CMPT 125

C Syntax

Crash Course in C

- Running a C program
- Compilation
- Python and C
- Variables and types
- Data and addresses
- Functions
- Performance

Running a C Program

Running a C Program

- Edit or write your program
  - Using a text editor like `gedit`
  - Save program with a `.c` extension
- Compile your program
  - Using `gcc`
  - This generates `a.out`
- Run your program
  - By typing `./a.out`

Using a Text Editor

- Write your C program using a simple editor
  - Like *Notepad* for Windows, or
  - *TextEdit* for the Mac, or
  - *gedit* for Linux
- *gedit*, and other editors highlight text for C syntax
- Save your program with a `.c` extension
  - Programs, like variables and functions, should be given sensible names

C Program Skeleton

```
#include <stdio.h>

int main(){

}
```

#include is like *import* in Python

When your program is run it calls the *main* function

{}s denote a block of code, like indentation in Python

C Hello World Program

```
#include <stdio.h>

int main(){
    printf("Hello World!\n");
}
```

printf is the standard output function – and is in *stdio.h*

All statements end with a ;

\n is the *newline* character, this statement prints a new line after the message

Compiling Your Program

- Save your program as a `.c` file
  - Let's say we've called it `hello.c`
- Compile your program using *gcc*
  - *gcc* used to stand for *Gnu C Compiler*
  - Now stands for *Gnu Compiler Collection*
- Open a console window and run *gcc* at the prompt
  - `>$ gcc hello.c`
  - If the command is successful it creates an executable program called `a.out`

