Week 4 Lecture 1

NWEN 241
Systems Programming

Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

- **More on Pointers**
- **Storage Classes**
- **C Process Layout**

Clarification on Pointer & Arrays

- Consider:

```
int arr[10] = {1, 2, 3};
```

- Since arrays decay to fixed pointer:
  - `arr` is a (fixed) pointer
  - `arr` is the address of the array
  - `&arr` is also the address of the array
- Hence, to let a pointer p point to z, we can write in 3 ways:

```
int *p;
p = arr;
```

```
int *p;
p = &arr;
```

```
int *p;
p = &arr[0];
```

More on Pointers

Recap: Usage of Pointers

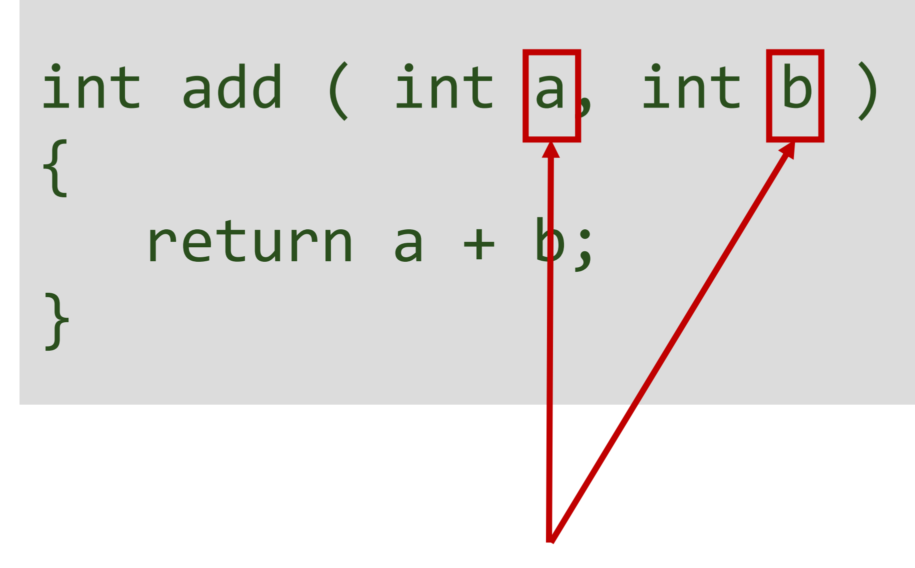
1) Provide an alternative means of accessing information stored in arrays

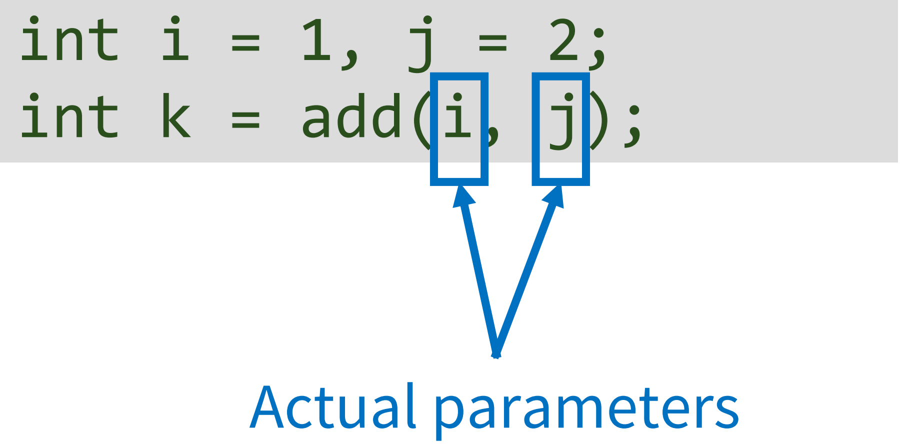
2) Provide an alternative (and more efficient) means of passing parameters to functions

3) Enable dynamic data structures, that are built up from blocks of memory allocated from the heap at run time

Passing Function Parameters

- Recall:





Call by Value

```
int add ( int a, int b )
{
    return a + b;
}
```

Formal parameters

```
int i = 1, j = 2;
int k = add(i, j);
```

Actual parameters

- The values of **actual parameters** (i, j) are copied to **formal parameters** (a, b)
  - Actual and formal parameters are separate entities
- What happens thereafter to formal parameters has no effect on actual parameters
  - Any changes on a, b will not be transferred back to i, j

Example

```
void swap( int a, int b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

