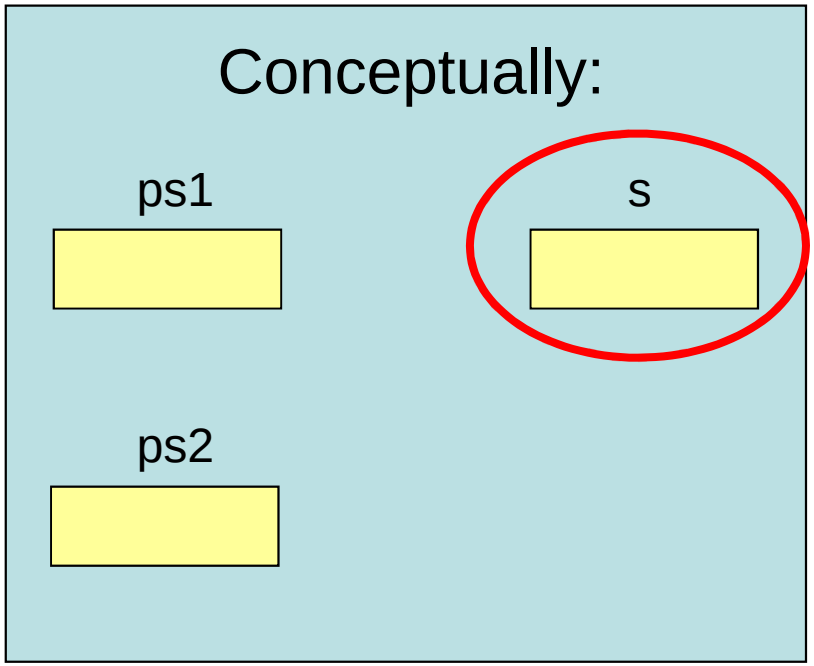
Pointers and addresses

- A pointer is like an address in an address book
- You can keep multiple copies of an address
  - You can copy the address into another place
- You can change the address
- You can use it to send a letter or visit a friend

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

- **Q: What goes into the red circled parts?**



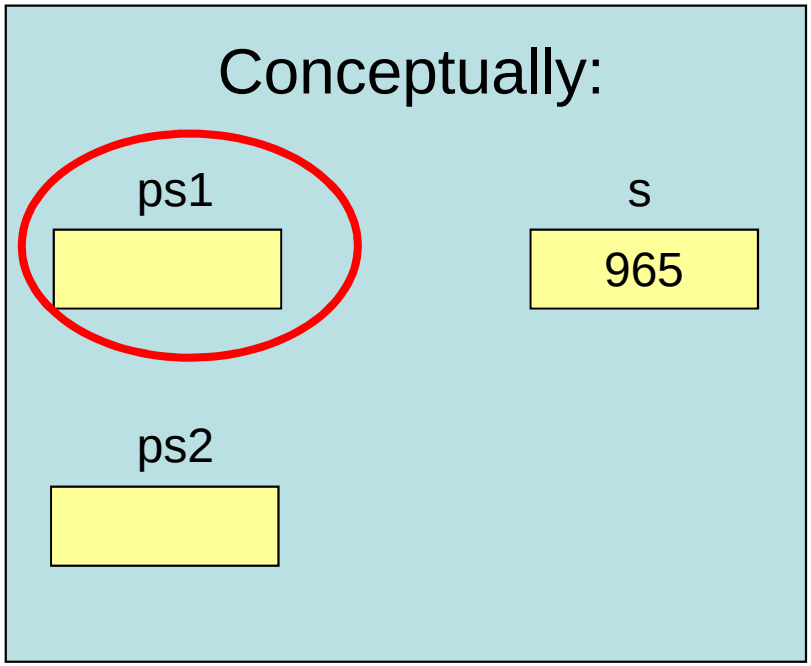
Actually: (example addresses)