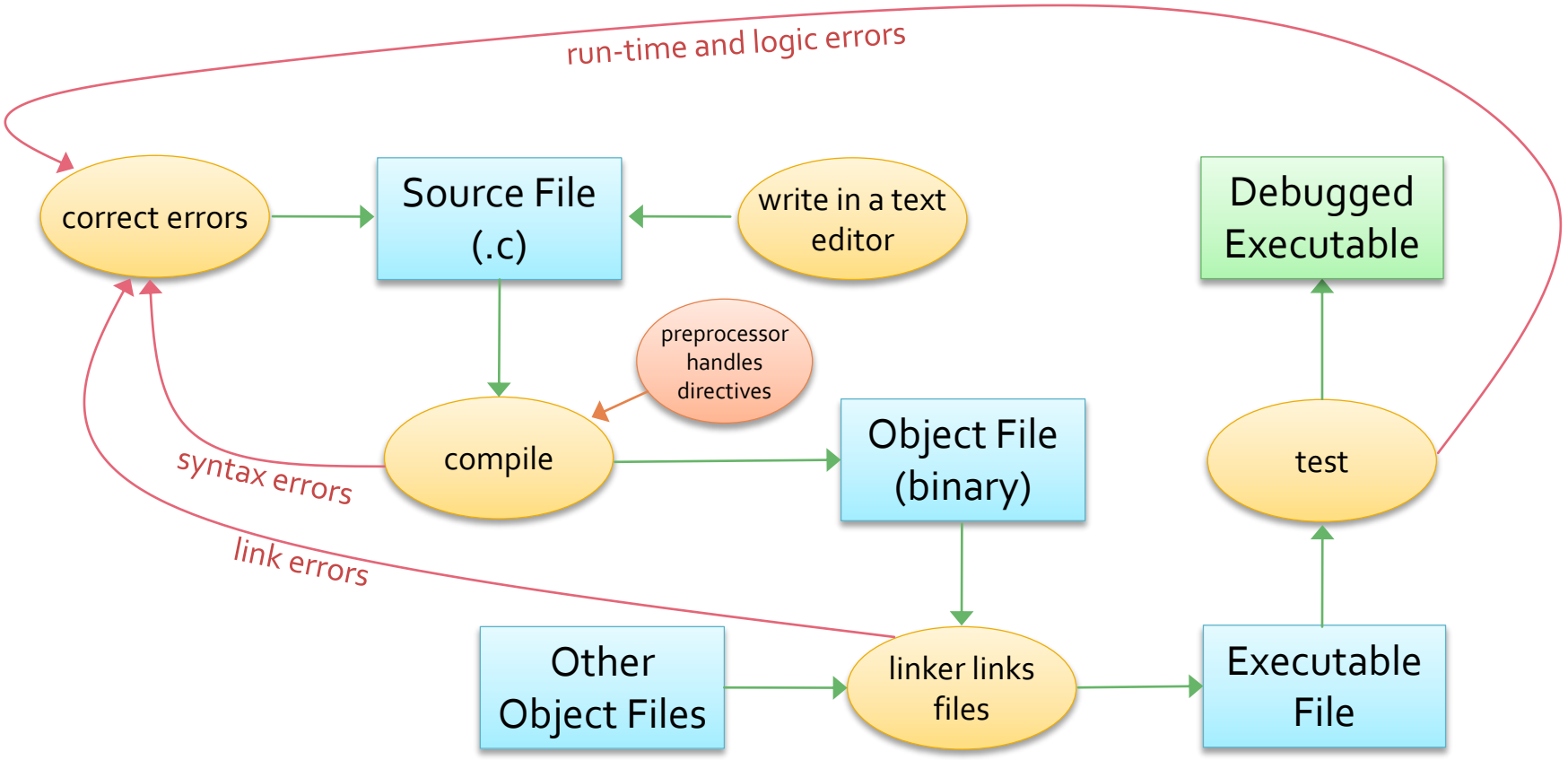
Running Your Program

- Run your program at the command prompt
  - By entering `./a.out`
  - `>$ gcc hello.c`
  - `>$ ./a.out`
  - `Hello World!`
  - `>$`
- When compiling your programs it is useful to name the output program something sensible
  - By using the `-o` flag
  - `>$ gcc -o hello hello.c`
  - Now the program is called *hello* instead of *a.out*

Compilation

In Linux using GCC

Processing a C Project



From Python to C

- The Python IDLE editor can be used as an interpreter
  - That processes one instruction at a time
- C programs have to be compiled
  - A compiler translates the entire program
  - Into machine language
- A machine language program can be directly processed by a computer
  - Each instruction is represented in binary
  - Machine languages are very hard for humans to read and write

Assembly Language

- Assembly languages are higher level than machine languages
  - But lower level than C
  - Operation codes are used to identify instructions
  - Memory addresses are given labels
    - Like very basic variable names
  - An assembler translates an assembly language to machine language
    - Where operation codes and memory addresses are binary

```
        .section
__TEXT,__text,regular,pure_instructions
        .globl    _main
        .align    4, 0x90
_main:                                  ## @main
        .cfi_startproc
## BB#0:
        pushq    %rbp
Ltmp2:
        .cfi_def_cfa_offset 16
Ltmp3:
        .cfi_offset %rbp, -16
        movq    %rsp, %rbp
Ltmp4:
        .cfi_def_cfa_register %rbp
        subq    $16, %rsp
        leaq    L_.str(%rip), %rdi
        movl    $0, -4(%rbp)
        movb    $0, %al
        callq    _printf
        movl    $0, %ecx
        movl    %eax, -8(%rbp)     ## 4-byte Spill
        movl    %ecx, %eax
        addq    $16, %rsp
        popq    %rbp
        ret
        .cfi_endproc

        .section    __TEXT,__cstring,cstring_liter
als L_.str:                             ## @.str
        .asciz     "Hello World!\n"

.subsections_via_symbols
```

Formal Languages

- C is a high level programming language
  - It can be compiled into machine code
  - And executed on a computer
- Programming languages are formal and lack the richness of human languages
  - If a program is *nearly*, but not quite syntactically correct then it will not compile
  - The compiler will *not* "figure it out"

Python and C

Python and C

- Python
  - print arg1, . . .
  - arg1 = raw_input()
  - int, float, str, bool, …
  - variables declared during execution
  - and, or, not
  - if-elif-else
  - for i in range(n)
  - indented blocks
  - lists may grow/shrink
- C
  - printf(format, arg1, . . .)
  - scanf(format, &arg1, . . .)
  - int, float, char, …
  - variables declared at compile time
  - &&, ||, !
  - if { } else if { } else { }
  - for (i = 0; i < n; i++) { }
  - { blocks in curly braces }
  - arrays are fixed size

Variables and Types

Variable Declaration

- Variables must be declared before being used

```
int main(){
  int a = 5;
  int b = 17;
  printf("Sum of %d + %d is %d", a, b, a+b);
}
```

- `int a = 5;` declares an *int*eger variable named *a* and gives it an initial value of 5
  - Declaration does not have to include initialization
    - `int a; //declares an integer called a`
  - Un-initialized variables may have garbage values

Strong Typing

- The type of a variable in C cannot be changed
  - Once a variable is declared as an *int* it stays an *int*
    - Or a *char*, *float*, *double*, etc.
  - It is possible to change the type of a variable in Python
- When the program is run space is reserved for variables in main memory
  - Usually 4 bytes for an *int* or a *float*
  - Usually 8 bytes for a *long long* or a *double*
  - Usually 1 byte for a *char*

