Introduction to the C Programming Language

Quick History

Designed by Dennis Ritchie

Developed by him and Bell Laboratories

First appeared in 1972 (43 years old)

Compilers Linux and OSX

On linux: gcc and clang

clang gives better warnings and error messages in my opinion

```
clang -Wall main.c -o main
```

creates a compiled executable called main

Run with ./main

Compilers Windows

Visual studio C++ compiler

Visual studio 2013 is now free for all non commercial use

A bit tricky to work from the commandline

Computer Memory

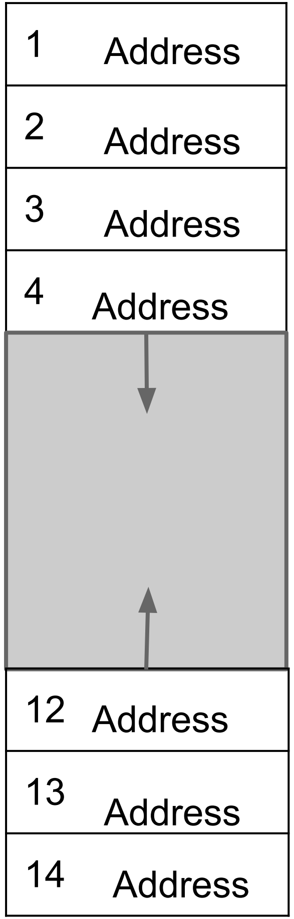
Stack

Grows and shrinks

Heap

gets allocated

and freed



Pointers

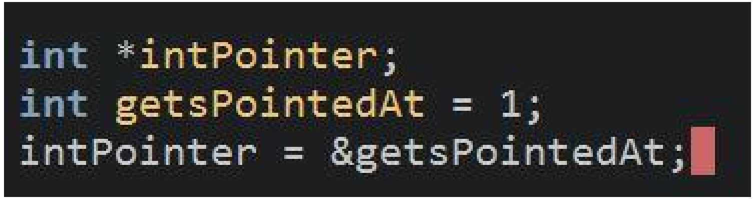
Is a variable that contains the memory address of another variable.

```
int *intPointer;
```

This Pointer can contain the memory address of a value of type int

Get the memory address

the ‘&’ symbol in front of a variable will give you its memory address

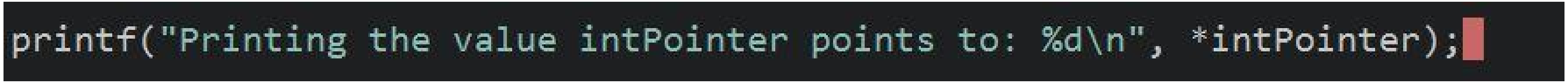


```
int *intPointer;
int getsPointedAt = 1;
intPointer = &getsPointedAt;
```

“intPointer” now points to the variable “getsPointedAt”.

Get the value a pointer points to

If you want the value the pointer points to you have to dereference it

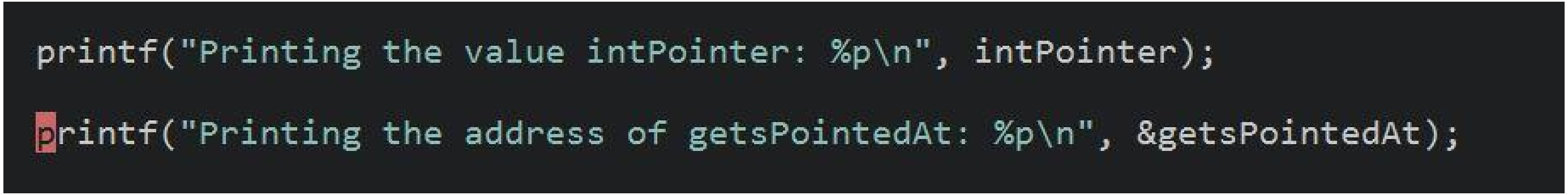


```
printf("Printing the value intPointer points to: %d\n", *intPointer);
```

Putting the * symbol in front dereferences the pointer and printf prints:

“Printing the value intPointer points to: 1”