Address	Name	Value
3000	s	
5232	ps1	
6044	ps2	

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

- **Q: What goes into the red circled parts?**



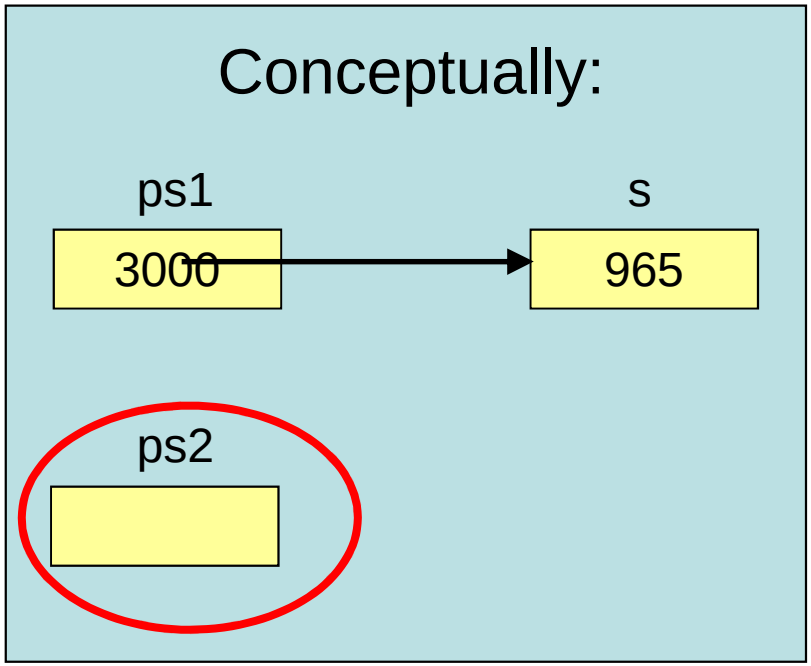
Actually: (example addresses)

Address	Name	Value
3000	s	965
5232	ps1	
6044	ps2	

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

- **Q: What goes into the red circled parts?**



Actually: (example addresses)

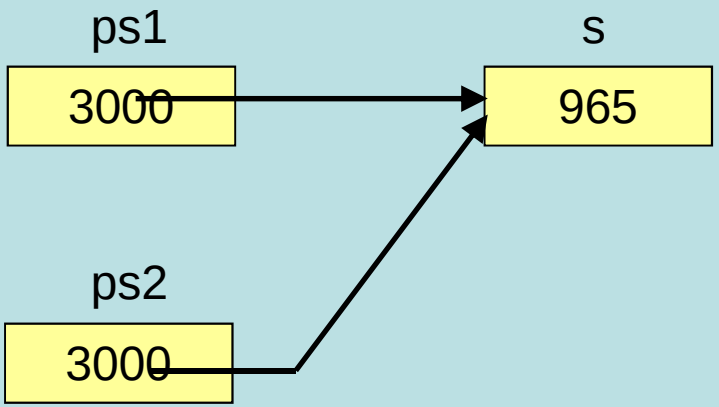
Address	Name	Value
3000	s	965
5232	ps1	3000
6044	ps2	

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

- **So, assigning one pointer to another means:**
  - **It points at the same object**
  - **It has the same address stored in it (i.e. the same value)**

Conceptually:



Actually: (example addresses)

Address	Name	Value
3000	s	965
5232	ps1	3000
6044	ps2	3000

Sending a letter (again)

- **Does the postman/woman need to know the person it is being delivered to in order to deliver the letter?**

Sending a letter

- Does the postman/woman need to know the person it is being delivered to in order to deliver the letter?

- **No!**
  It is sufficient that he/she knows where the recipient lives!