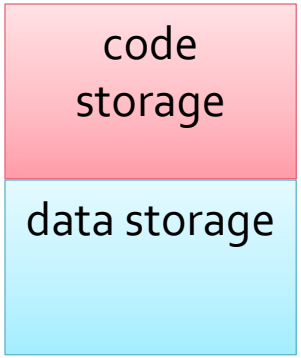
Memory Model

- Variables are stored in unique locations in memory
  - This location is referred to as its address
    - Which is represented by an integer
- A variable can therefore be described in three ways
  - Its type (e.g *int*)
  - Its value (e.g. 42)
  - Its address (its main memory location)
- Sometimes a program need to explicitly use the address of a variable

Data and Addresses

Memory Management

- When a program runs it requires main memory (RAM) space for
  - Program instructions (the program)
  - Data required for the program
- There needs to be a system for efficiently allocating memory
  - We will only consider how memory is allocated to program data (variables)



RAM

- RAM can be considered as a long sequence of bytes
  - Starting with 0
  - Ending with the amount of main memory (-1)
- RAM is addressable and supports *random access*
  - That is, we can go to byte 2,335,712 without having to visit all the preceding bytes

RAM Illustrated

Consider a computer with 1GB * of RAM

*1 GB = 1,073,741,824 bytes

RAM can be considered as a sequence of bytes, addressed by their position

0	1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16	…
…	1073741816	1073741817	1073741818	1073741819	1073741820	1073741821	1073741822	1073741823

This is a simplified and abstract illustration

Variable Declaration

- Declaring a variable reserves space for the variable in main memory
  - The amount of space is determined by the type
- The name and location of variables are recorded in the *symbol table*
  - The symbol table is also stored in RAM
  - The symbol table allows the program to find the address of variables
    - We will pretty much ignore it from now on!

Variable Declaration Example

For simplicity's sake assume that each address is in bytes and that memory allocated to the program starts at byte 2048

```
int x, y;
x = 223;
x = 17;
y = 3299;
```

Creates entries in the symbol table for *x* and *y*

variable	address
x	2048
y	2052

These lines change the *values* stored in *x* and *y*, but do not change the location or amount of main memory that has been allocated

data	17				3299											
address	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	...

Simple Memory Model

- Variables are stored in main memory
  - We can find out the value of a variable
  - And its address
    - To retrieve the address write the variable name preceded by an ampersand (&)
- The *value* of a variable can be changed by assignment
  - But its storage location and the amount of memory allocated to a variable cannot change

Printing Variables

- We can use the *printf* function to print the value of a variable, or its address
  - int x = 12;
  - printf("x = %d, ", x);
  - printf("the address of x is %d", &x);

I wouldn't usually use *x* or *y* as the name of a variable since it doesn't convey any meaning, but in this example they are just arbitrary values

- Here is another example
  - float y = 2.13;
  - printf("y = %f, ", y);
  - printf("the address of y is %d", &y);

Note the use of *%f* to print a floating point value, and also note that the address of *y* is still an integer so its format specification is still *%d*

Pointers

- A variable can be declared that stores the *address* of another variable
  - Such variables are referred to as pointers
  - Pointers are declared with the type of the variable they point to followed by an *
- Pointers are used for a number of reasons including
  - Passing addresses to functions
    - The first example of this is the input function, *scanf*
  - Declaring pointers to arrays in dynamic memory