More print-outs



```
printf("Printing the value intPointer: %p\n", intPointer);

printf("Printing the address of getsPointedAt: %p\n", &getsPointedAt);
```

These two printf’s prints the same thing, the memory address of the variable “getsPointedAt” in hexadecimal

“27F42CF828”

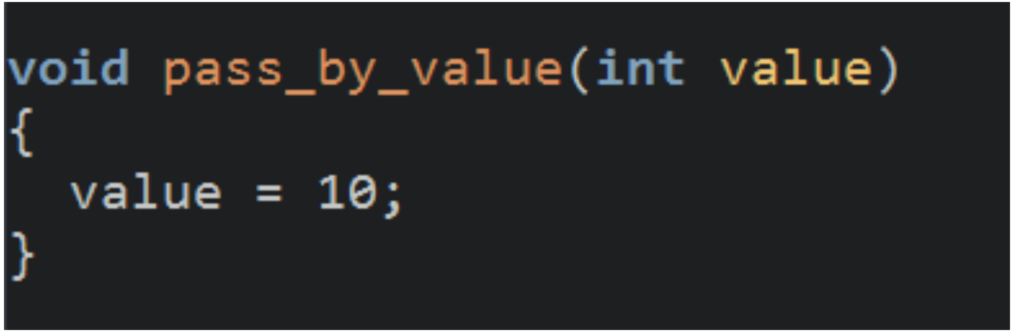
Pointers and functions

Pass by reference and pass by value

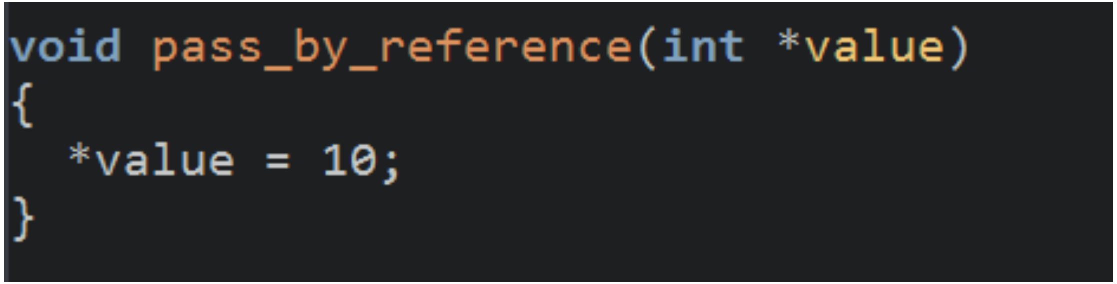
If you pass the address of a variable you can change the actual value

If you pass the value, the function gets a local copy that gets destroyed when the function returns

Two functions

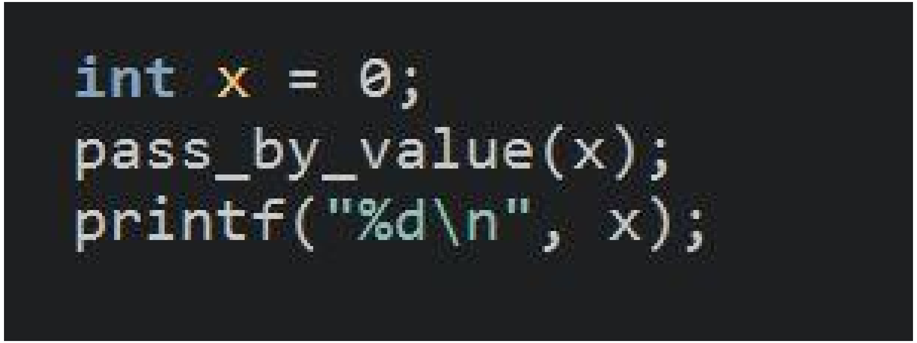


```
void pass_by_value(int value)
{
  value = 10;
}
```



