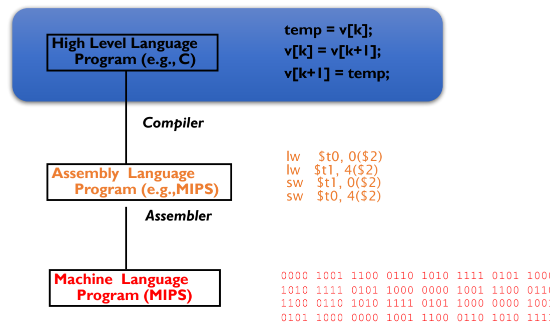
Lecture 2: Introduction to C

(CPEG323: Intro. to Computer System Engineering)

Levels of Program Code



The Lecture Plan

- You will not learn how to fully code in C in these lectures.
- We will only review a few key C concepts
  - Pointers
  - Arrays
  - Memory management

Compilation: Overview

- C *compilers* map C programs into architecture-specific machine code (string of 1s and 0s)
  - compiling .c files to .o files, then linking the .o files into executables; Assembling is also done (but is hidden, i.e., done automatically, by default)
- Advantages:
  - Excellent run-time performance
  - Allow us to exploit underlying features of the architecture
- Disadvantages:
  - Compiled files are architecture-specific, depending on CPU type and the operating system
  - Executable must be rebuilt on each new system
    - i.e., "porting your code" to a new architecture
  - "change→ compile→ run" iteration cycle can be slow, during the development cycle.

Actual C Code

```
#include <stdio.h>
#define REPEAT 5

int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < REPEAT; i = i + 1) {
                printf("hello, world\n");
    }
    return 0;
}
```

Typed Variables in C

```
int   x = 2;
float y = 1.618;
char  z = 'A';
```

Type	Description
int	signed integer
char	single text character or symbol
float	floating point non-integer number

- More about the integers:
  - The size of an integer is machine dependent! It is typically 4 bytes, but cannot assume it.
  - An integer can be either signed or unsigned (e.g., unsigned int).

C Syntax : Operators

- Arithmetic: +, -, *, /, %
- Assginment: =
- Bitwise
  - Logic: ~, &, |, ^
  - Shifts: <<,   >>
- Boolean logic: !, &&, ||
- Equality testing: ==, !=
- Order relations: <, <=, >, >=
- …

7

C Syntax : Control Flow

- if-else
  - if (expression) statement
  - if (expression) statement1
    else statement2
- while
  - while (expression)
    statement
  - do
    statement
    while (expression);
- for
  - for (initialize; check; update) statement
- switch
  - switch (expression){
    case const1:   statements
    case const2:   statements
    default: statements
    }
  - break

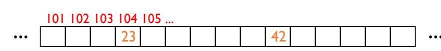
8

Pointers

9

Address vs. Value

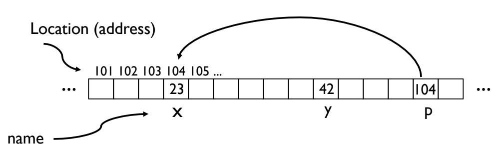
- Memory can be considered as a single huge array:
  - Each cell of the array has an address associated with it.
  - Each cell also stores some value.
- Don't confuse the address referring to a memory location with the value stored in that location.



10

Pointers

- A pointer is a variable that contains an address.
  - An *address* refers to a particular memory location, usually also associated with a variable name.



11

Pointer Operations

- int *x;
  - Declares variable x as the address of an integer.
- x= &y;
  - Assigns address of y to x.
- z= *x;
  - Assigns the value at address in x to z

12

Pointer Examples

```
int *p,x,y  p [ ? ]  x [ ? ]  y [ ? ]
```


Example

```
int x=1, y=2, z[10];
int *ip;

ip = &x;
 y = *ip;
*ip = 0;
ip = &z[0];
```


C pass parameters “by value”!

```
void AddOne (int x) {
    x = x + 1;
 }
 int y = 3;
 AddOne(y);
```

Y remains equal to 3

Function AddOne gets a copy of the parameter, so changing the copy cannot change the original!

How to get a function to change a value?

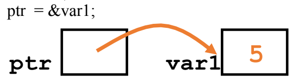
- Pass a pointer!
  - Function accepts a pointer and then modifies value by dereferencing it.

```
void AddOne (int *p) {
  *p = *p + 1;
}
int y = 3;

AddOne(&y);
```

y is now equal to 4

Pointer Declaration and Allocation

- Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to!
- How to make it point to something meaningful?
  - Make it point to something that already exists
    int *ptr, var1;
    var1 = 5;
    ptr = &var1;



  - Allocate room in memory for something new that it will point to. (e.g., malloc)

An Example of buggy code

```
int *p;
*p = 5;
printf("%d\n",*p);
```

What is the result from executing this code?