Weird Avenue

- We can use an address to find someone and do something to them
- We don't **need** to know who lives there, or what the house is like, just **where** it is
- E.g. "The person who lives in 3 Weird Avenue must pay this bill"
  - You can make the person pay without knowing them
- Or: "Give this present to the person at 1 Weird Avenue"
  - You can give the person a present without knowing who they are

The buildings on Weird Avenue

No. 1

No. 2

No. 3

No. 4

Dereferencing pointers

- We can use the thing pointed at, without knowing what it is
  - e.g. without knowing which variable it is
- As long as we know what *type* of thing it is

- Getting the thing pointed at is called **de-referencing the pointer**

Dereferencing operator : *

- The `*` operator is used to access the ‘thing’ that a pointer points at
- For example: define a `char` and `char*`

```
char c1 = 'h';
char* pc2 = &c1; // pc2 is a pointer to c1
```

- Ask for the value of the thing pc2 points at

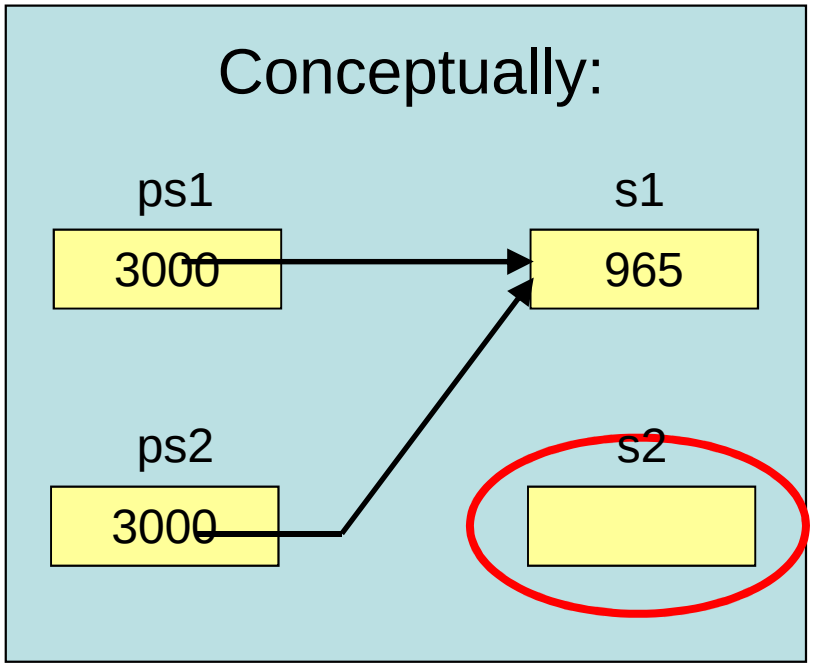
```
char c3 = *pc2;  // *pc2 is thing pointed at
```

- Thinking in terms of pointers holding addresses…
  - `pc2` is a `char*`, so it is the address of a char
  - `*pc2` is the `char` pointed at, i.e. `c1`!
  - So, `*pc2` is (now) another name for `c1`

Dereferencing example

```
short s1 = 965;
short* ps1 = &s1;
short* ps2 = ps1;
short s2 = *ps2;
```

- **What goes into the red circled parts?**
  - Hint: What is ***ps2**?



Actually: (example addresses)

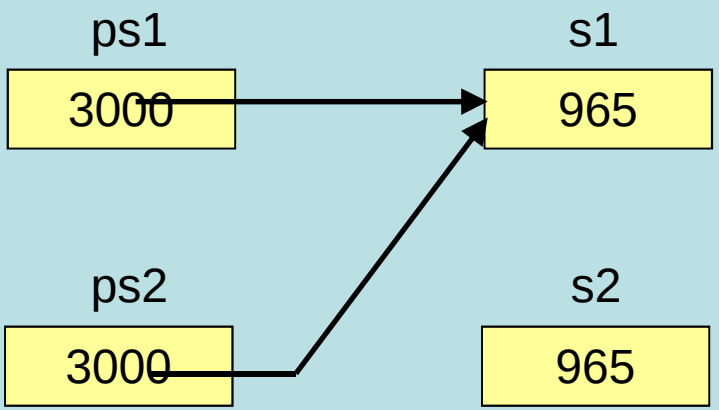
Address	Name	Value
3000	s1	965
5232	ps1	3000
6044	ps2	3000
6134	s2	

