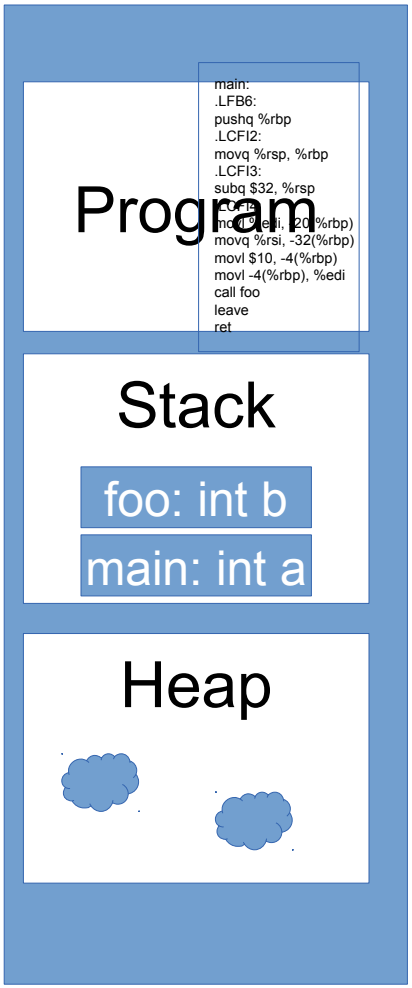
Function Pointers

Memory layout

```
int foo() {
    int b;
    ...
}
int main() {
    int a;
    foo();
}
```

compile

```
main:
.LFB6:
pushq %rbp
.LCFI2:
movq %rsp, %rbp
.LCFI3:
subq $32, %rsp
.LCFI4:
movl %edi, -20(%rbp)
movq %rsi, -32(%rbp)
movl $10, -4(%rbp)
movl -4(%rbp), %edi
call foo
leave
ret
```



Functions as pointers

- Function code is stored in memory
- So a function should have an address
- The address of a function is a “function pointer”
- Function pointers can be passed as arguments to other functions or return from functions

Why do we need function Pointers?

- Efficiency
  - Less # of if statements
- Runtime binding
  - Determine sorting function based on type of data at run time
    - Insertion sort for smaller data sets (n <100)
    - Better algorithms for large data sets

Define a Function Pointer

- int (*funcPointer) (int, char, int);

- funcPointer is a variable name pointing to a function
- The extra parentheses around (*funcPointer) is needed

Using function pointers

```
 //assign an address to the function pointer
int (*funcPointer) (int, char, int);

int firstExample ( int a, char b, int c){
 printf(“ Welcome to the first example”);
 return a+b+c;
}
funcPointer= firstExample; //assignment
funcPointer=&firstExample; //alternative using
    address operator
```

Assign an address to a Function Pointer

- It is optional to use the address operator & infront of the function's name
- Similar to the fact that a pointer to the first element of an array is generated automatically when an array appears in an expression

Comparing Function Pointers

- Can use the (==) operator

```
 //comparing function pointers
If (funcPointer == &firstExample)
   printf (“pointer points to firstExample”);
```

Calling a function using a Function Pointer

- There are two alternatives

```
// calling a function using function pointer
 int answer= funcPointer (7, ’A’ , 2 );
 int answer=(* funcPointer) (7, ’A’ , 2 );
```

Arrays of Function Pointers

- C treats pointers to functions just like pointers to data therefore we can have arrays of pointers to functions
- This offers the possibility to select a function using an index

```
void (*file_cmd[]) (void) =
{    new_cmd,
     open_cmd,
     close_cmd,
     save_cmd ,
     save_as_cmd,
     print_cmd,
     exit_cmd
};
```

Compare the syntax with

```
int a[]=
{
   1,
   2,
   3
};
```

If the user selects a command between 0 and 6, then we can subscript the file_cmd array to find out which function to call

```
file_cmd[n]();
```

Trigonometric Functions

```
// prints tables showing the values of cos,sin
#include <math.h>
#include <stdio.h>
void tabulate(double (*f)(double), double first, double last, double incr);
main()
{
    double final, increment, initial;

    printf (“Enter initial value: “);
    scanf (“%lf”, &initial);

    printf (“Enter final value: “);
    scanf (%lf”, &final);

    printf (“Enter increment : “);
    scanf (%lf”, &increment);

    Printf(“\n    x   cos(x) \n”
            “  ----------  ----------\n”);
    tabulate(cos, initial,final,increment);

    Printf(“\n     x    sin (x) \n”
            “  ----------  ----------\n”);
    tabulate(sin, initial,final,increment);

    return 0;
}
```