```
int i = 5;
int j = 10;
swap( i, j );
printf("%d %d", i, j);
```

Output

```
5 10
```

- After call to `swap()`, `i` and `j` values remain unchanged
- During execution of `swap()`, the copies of `i` and `j` are swapped inside the function, but not `i` and `j` themselves!

The Solution: Call by Reference

- Pass a copy of the address to the function
- **Both formal and actual parameters refer to the same address**
- This can be done using pointers as function parameters

```
void swap( int *a, int *b )
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int i = 5;
int j = 10;
swap( &i, &j );
printf("%d %d", i, j);
```

Output

```
10 5
```

Call by Reference for Efficiency

- Recall: When a structure variable is passed as an input argument to a function, all its component values are **copied** into the local structure variable
- Passing entire copy of a structure can be inefficient, especially for large structs
- **For efficiency, pass a copy of the address of structure to function**
- **This can be done using pointer to struct as function parameter**

```
typedef struct student_info {
        char name[40];
        int student_id;
        int age;
} StudentInfo;

void print_student(StudentInfo s)
{
        printf("Name: %s\n", s.name);
        printf("Student ID: %d\n", s.id);
        printf("Age: %d\n", s.age);
}

…

StudentInfo s1 = {"John", 12345, 20};
print_student(s1);
```

Call by Reference for Efficiency

```
typedef struct student_info {
	char name[40];
	int student_id;
	int age;
} StudentInfo;

void print_student(StudentInfo *s)
{
	printf("Name: %s\n", s->name);
	printf("Student ID: %d\n", s->id);
	printf("Age: %d\n", s->age);
}

…

StudentInfo s1 = {"John", 12345, 20};
print_student(&s1);
```

Copy of address of s1 is passed instead of a copy of the entire structure s1

Call by Reference: Placing Restrictions

```
…

void print_student(StudentInfo *s)
{
    printf("Name: %s\n", s->name);
    printf("Student ID: %d\n", s->id);
    printf("Age: %d\n", s->age);

    s->age = 1000;
}


…
```

- `print_student()` can actually modify the value of s
- How to restrict function from modifying parameters passed by reference?
- Add `const` modifier to parameter

Call by Reference: Placing Restrictions

```
…

void print_student(const StudentInfo *s)
{
    printf("Name: %s\n", s->name);
    printf("Student ID: %d\n", s->id);
    printf("Age: %d\n", s->age);

    s->age = 1000; // compiler will not
                   // allow this
}

…
```

- `print_student()` can actually modify the value of s
- How to restrict function from modifying parameters passed by reference?
- Add `const` modifier to parameter

Function Returning a Pointer

- Functions can return a pointer
- **Make sure that returned pointer points to a valid memory location**

```
float *find_max(float A[], int N)
{
   int i;
   float *the_max = &(A[0]);
   for (i = 1; i < N; i++)
      if (A[i] > *the_max) the_max = &(A[i]);
   return the_max;
}

int main(void)
{
   float scores[5] = {10.0, 8.0, 5.5, 2.0, 4.1};
   float *max_score;

   max_score = find_max(scores, 5);
   printf("%.1f\n",*max_score);
   return 0;
}
```

Storage Classes

Variable Lifetime and Scope

Variables have lifetime*: it begins when the variable is allocated memory and ends when the memory is “de-allocated”

Variables have scope: these are parts of the program where a variable is visible

*Lifetime is also known as **storage duration**

Lifetime

- Static
  - A static variable exists for the entire program execution duration; allocated memory when program starts
- Automatic
  - An automatic variable exists within a block that contains it; allocated memory when execution enters the block and de-allocated when execution leaves the block
- Dynamic
  - A dynamic variable exists from allocation to de-allocation of memory; allocation and de-allocation are done explicitly through dynamic memory allocation function calls

Lifetime

Program start

Program end

Static

Block start

Block end

Automatic

Explicit allocation

Explicit deallocation

Dynamic

Scope

- Local
  - A local variable is only visible inside the current, innermost block
- Global
  - A global variable is visible in the *whole* compilation unit, from the line of declaration to the end of file
- External
  - An external variable is visible in *all* compilation units

Local Scope

file1.c

```
{
    variable declaration



}
```

file2.c

```
{
    {
        variable declaration


    }
}
```

Global Scope

`file1.c`

```
variable declaration
```

`file2.c`

```
variable declaration
```

External Scope

`file1.c` `file2.c`

```
variable declaration
```

Storage Classes at a Glance

C storage class	Declaration	Default init value	Init frequency	Stored in	Scope	Lifetime
`auto`	Inside block	Garbage	Every time block is entered	Memory	Local	Automatic
`static`	Inside block	0	Once at program start	Memory	Local	Static
	Outside any block	0	Once at program start	Memory	Global	Static
`extern`	Outside any block	0	Once at program start	Memory	External	Static
`register`	Inside block	Garbage	Every time block is entered	Maybe in register	Local	Automatic

auto Storage Class Declaration