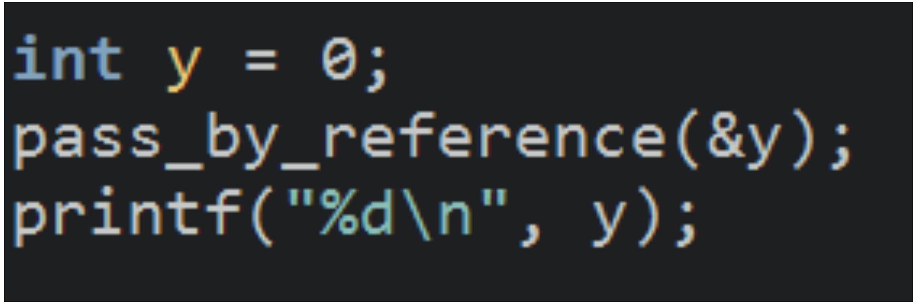
```
void pass_by_reference(int *value)
{
  *value = 10;
}
```

Calling both functions



```
int x = 0;
pass_by_value(x);
printf("%d\n", x);
```

Print-out: 0

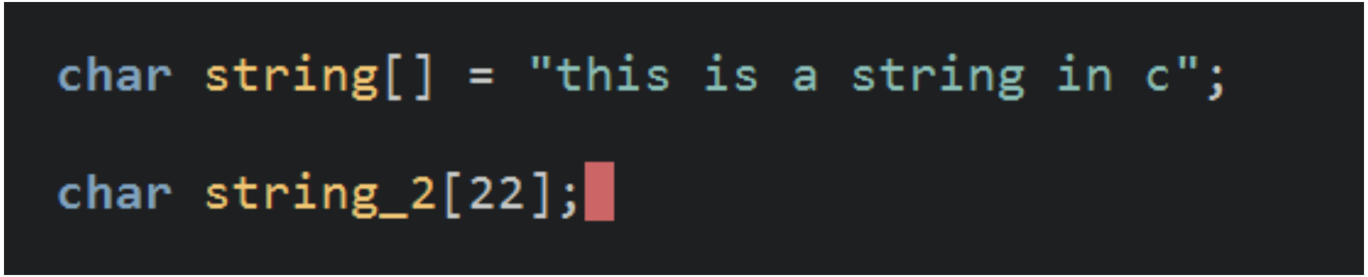


```
int y = 0;
pass_by_reference(&y);
printf("%d\n", y);
```

Print-out: 10

C Strings

A string is just an array of characters

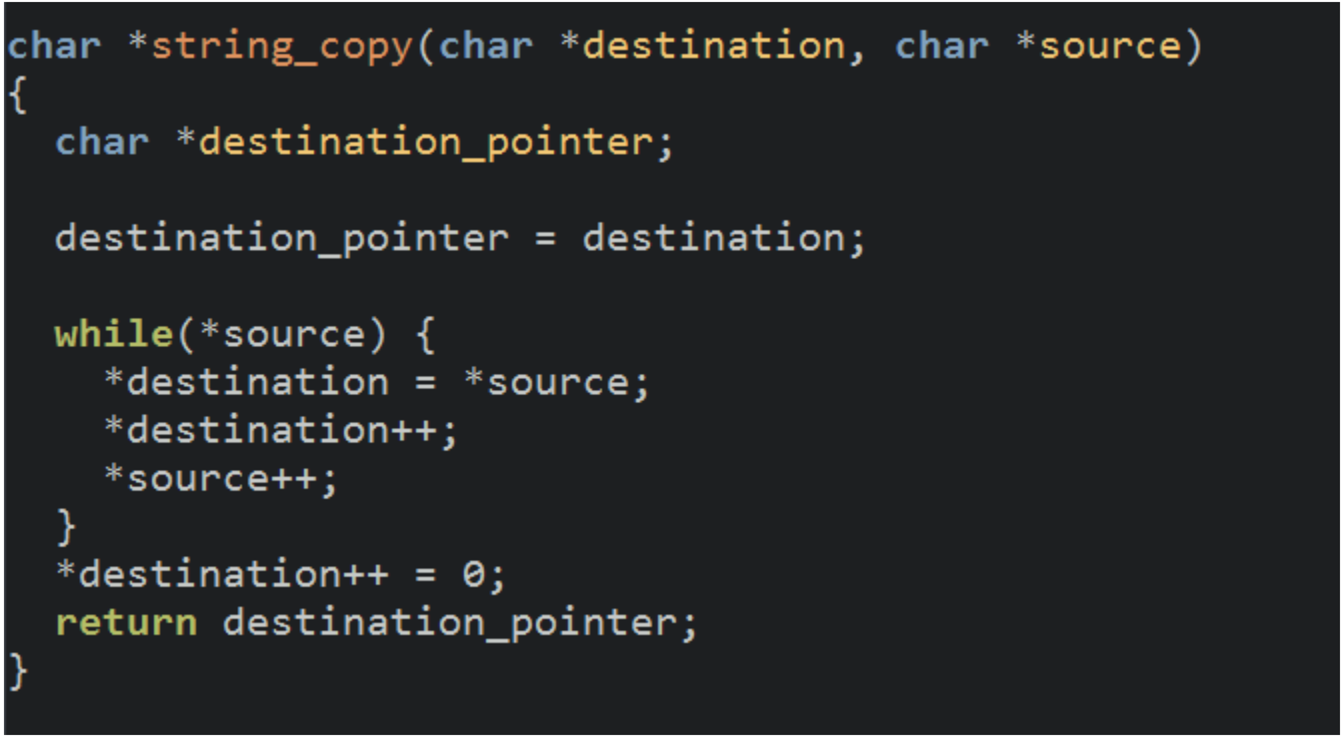


```c
char string[] = "this is a string in c";

char string_2[22];
```

