Computer Programming

“Learn science first and then continue with the practice born from that science”

Leonardo da Vinci

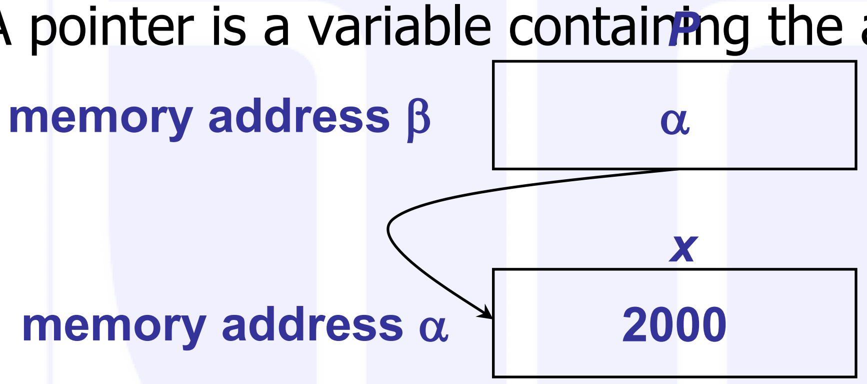
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA
Computer Science

Outline

- Pointers
  - Declaration
  - Operations
- Memory Management
- Pointers and arrays

Pointers

- **Definition:** A pointer is a variable containing the address of another variable.
- Example:



- A pointer is *bound to a type*. E.g. if *x* is of type **int** then pointer *p* is bound to type **int**
- Pointer declaration:

```
type *identifier
```

  - **Note the * preceding the pointer identifier.**
- Example:

```
int *p;
```

- The address of a variable is obtained with the **referencing operator, &**

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Pointers. Example

- Consider the declarations:

```
int x;
int *p;
```

  then

```
p=&x;
```

  causes the assignation of the address of variable *x* to pointer *p*.
  - For the previous example, the value of *p* will be α
- The value stored in the memory area pointed to by *p* is obtained with the unary **dereferencing operator, ***
  - Example: statement **x=y** is equivalent with one of the sequences:

```
p=&x;          or    p=&y
*p=y;                x=*p;
```

Common programming errors

- The (*) notation used in declaring pointers does not distribute:
  - E.g. int *a, b; /* a is of type pointer to int, b is of type int */
- Declaring a pointer does not initialize it
  - E.g. char *a; // just declares a pointer to char
  - Common mistake

```
char *s; // s does not point anywhere useful
scanf("%s", s); // note we did not use &s
```

Pointers. Pointer to **void**

- Example: statement **x++** is equivalent with the sequence:

```
p = &x;
(*p)++;
```

Note that for both examples *p* must be bound to the type of *x* and *y*

- When a pointer must not be bound to any type use:

```
void *identifier;
```

- A pointer to void cannot be dereferenced
- Here if we have

```
int x;
float y;
void * p;
```

then any of `p = &x; p = &y;` is correct.

- Type casts must be used, `(type *)p`, to specify the type to which *p* points
- Referencing/dereferencing example (demoRefDeref)
- Demo: use pointers to uppercase string (cnv2Uppercase)

Pointers. Pointer to **void**

- **Note**: at any given moment the programmer must know the type of value stored where a pointer points to
- Example:

```
int x;
void *p;
```

  then a statement like `x = 10;` is equivalent to the sequence

```
p = &x;
*(int *)p = 10;
```

- If *p* is declared as before, then dereferencing cannot be used without a type cast

Constant pointers

- Constant pointer can be declared as follows:

```
type* const identifiers=value;
```

  - Pointer *identifier* is a constant pointer to the memory area holding a *value* of type *type*
- Example. With the declarations:

```
char* const string1="Character string";
char* string2;
```

  the assignments:

```
*string1='C'; *(string1+1)='h';
*(string1+2)='a';
```

  are correct, but

```
string1=string2;
```

  is wrong.
- Demo: attempt to change const pointer to non-const data (const2Non)

Constant pointers

- A pointer to a constant value can be defined as:

```
    type const *identifier=value;
```

  or

```
    const type *identifier=value;
```

- Example. Consider the declarations:

```
    char const *string1="Character string";
    char *string2;
```

  Correct assignment: `string2=string1;`

  Wrong:

```
    *string1='C'; *(string1+1)='h';
    *(string1+2)='a';
```

- You can change the string referred by `string1`:

```
    string2=string1;
    *string2='s';
    *(string2+1)='t';
    *(string2+2)='r';
```