- Prints 5
- Prints garbage
- Always crashes.
- Almost always crashes.

An Example of buggy code (Cont.)

```
int *p;
*p = 5;
printf("%d\n",*p);
```

What is the result from executing this code?

- Prints 5
- Prints garbage
- Always crashes.
- Almost always crashes.

Arrays

Basic Concepts of Arrays

- **Declaration:**
  - int arr[2];
  - int arr[] = {795, 635};
- **Accessing elements:**
  - arr[k]: returns the k^{th} element
  - Array size n: access entries 0 to n-1
  - Warning: An array in C does not know its own size, and its bounds are not checked! So, be careful with segmental faults and bus errors.

Pointer vs. Array

- Array variable is a "pointer" to the first (0^{th}) element
- arr[i] is treated as *(arr+i)
  - arr[0] is the same as *arr
  - arr[2] is the same as *(arr+2)
- Here are three equivalent ways to set all array elements to zero.
  - for (i=0; i < size; i++) arr[i] = 0;
  - for (i=0; i < size; i++) *(arr+i) = 0;
  - for (p=a; p < arr + size; p++) *p = 0;

Pointer Arithmetic

- A pointer is a memory address, so we can add to or subtract from it to move through the memory space.

Pointer Arithmetic: An Example

Assume variable a is at address 100.

```
char   *p;
char    a;

 p = &a;
```

```
int   *p;
int    a;

 p = &a;
```

What does the following line yield?
`printf("%u %u\n",p,p+1);`

100 101

Adds `1*sizeof(char)` to the memory address

100 104

Adds `1*sizeof(int)` to the memory address

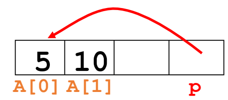
Pointer arithmetic should be used cautiously!!!

Pointer Arithmetic

- A pointer is a memory address, so we can add to or subtract from it to move through in the memory space.
- p+1 means increments p by sizeof(*p)
  - i.e., moves pointer to the next array element.
- What is valid pointer arithmetic?
  - Add an integer to a pointer
  - Subtract 2 pointers (in the same array)
  - Compare pointers (<, <=, ==, !=, >, >=)
  - Compare pointer to NULL (indicates that the pointer points to nothing)
- Everything else is illegal since it makes no sense:
  - Adding two pointers
  - Multiplying pointers
  - Subtract pointer from integer

Question

```
int main(void){
  int A[] = {5,10};
  int *p = A;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
  p =  p + 1;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
  *p = *p + 1;
  printf("%u %d %d %d\n",p,*p,A[0],A[1]);
}
```



If the first printf outputs 100 5 5 10, what will the other two printf output?

1: 101 10 5 10 then 101 11 5 11
2: 104 10 5 10 then 104 11 5 11
3: 101 <other> 5 10 then 101 <3-others>
4: 104 <other> 5 10 then 104 <3-others>
5: One of the two printfs causes an ERROR
6: I surrender!

Question (2)

How many of the following are invalid?

- pointer + integer
- integer + pointer
- pointer + pointer
- pointer – integer
- integer – pointer
- pointer – pointer
- compare pointer to pointer
- compare pointer to integer
- compare pointer to 0
- compare pointer to NULL

#invalid
1
2
3
4
5
6
7
8
9
10

Answer (2)

How many of the following are invalid?

pointer + integer	ptr+1
integer + pointer	1+ptr
pointer + pointer	ptr+ptr
pointer – integer	ptr-1
integer – pointer	1-ptr
pointer – pointer	ptr-ptr
compare pointer to pointer	ptr1==ptr2
compare pointer to integer	ptr==1
compare pointer to 0	ptr==0
compare pointer to NULL	ptr==NULL

#invalid
1
2
3
4
5
6
7
8
9
(1)0

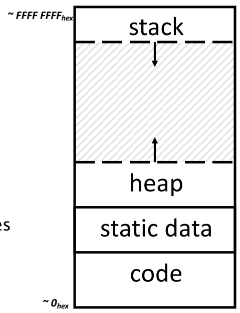
Summary

- All data is in memory
  - Each memory location has an address to use to refer to it and a value stored in it.
- Pointer is a variable whose value is an address
  - Operations:
    - * "follows" a pointer to its value
    - & gets the address of a value
  - Pointers and array variables are very similar.
    - Adding 1 to a pointer moves the pointer by the size of the thing it's pointing to.
  - When the pointers are useful?
    - If we want to pass a large array, it's easier to pass a pointer than the whole array.
    - In general, pointers allow cleaner, more compact code
  - So what are the drawbacks?
    - Pointers are probably the single largest source of bugs in C, so be careful anytime you deal with them, such as dangling references and memory leaks.