Input with scanf

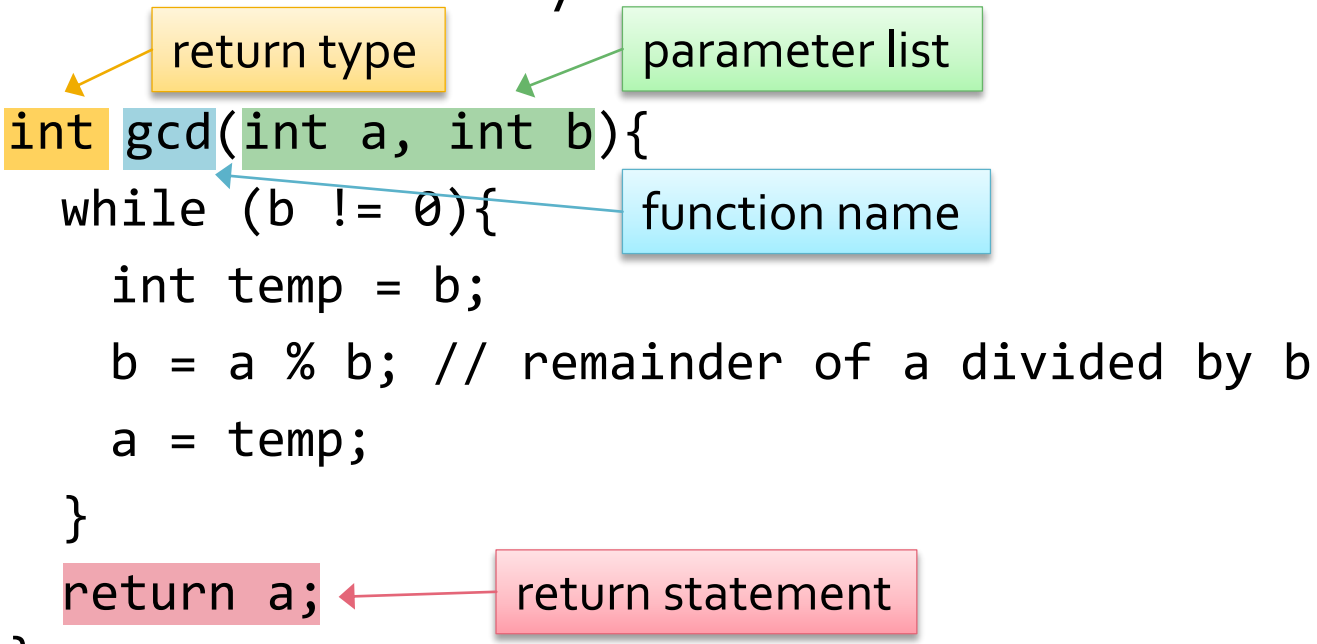
- The *scanf* function requires the address of the variable that input is to be stored in

```
int main(){
  int a = 0, b = 0;
  printf("Enter an integer: ");
  scanf("%d", &a);
  printf("Enter another integer: ");
  scanf("%d", &b);
  printf("Sum of %d + %d is %d", a, b, a+b);
}
```

Functions

Functions

- Functions must be defined outside main
  - Note that main is itself a function
- Function anatomy



Pass By Value

- All C functions are pass by value
  - Data in the argument is copied to the parameter
    - The scope of the parameter is the scope of its function
  - This prevents *side-effects*, where the function can unexpectedly modify data passed to it
- Functions in Python are pass by reference
  - Which can result in side-effects but only when the data is mutable
- Java is a mix of pass by value and pass by reference

A (failed) Swap Function

- Let's say we want to write a function to swap the values in two variables
- Here is a first attempt

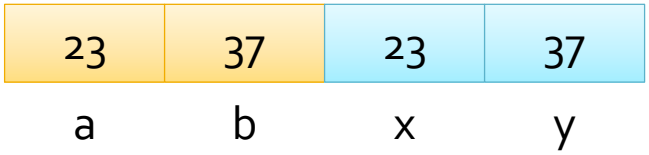
```
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Does this work?

Swap Function

```
// Calling code (in main)
int a = 23;
int b = 37;
swap(a, b);
```

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

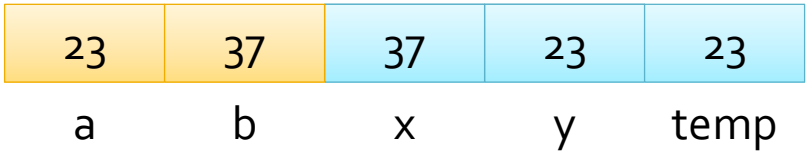


x and *y* are parameters of the swap function so have their own space in main memory

Swap Function

```
// Calling code (in main)
int a = 23;
int b = 37;
swap(a, b);
```

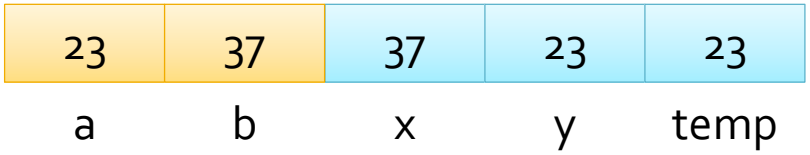
```
void swap(int x, int y)
{
        int temp = x;
        x = y;
        y = temp;
}
```



Swap Function