Strings in C must end with a nullByte: \0 or 0

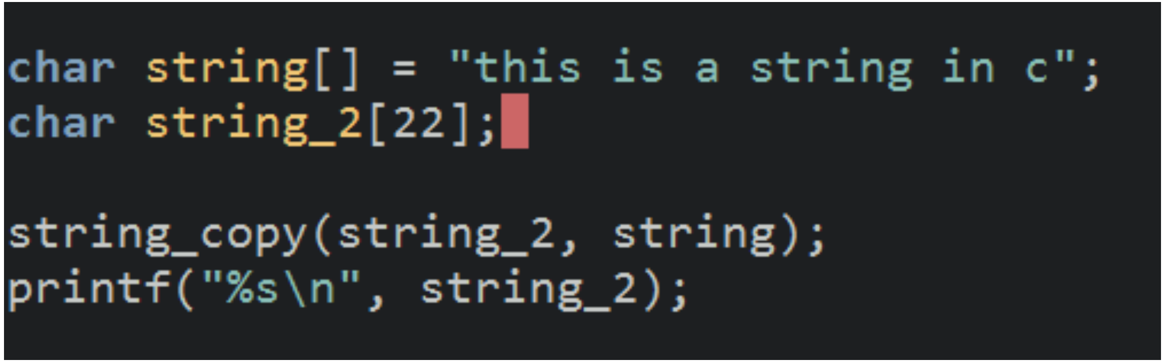
Strings and pointers

A pointer can point to the start of a string



```
char *string_copy(char *destination, char *source)
{
  char *destination_pointer;

  destination_pointer = destination;

  while(*source) {
    *destination = *source;
    *destination++;
    *source++;
  }
  *destination++ = 0;
  return destination_pointer;
}
```