Trigonometric Functions

```
// when passed a pointer f prints a table showing the value of f
void tabulate(double (*f) (double), double first, double last, double
                        incr)
{
    double x;
    int i, num_intervals;
    num_intervals = ceil ( (last -first) /incr );
    for (i=0; i<=num_intervals; i++){
            x= first +i * incr;
            printf("%10.5f %10.5f\n", x , (*f) (x));
}
}
```

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1

X             cos(x)
----  --------  ----------
0.00000   1.00000
0.10000   0.99500
0.20000   0.98007
0.30000   0.95534
0.40000   0.92106
0.50000   0.87758

X             sin(x)
----  --------  ----------
0.00000   0.00000
0.10000   0.09983
0.20000   0.19867
0.30000   0.29552
0.40000   0.38942
0.50000   0.47943
```

Sorting

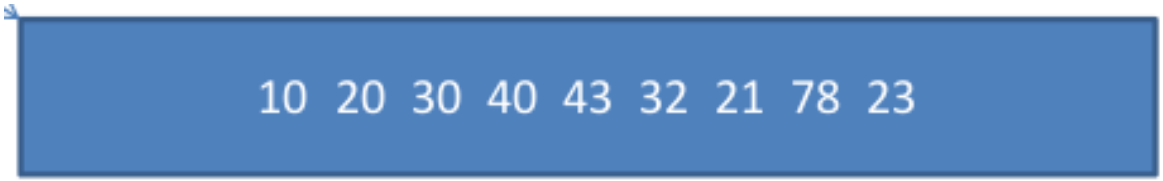
- **qsort** will sort an array of elements. This is a wild function that uses a pointer to another function that performs the required comparisons.
- Library: stdlib.h Prototype:

void qsort (void *base , size_t num , size_t size , int (*comp_func) (const void *, const void *))

-
- base is a pointer to the array to be sorted. This can be a pointer to any data type
- num The number of elements.
- size The element size.
- int (*comp_func)(const void *, const void *))
  - This is a pointer to a function.

Using qsort

- A



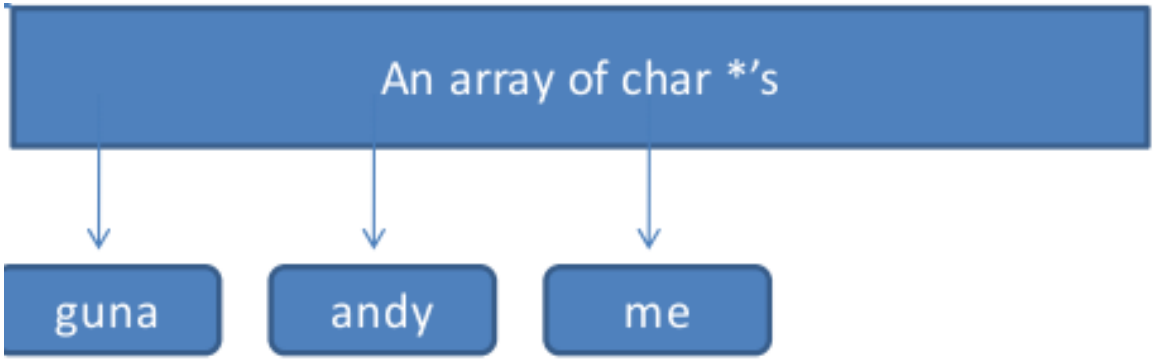
- A is an array of integers. Sort it using qsort with natural integer order
- Write the compare function:
- int intcomp(const void* a, const void* b);

```c
#include <stdio.h>
#include <stdlib.h>

int compare(const void* a, const void* b)
{
    return *( (int*)a )- *( (int*)b );
}
int main()
{
    int i;
    int a[]={5,4,3,2,1};
    qsort( a, 5, sizeof(int), compare);
    for(i=0; i<5; i++) printf("%d\n", a[i]);
}
```

Using qsort

- A



- Write the compare function to sort by alphabetical order
-
- int strcomp(const void* a, const void* b)
- { return strcmp((const char*)a, (const char*)b); }
-

Sorting

- qsort thus maintains it's data type independence by giving the comparison responsibility to the user.
- The compare function must return integer values according to the comparison result:
  - less than zero : if first value is less than the second value
  - zero : if first value is equal to the second value
  - greater than zero : if first value is greater than the second value
- The generic pointer type void * is used for the pointer arguments, any pointer can be cast to void * and back again without loss of information.

Conclusions

- Function pointers can be passed as arguments to other functions or return from functions
- Unfortunately, their cumbersome syntax baffles both novices and experienced programmers.
  - → C++ reduces the necessity of function pointers with classes and virtual functions