```
// Calling code (in main)
int a = 23;
int b = 37;
swap(a, b);
```

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



note that neither *a* nor *b*'s values have changed, so the swap function achieved **nothing**!

once swap has executed its memory is released

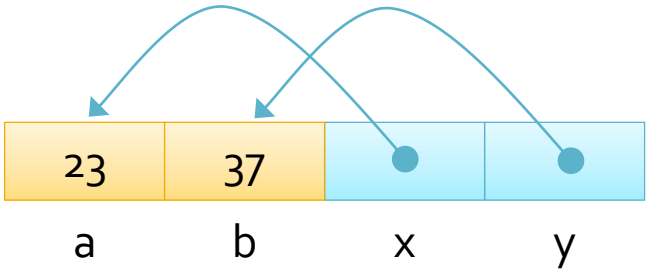
Passing Addresses

- Remember that *scanf* accepts the address of a variable as an argument
  - And that the value of the variable is changed once *scanf* has finished its execution
- We can specify an address using &
  - A function definition also needs to specify that it expects an address
    - Note that the *address* of a float is not the same as a float

Swap Function

```
// Calling code
int a = 23;
int b = 37;
swap(&a, &b);
```

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

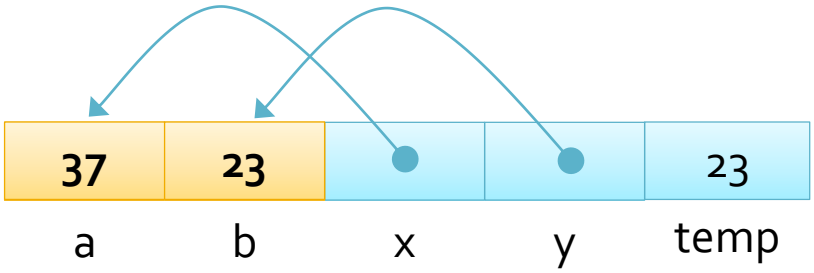


x and y contain the **addresses** of a and b, not their values

Swap Function

```
// Calling code
int a = 23;
int b = 37;
swap(&a, &b);
```

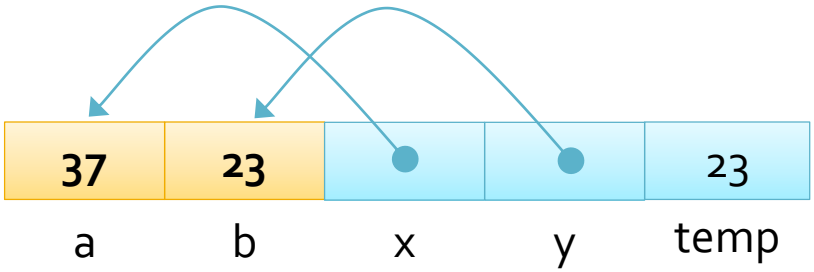
```
void swap(int* x, int* y)
{
        int temp = *x;
        *x = *y;
        *y = temp;
}
```



Swap Function

```
// Calling code
int a = 23;
int b = 37;
swap(&a, &b);
```

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



Because swap's parameters are passed addresses *a* and *b* have changed

once swap has executed its memory is released

What's with all the *s?

- The * is used to mean a number of different things, dependent on the context
  - Multiplication
  - **Declaration of a pointer variable**, when used after a type name
    - *int * p;* declares a pointer called *p* that will store the *address* of an *int*
  - **Dereferencing of a pointer** to access the variable that it points to
    - *p = 7; assigns 7 to the *int* that *p* points to

Pointers and Types

- Pointers store addresses
  - Addresses are always the same size on the same system
- So why do we have to say what type of data is going to be pointed to?
  - To reserve enough space for the data and
  - To ensure that the appropriate operations are used with the data

Declaring a Pointer

- Pointer variable are identified by an * that follows the type in the declaration
  - `int * p;`
- This declares a variable called *p* that will point to (or refer to) an integer
- Note that the type of a pointer is *not* the same as the type it points to
  - p is a *pointer to an int*, and *not* an int

Pointers and Values

- The operation shown below is unsafe
  - `int x = 12;`
  - `int *p = x;`

This is not a good thing to do and will result in a compiler warning or error

- Remember that the type of *p* is an address (to an *int*), and not an *int*
  - Addresses are actually whole numbers but assigning arbitrary numbers to them is a bad idea
  - Since a programmer is unlikely to know what is stored at a particular memory address

Address Operator

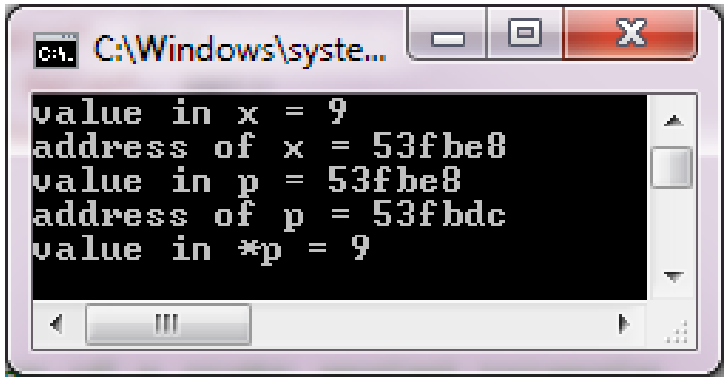
- Pointers can be assigned the address of an existing variable
  - Using the address operator, &
  - In this way it is possible to make a pointer refer to a variable

Dereferencing

- Pointers can be used to access variables
  - But only after they have been assigned the address of a variable
- To change the value of a variable a pointer points to the pointer has to be *dereferenced*
  - Using the * operator which can be thought of meaning *the variable pointed to*

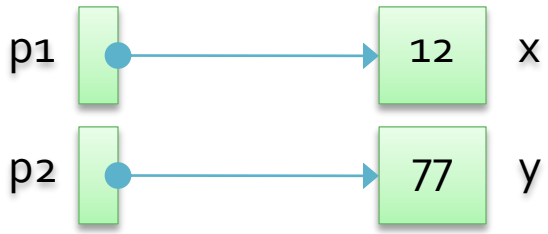
Pointer Assignment