The call



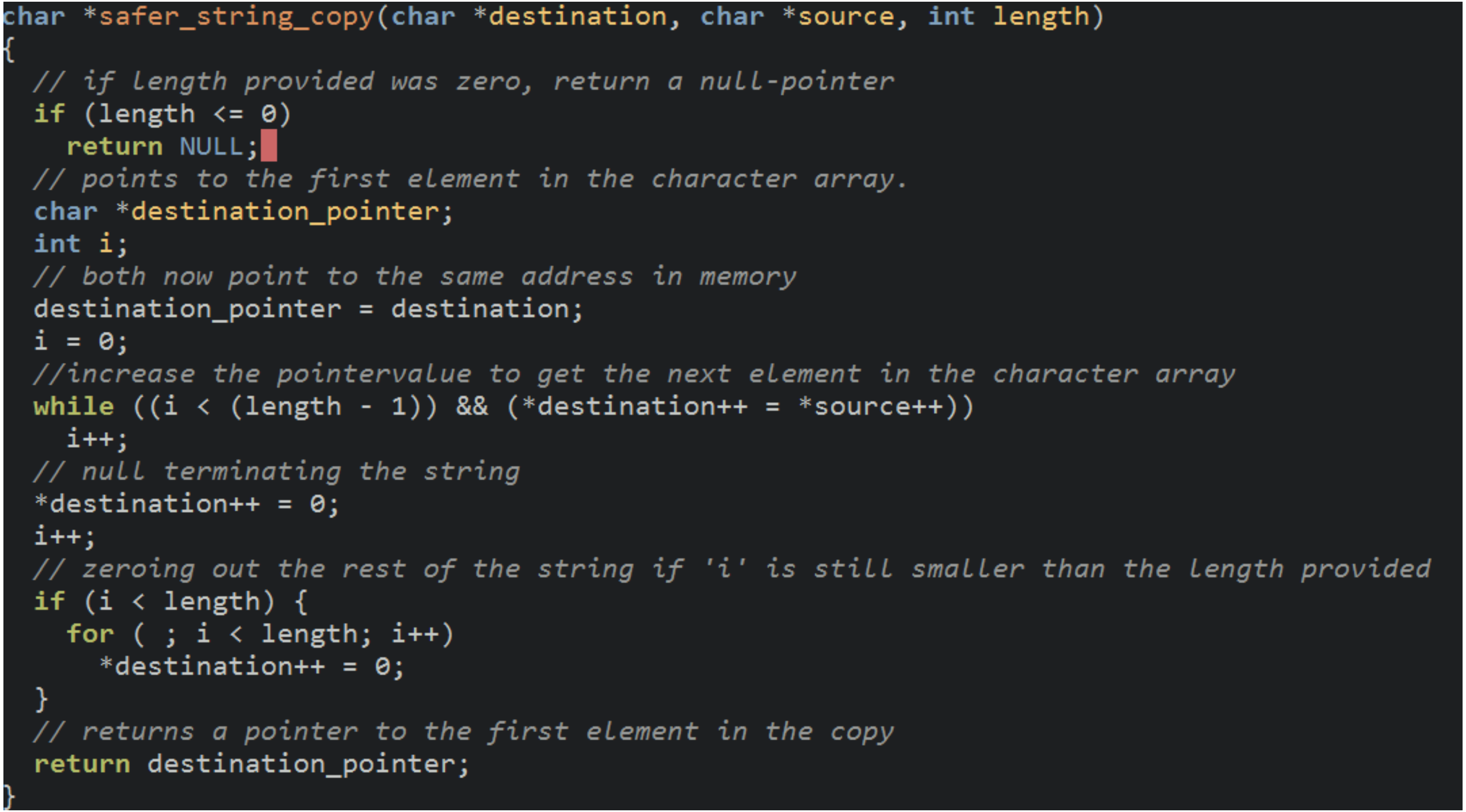
```
char string[] = "this is a string in c";
char string_2[22];

string_copy(string_2, string);
printf("%s\n", string_2);
```

Print-out: “this is a string in c”

Problems

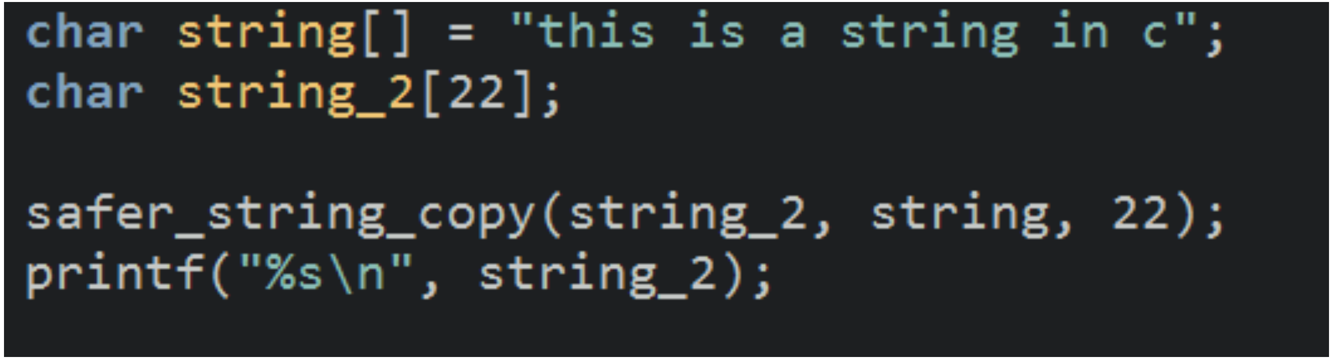
What happens if the string character is too small to hold the copied string?