Memory Management

C Memory Layout

- There are four components in a program's address space.
  - stack: local variables, grows downward
  - heap: space requested for pointers via `malloc()`, grows upward
  - static data: variables declared outside the main function, does not grow or shrink
  - code: loaded when program starts, does not change.



31

Where are Variables Allocated?

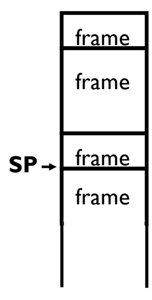
- Declared outside a procedure
  - allocated in "static" storage
- Declared inside procedure
  - allocated on the "stack" and freed when procedure returns
  - main() is treated like a procedure
- Dynamically allocated via malloc:
  - Allocated on the "heap"

```
#include <stdio.h>

int myGlobal;

Int main() {
  int myTemp;
  int *varDyn = malloc(sizeof(int));
}
```

32

Stack

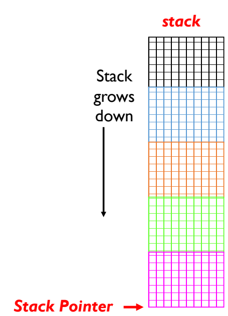
- Stack consists of stack frames, i.e., a contiguous block of memory holding the local variables of a single procedure.
- A stack frame includes:
  - location of caller function (i.e., return address)
  - function arguments
  - local variables
- Stack pointer (SP) tells where the current stack frame is.
- When procedure ends, SP is moved back.



33

Stack – Last In, First Out

```
main ()
{ a(0);
  }
   void a (int m)
     { b(1);
       }
     void b (int n)
       { c(2);
         }
       void c (int o)
         { d(3);
           }
         void d (int p)
           {
            }
```



34

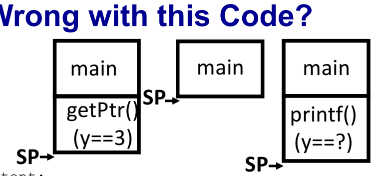
What's Wrong with this Code?

```
int *getPtr() {
 int y;
 y = 3;
 return &y;
};

main () {
 int *stackAddr,content;
 stackAddr = getPtr();
 content = *stackAddr;
 printf("%d", content); /* 3 */
 content = *stackAddr;
 printf("%d", content); /*13451514 */
};
```

35

What's Wrong with this Code?

```
int *getPtr() {
 int y;
 y = 3;
 return &y;
};

main () {
 int *stackAddr,content;
 stackAddr = getPtr();
 content = *stackAddr;
 printf("%d", content); /* 3 */
 content = *stackAddr;
 printf("%d", content); /*13451514 */
};
```



Problem: *printf* overwrites stack frame

Never return pointers to local variable from functions
Your compiler will warn you about this – don't ignore such warnings!

36

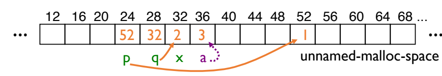
Using the heap - Dynamic Memory Allocation

- malloc(n):
  - Allocate a block of uninitialized memory
  - Most often, malloc used to allocate space for an array of items.
    - int *p = malloc (n*sizeof(int));
    - Allocates space for n integers.
- free(p):
  - Releases memory allocated by malloc()
  - p is pointer containing the address originally returned by malloc()

37

Example

```
void foo() {
 int *p, *q, x, a[1];
 p = (int *) malloc (sizeof(int));
 q = &x;

 *p = 1;
 *q = 2;
 *a = 3;

 printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
 printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
 printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```




```
*p:1, p:52, &p:24
*q:2, q:32, &q:28
*a:3, a:36, &a:36
```

38

Summary

- C has four pools of memory
  - Code
  - Static storage: global variable storage, permanent over entire program run
  - Stack: local variable storage, parameters, return address
  - Heap (dynamic storage): malloc() allocates space from here, free() returns it.
- Common Memory-related Bugs
  - Using uninitialized values
  - Accessing memory beyond your allocated region
  - Mismatched malloc/free pairs

39