```
int x = 5;
int *p = &x; //assign p the address of x
// Use p to assign 23 to x
*p = 9; //dereferences p
printf("value in x = %d\n", x);
printf("address of x = %x\n", &x);
printf("value in p = %x\n", p);
printf("address of p = %x\n", &p);
printf("value in *p = %d\n\n", *p);
```



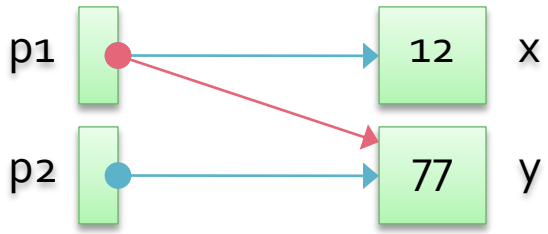
Pointers and Assignment

```
int x = 12;
int y = 77;
int *p1 = &x; //assign p1 the address of x
int *p2 = &y; //assign p2 the address of y
```



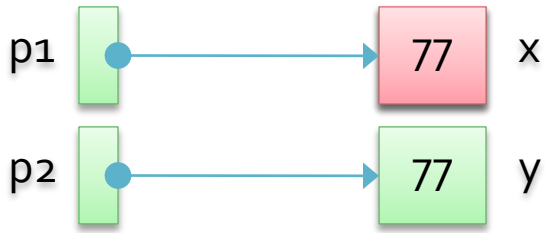
Pointers and Assignment

```
int x = 12;
int y = 77;
int *p1 = &x; //assign p1 the address of x
int *p2 = &y; //assign p2 the address of y
p1 = p2; //assigns the address in p2 to p1
```



Pointers and Assignment

```
int x = 12;
int y = 77;
int *p1 = &x; //assign p1 the address of x
int *p2 = &y; //assign p2 the address of y
*p1 = *p2;
```



Why Use Pointers?

- In practice we don't often use pointers like the preceding examples
- Pointers can be used to allow functions to change the value of their arguments
- They are also key to managing memory for objects that change size during run-time
  - Such objects are allocated space in another area of main memory – in *dynamic memory*

Performance

How Good is Your Code

- There are several measures
- Is it:
  - Correct (no bugs)?
  - Reliable?
  - Efficient?
  - Affordable?
  - Maintainable?
  - Easy to use?

How Good is Your Algorithm

- When we assess the performance of an algorithm we focus on its efficiency
- There are two main measures of efficiency
  - Time
  - Space (in main memory)
- Recently, time is considered to be the more important of these two
  - Memory is fairly cheap
  - Memory space is not usually a constraint
    - There are exceptions to this

Measuring Performance

- There are two main ways to measure performance
- Time the code on a variety of inputs
  - We can plot graphs and predict behavior
  - This measure is hardware dependent
- Count the number of operations performed by the algorithm
  - We can plot graphs or derive functions or
    - Use the big-O estimate
  - This measure is hardware independent