Make sure the buffer is always big enough, or make a new function that takes the number of bytes to be copied



```c
char *safer_string_copy(char *destination, char *source, int length)
{
  // if length provided was zero, return a null-pointer
  if (length <= 0)
    return NULL;
  // points to the first element in the character array.
  char *destination_pointer;
  int i;
  // both now point to the same address in memory
  destination_pointer = destination;
  i = 0;
  //increase the pointervalue to get the next element in the character array
  while ((i < (length - 1)) && (*destination++ = *source++))
    i++;
  // null terminating the string
  *destination++ = 0;
  i++;
  // zeroing out the rest of the string if 'i' is still smaller than the length provided
  if (i < length) {
    for ( ; i < length; i++)
      *destination++ = 0;
  }
  // returns a pointer to the first element in the copy
  return destination_pointer;
}
```

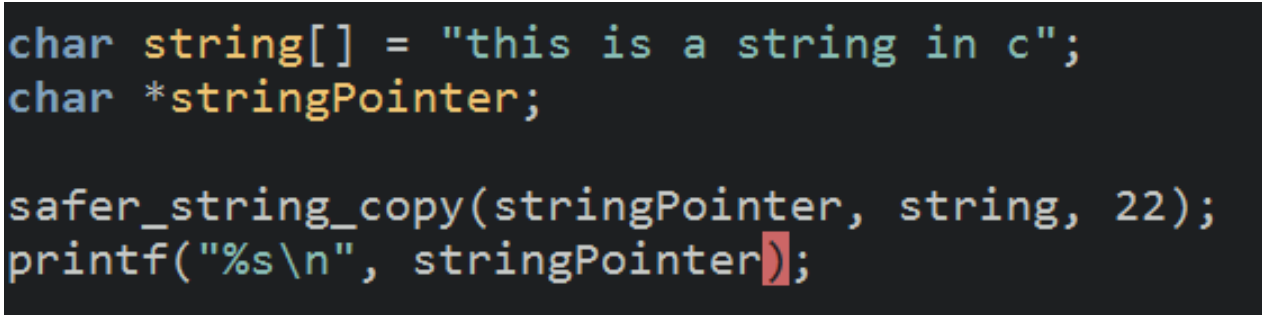
The call



```
char string[] = "this is a string in c";
char string_2[22];

safer_string_copy(string_2, string, 22);
printf("%s\n", string_2);
```

Print-out: “this is a string in c”

