Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 14

- Function Pointers
- Dynamic Memory Allocation
- Part 3 – Object Oriented Programming
  C++ Classes

Function Pointers

```
// pointer to function without parameter returning int
int (*pf)(void);

// pointer to function with 2 int params returning nothing
void (*pf)(int, int);
```

- In C/C++ there is no function data type
- But it is possible to declare pointers to functions that point to the first byte of the code
- These pointers can be used to call functions
- They also can be stored like any other types (e.g. in arrays) or used as parameters
- Declaration: Like function declaration. Pointer name prefixed by * and enclosed in ( )

Calling Functions Via Pointers

Syntax: `(*<function-pointer>)(<parameters>)`

Semantics:

- Evaluate parameter expressions
- Push values on stack
- Call function the pointer is pointing to
- Return the value to the calling environment

```
int foo(int x) { return x; }
int (*pf)(int);

pf = foo;
cout << (*pf)(10);  // calls foo with argument 10
```


Function Pointer Example (1)

```
#include <iostream>
// pointer to function mapping (int,int) to int
typedef int (*BinIntOp)(int, int);

int plus (int x, int y) { return x+y; }
int minus(int x, int y) { return x-y; }
int mult (int x, int y) { return x*y; }
int divi (int x, int y) { return x/y; }

int main()
{
  // f stores 4 function pointers
  BinIntOp f[] = { plus, minus, mult, divi };

  for (int i=0; i < sizeof(f)/sizeof(f[0]); ++i)
    std::cout << (*f[i])(7,3) << " ";
  return 0;
}
// Output: 10 4 21 2
```

Function Pointer Example (2)

- Library function qsort ("Quicksort")
- Generic sorting routine
- Average time complexity $C \cdot n \cdot \log n$
- Worst case time complexity $C' \cdot n^2$
- man qsort:

```
#include <cstdlib>

void qsort(void *base, size_t nmemb, size_t size,
       int(*compar)(const void *, const void *));
```

Huh?


```
void qsort(
  void *base,
  size_t nmemb,
  size_t size,
  int (*compar)(const void *, const void *)
);
```

`void *` : Generic pointer type. Variables of all pointer types can be assigned to void * pointers without cast

`size_t`	size type (usually unsigned int)
`base`	start address of array to be sorted
`nmemb`	number of elements
`size`	size of an element (in bytes)
`compar`	function that compares two elements


```
#include <cstdlib>
#include <iostream>
// a points to a char pointer, so does b
// returns 0 if strings *a and *b are equal
// return <0 if string *a < string *b, >0 otherwise

int my_strcmp(const void *a, const void *b) {
  return strcmp(*(char**)a, *(char **)b);
}

void sort_strings(char *A[], int n) {
  qsort(A, n, sizeof(A[0]), my_strcmp);
}

int main() {
  char *A[] = { "b", "c", "ccc", "a" }; // array of pointers
  const int N = sizeof(A)/sizeof(A[0]);
  sort_strings(A, N);
  for (int i=0; i < N; ++i) std::cout << A[i] << " ";
}
```


Dynamic Memory Allocation: new and delete

- Local variables and functions parameters are located on the stack (LIFO data structure)
- Dynamic memory is allocated from a different part of memory called heap
- Operator `new` dynamically allocates memory
- Operator `delete` is used to release it when no longer needed – can be done later, even in a different function
- As always, YOU are in control because the compiler cannot know when memory is no longer needed and can be deleted
- C/C++ does not have a garbage collector

Operator new

```
int *p = new int;       // allocates space
                        // for an int
                        // p now points to it

if (!p) { cerr << "out of memory" << endl; exit(1); }

*p = 0;                 // use allocated memory
```

- Syntax: new <type>
- Allocates space for a variable of type <type> on the heap and returns a pointer to it
- No initialization if <type> is a basic C – plain old data ("POD") – type
- Calls C++ class constructor (later)
- If no memory is available new returns 0

Operator delete (1)

```
int *p = new int;

...

// free memory when *p is no longer used

delete p;
p = 0;  // safeguard
```


Operator delete (2)

- Frees memory when it is no longer used
- Calls class destructor for non-PODs (later)
- Syntax: delete <pointer-to-allocated-mem>
- Good practice: set pointer to 0 after delete to prevent further access of this address through this pointer
- Also: make sure each heap object has exactly one owner who is responsible for its deletion
- 0 special pointer value: can be assigned to any pointer variable regardless of type
- 0 not part of process memory. Can indicate no memory, invalid pointer, no successor, etc.

Dynamic Arrays

```
float *p = new float[100];
if (!p) { exit(1); } // out of memory
...
for (int i=0; i < 100; ++i) p[i] = 0.0;
...
// free memory when *p is no longer used
delete [] p;
p = 0;  // safeguard
```

- Syntax: new <type>[<num-of-elements>]
- Allocates an array of elements of type <type>
- Elements are not initialized for basic C types
- When no longer used free memory with delete [ ] <pointer-to-dynamic-array>

new/delete Match

- new/delete come in pairs: for every new there should be a delete in your program
- More specifically:
  - For every new at least one corresponding delete
  - For every new[] at least one corresponding delete[]
- If mixed, result of computation is undefined!

Speed / Memory Issues

- Allocating dynamic memory is SLOW
- Program has to maintain list of available memory blocks
- If speed is important try to minimize new/delete. E.g. by reusing arrays
- new allocates more memory than you think (overhead usually 4 or 8 bytes per call, getting smaller with better implementations)
- Allocating arrays is therefore more efficient than single variables
- You can roll your own memory allocation by overloading the new/delete operators (later)

Memory Allocation in C

```
float *p = (float*) malloc(100*sizeof(float));
if (!p) { exit(1); } // out of memory
...
for (int i=0; i < 100; ++i) p[i] = 0.0;
...
// free memory when *p is no longer used
free(p);
p = 0;  // safeguard
```

- There are no new/delete operators in C
- Use library function calls

  `void *malloc(size_t n);` : allocates n bytes

  `void free(void *p);` : releases memory
- To learn about them: man malloc

Abstract Data Types in C

C-structs can only have data members

Global functions act on structs; usually pointer to struct as first argument

Abstract Data Types = struct + global functions

```
struct Foo
{
  ...
};

void Foo_init(struct Foo *p);
bool Foo_write(struct Foo *p, FILE *fp);
bool Foo_read(struct Foo *p, FILE *fp);
...
```

Part 3: Object Oriented Programming

C-structs vs. C++ classes

- Structures are special cases of classes
- Structures don't impose any overhead
- Structures are not initialized
- Manual structure clean-up when no longer needed

C++ Classes

Provide additional functionality (some introduce run-time overhead):

- Member functions
- Automatic initialization, destruction
- Access restrictions
- Separation of interface and implementation
- Public inheritance (modeling is-a relationship)
- Multiple inheritance

Classes are also called "objects" = data + member functions

Object-oriented programming makes it possible to maintain large software projects