```
{
    auto double x; /* Same as: double x */
    int num;       /* Same as: auto int num; */
    ...
}
```

- **auto** is the default storage class for a variable defined inside a function body or a statement block
- **auto** prefix is optional; *i.e.*, any locally declared variable is automatically **auto**, unless specifically defined to be static

auto Storage Class Example (Scope)

```
int func(float a, int b)
{
  int i;
  double g;

  for (i = 0; i < b; i++) {
    double h = i*g;
    // loop body - may access a, b, i, g, h
  } // end of for-loop
  // func body - may access a, b, i, g
} //  end of func()
```

i is visible from this point to end of **func**

g is visible from this point to end of **func**

h is only visible from this point to end of loop!

auto Storage Class Example (Lifetime)

```
int func(float a, int b)
{
  int i;
  double g;

  for (i = 0; i < b; i++) {
    double h = i*g;
    // loop body - may access a, b, i, g, h
  } // end of for-loop
  // func body - may access a, b, i, g
} // end of func()
```

Storage for **i** allocated

Storage for **g** allocated

Storage for **h** allocated

Storage for **h** released

Storage for **i** released

Storage for **g** released

`static` Storage Class Declaration

- The **static** prefix *must* be included

```
{

   static double local_static;
   …

}
```

```
void func1(int a)
{
    …
}

static int global_static;

void func2(int b)
{
    …
}
```

static Storage Class Example

```
#include <stdio.h>
void strange( int x )
{
        static int y;
        if ( x == 0 )
                printf( "%d\n", y );
        else if ( x == 1 )
                y = 100;
        else if ( x == 2 )
                y++;
}
int main (void)
{
        strange(1);  /* Set y in strange to 100 */
        strange(0);  /* Will display 100        */
        strange(2);  /* Increment y in strange  */
        strange(0);  /* Will display 101        */
        return 0;
}
```

Program output

```
$ gcc -o strange strange.c
$ ./strange
100
101
$
```

`extern` Storage Class Declaration

- `extern` is the default storage class for a variable defined *outside* any function's body

```
void func1(int a)
{
    …
}

int global;

void func2(int b)
{
    …
}
```

`extern` Storage Class Example 1

```
#include <stdio.h>

float x = 1.5;

void show (void)
{
        printf("%f\n", x); /* Access external x */
}

int main (void)
{
        printf("%f\n", x); /* Access external */
        show();
        return 0;
}
```

`extern` Storage Class Example 2a

What if x is defined after `main` and you want to use it in `main`?

```
#include <stdio.h>

extern float x;

void show (void)
{
        printf("%f\n", x); /* Access external x */
}

int main (void)
{
        printf("%f\n", x); /* Access external x */
        show();
        return 0;
}

float x = 1.5;
```

extern Storage Class Example 2b

What if x is defined in another source file?

```
#include <stdio.h>

void show (void);

int main (void)
{
    printf("%f\n", x); /* Access external x */
    show();
    return 0;
}