String Pointer

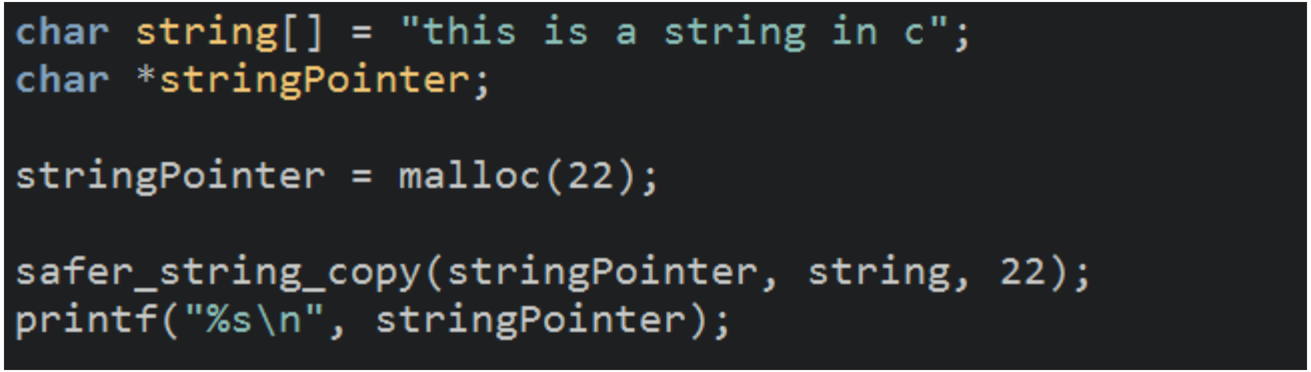


```
char string[] = "this is a string in c";
char *stringPointer;

safer_string_copy(stringPointer, string, 22);
printf("%s\n", stringPointer);
```

we said to copy 22 bytes, but the “charPointer” only points to one char (1 byte)

when we then increment the “charPointer” it will point to unallocated memory

Solution, Malloc



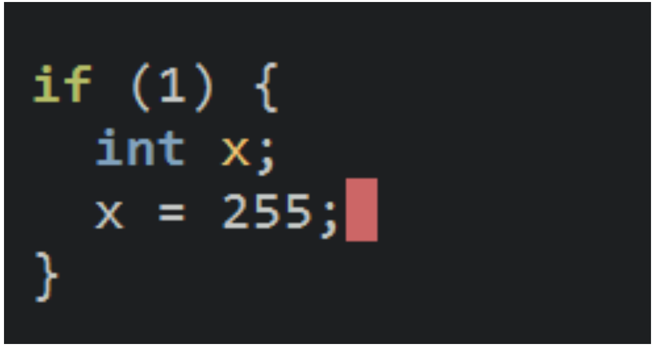
```
char string[] = "this is a string in c";
char *stringPointer;

stringPointer = malloc(22);

safer_string_copy(stringPointer, string, 22);
printf("%s\n", stringPointer);
```

We allocate enough bytes to the pointer

Fun with Pointers

Any variable created inside a code block can not be accessed outside the block

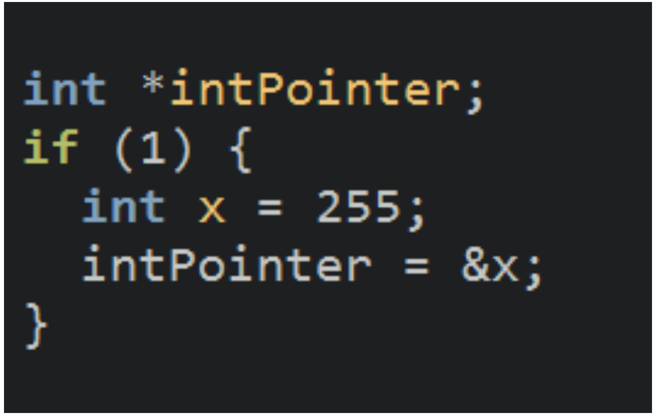
example:



```
if (1) {
  int x;
  x = 255;
}
```

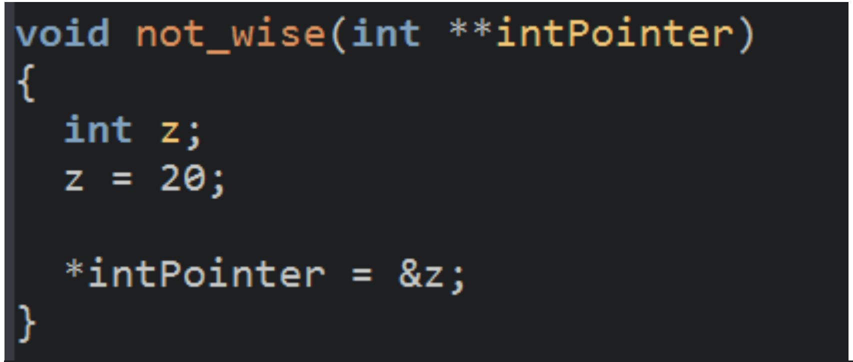
Undefined Behaviour

But with pointers we can actually get the value out (maybe).