Dereferencing example

```
short s1 = 965;
short* ps1 = &s1;
short* ps2 = ps1;
short s2 = *ps2;
```

- So, we can access (use) the value of **`s1`** without knowing it is the value of variable **`s1`** (just the value **at** address **`ps2`**)

Conceptually:



Actually: (example addresses)

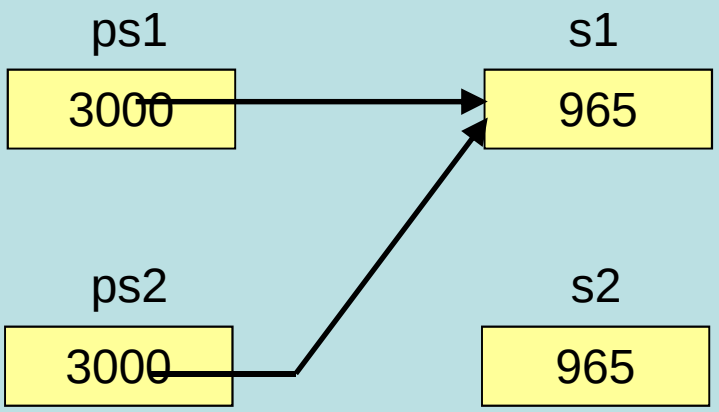
Address	Name	Value
3000	s1	965
5232	ps1	3000
6044	ps2	3000
6134	s2	965

Dereferencing example

```
short s1 = 965;
short* ps1 = &s1;
short* ps2 = ps1;
short s2 = *ps2;
*ps1 = 4;
```

Q: What does this do?

Conceptually:



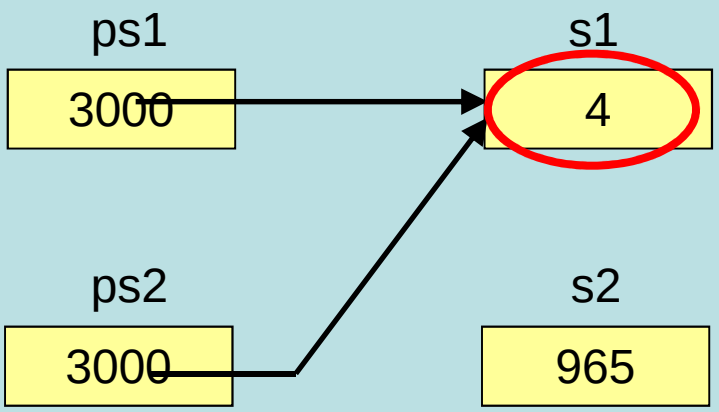
Actually: (example addresses)

Address	Name	Value
3000	s1	965
5232	ps1	3000
6044	ps2	3000
6134	s2	965

Dereferencing example

- '`*ps1 = 4`' changes the value pointed at by `ps1`
- We can change the thing pointed at without knowing what variable the address actually refers to (just 'change the value at this address')
- The value of `s1` changed without us mentioning `s1`

Conceptually:



Actually: (example addresses)

Address	Name	Value
3000	s1	4
5232	ps1	3000
6044	ps2	3000
6134	s2	965

Uninitialised Pointers

- In C and C++, variables are NOT initialised unless you give them an initial value
- Unless you initialise them, the value of a pointer is undefined
  - Always initialise all variables, including pointers
  - You can use NULL
- Dereferencing an unitialised pointer has undefined results
  - Could crash your program (likely)
  - Could crash your computer (less likely)
  - Could wipe your hard drive? (unlikely)

Next lecture

Pointers and arrays
char* and strings
argc and argv