float x = 1.5;
```

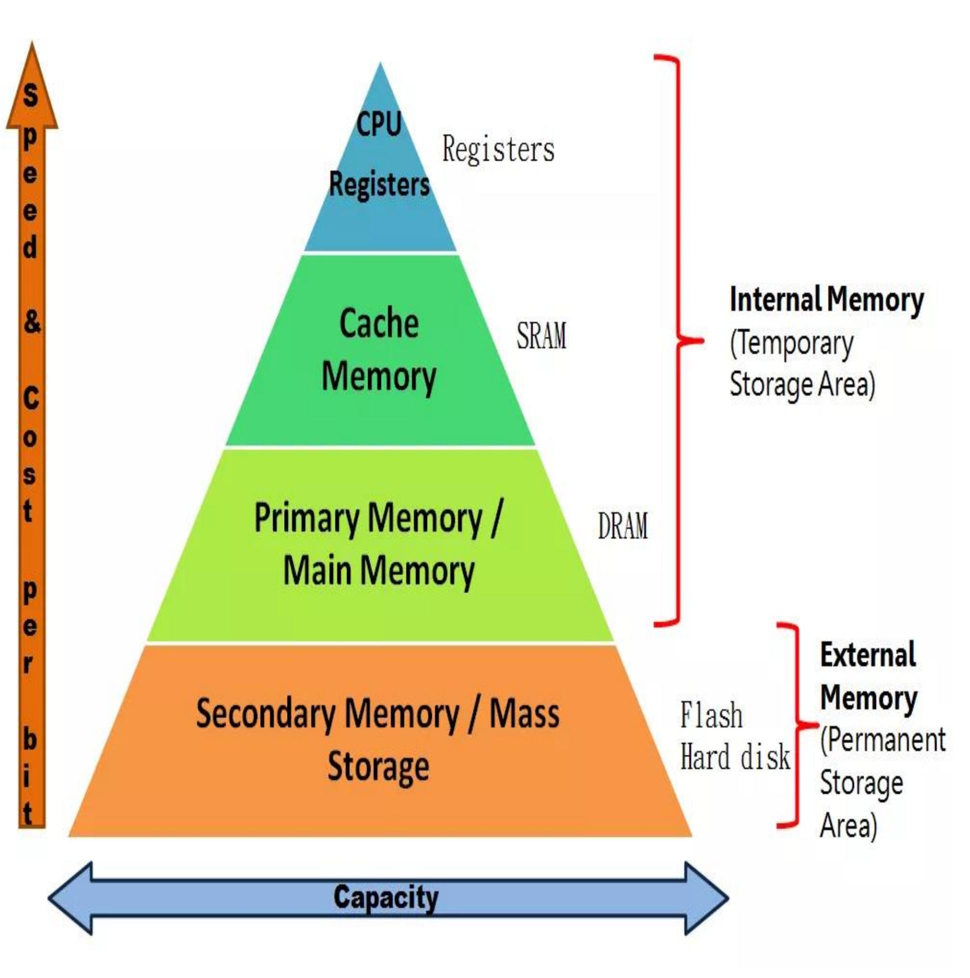
```
#include <stdio.h>

extern float x;

void show (void)
{
    printf("%f\n", x); /* Access external x */

}
```

`register` storage class



- The fastest storage resides within the CPU itself in high-speed memory cells called *registers*
- The programmer can *request* the compiler to use a CPU register for storage

- Example:

```
register int k;
```

`register` storage class

- The compiler can ignore the request, in which case the storage class defaults to `auto`

- A register variable is local to the block which contains it

C Process Layout

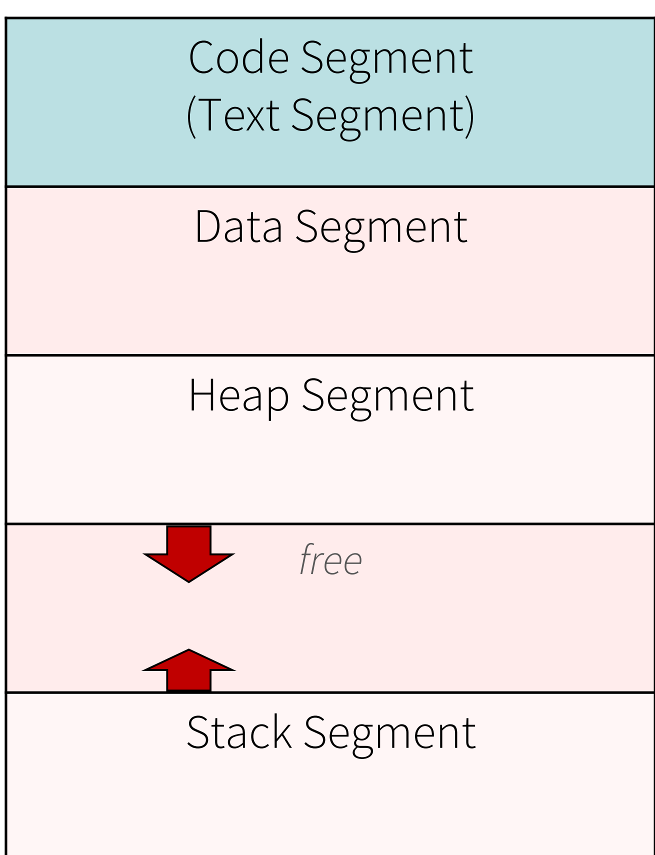
C Process Layout

- Memory space for program code includes space for machine language **code** and **data**
- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs

Code Segment (Text Segment)
Data Segment
Heap Segment
free
Stack Segment

C Process Layout

- Memory space for program code includes space for machine language **code** and **data**
- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs



Storage Layout

- Where are auto, static, and extern variables stored?

Contains the program's machine code	Code Segment (Text Segment)
Contains static data (e.g., **`static`** class, **`extern`** class)	Data Segment
Contains dynamically allocated data – later…	Heap Segment
Unallocated memory that the stack and heap can use	*free*
Contains temporary data (e.g., **`auto`** class)	Stack Segment

Next Lecture

- Dynamic Memory Allocation