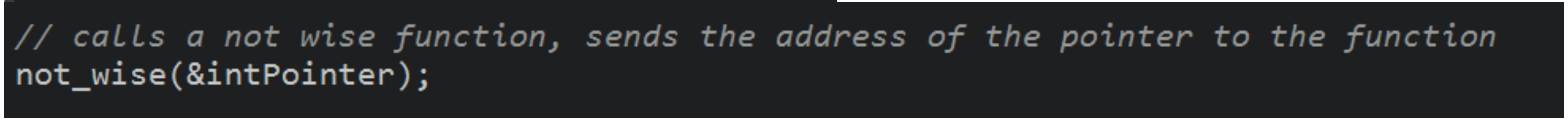


```
int *intPointer;
if (1) {
  int x = 255;
  intPointer = &x;
}
```

Pointer to a Pointer



```
void not_wise(int **intPointer)
{
  int z;
  z = 20;

  *intPointer = &z;
}
```

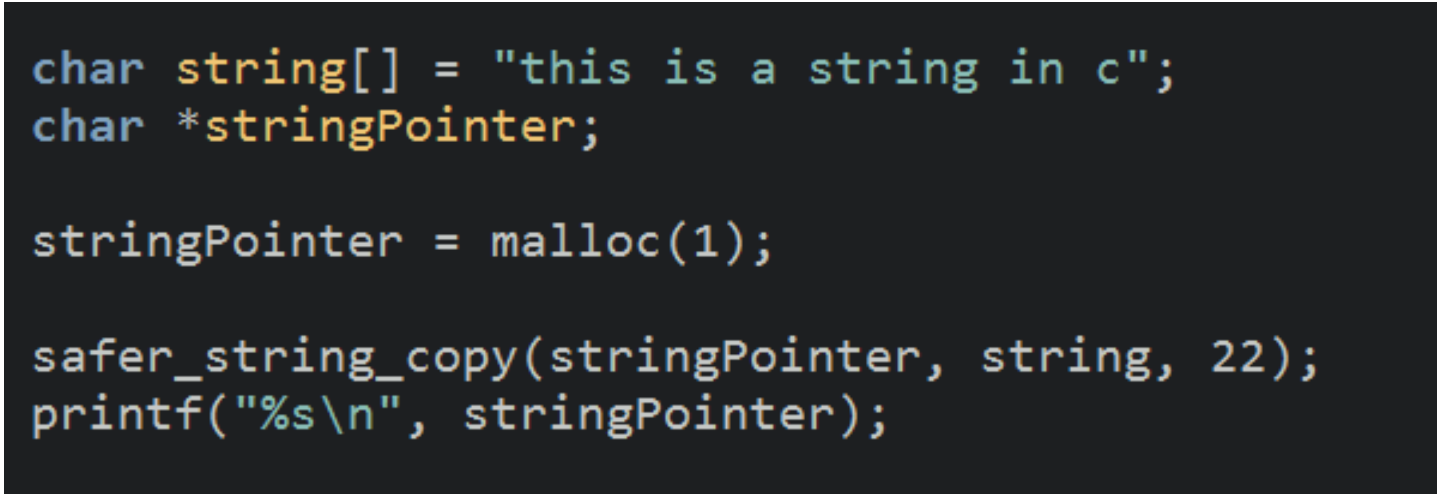


```
// calls a not wise function, sends the address of the pointer to the function
not_wise(&intPointer);
```

But pointers to pointers are really useful if used correctly.

Fun with malloc

You can get wired results with malloc



```c
char string[] = "this is a string in c";
char *stringPointer;

stringPointer = malloc(1);

safer_string_copy(stringPointer, string, 22);
printf("%s\n", stringPointer);
```

Important things not discussed

Structs, especially struct pointers

Sizeof

Bitwise operators

Debugging

Github

These slides, and a well documented source will be pushed to

https://github.com/Artigmann/C_intro_TG15

Everything will be up later today!