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Constant pointers

- Note the difference between
  - constant pointer: type* const identifier;
  - pointer to constant: type const *identifier;
- A formal parameter declared as:

```
const type *formal_parameter
```

will prevent changing the values stored in the memory area referred by the corresponding actual parameter

Pointers and arrays

- *An array name has as a value the address of its first element – an array name is a constant pointer to its first element and cannot be changed during execution*
- Example:

```
int arr1[100], arr2[100];
int *p;
int x;
...
p=arr1; /* p will hold the address of
arr1[0] */
arr2=arr1; /* WRONG */
```

  - In the example above the assignment:
    `x=arr1[0]` is equivalent to `x=*p;`
    `x=arr1[10]` is equivalent to `x=*(p+10); // eleventh element`

Operations with pointers

- Increment/decrement by 1
  - typically use operators ++ and – –
  - Example:

```
 int arr[100];
 int *p;
 ...
 p=&arr[10];
 p++; /* p points to arr[11]. Its
value is incremented by the size of
an int */
```

Operations with pointers

- Add/subtract an integer to/from a pointer
  - Operation *p±n* results in the increase/decrease of the value of *p* with *n × the number of bytes needed to store a data item of the type of p*
- Example:

```
int arr[100];
int x, i;
...
x=arr[i];
/* equivalent to */
x=*(arr+i);
```

Operations with pointers

- Pointer *difference*
  - If pointer *p* and *q* point to the elements *i* and *j* of a table, i.e. *p=&arr[i]* and *q=&arr[j]* and, *j>i* then *q–p=j–i*
  - In general, if the operation is legal, then the difference is the number of bytes between the two addresses
- Pointer *comparison*
  - Two pointers to the same array can be compared using the relation and equality operators < <= > >= == !=

Operations with pointers. Example

```
#include <stdio.h>

void Max_min1(int n, int a[], int *max, int* min)
{
   int i;
   *max=a[0];
   *min=a[0];
   for (i=1; i<n; i++)
   {
      if (a[i] >*max) *max=a[i];
      else if (a[i] < *min) *min=a[i];
   }
}
void Max_min2(int n, int *a, int *max, int *min)
{
   int i;
   *max=*a;
   *min=*a;
   for (i=1; i<n; i++)
   {
      if (*(a+i) >*max) *max=*(a+i);
      else if (*(a+i) < *min) *min=*(a+i);
    }
 }
```

Operations with pointers. Example (cont'd)

```
int main()
{
   int i, n, maximum, minimum, x[100];

   /* Data input */
   printf("\nMaximum and minimum element of integer array x.\nArray size
   is:");
   scanf("%d", &n); // should validate n
   for (i = 0; i < n; i++)
   {
       printf("\nx[%d]=", i);
       scanf("%d", &x[i]);
   }
   /* Call of the first procedure */
   Max_min1(n, x, &maximum, &minimum);
   printf("Max_min1: maximum=%d and minimum=%d\n", maximum, minimum);
   /* Call of the second procedure */
   Max_min2(n, x, &maximum, &minimum);
   printf("Max_min2: maximum=%d and minimum=%d\n", maximum, minimum);
   return 0;
}
```

Bad Pointer Examples

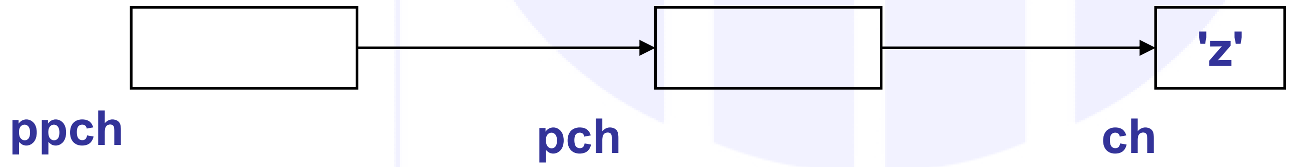
```
void BadPointer()
{
  int* p; // allocate the pointer, but not the pointee
  *p = 42; // this dereference is a serious runtime error
}
```

- Note that pointers **look** like simple variables but they require the extra initialization before use.

```
int array[10]; /* An array for our data */
int main() {
   int *data_ptr; /* Pointer to the data */
   int value; /* A data value */
   data_ptr = &array[0];/* Point to the first element */
   value = *data_ptr++; /* Get element #0, data_ptr points to
   element #1 */
   value = *++data_ptr; /* Get element #2, data_ptr points to
   element #2 */
   value = ++*data_ptr; /* Increment element #2, return its value */
   /* Leave data_ptr alone */
```

Pointers to Pointers

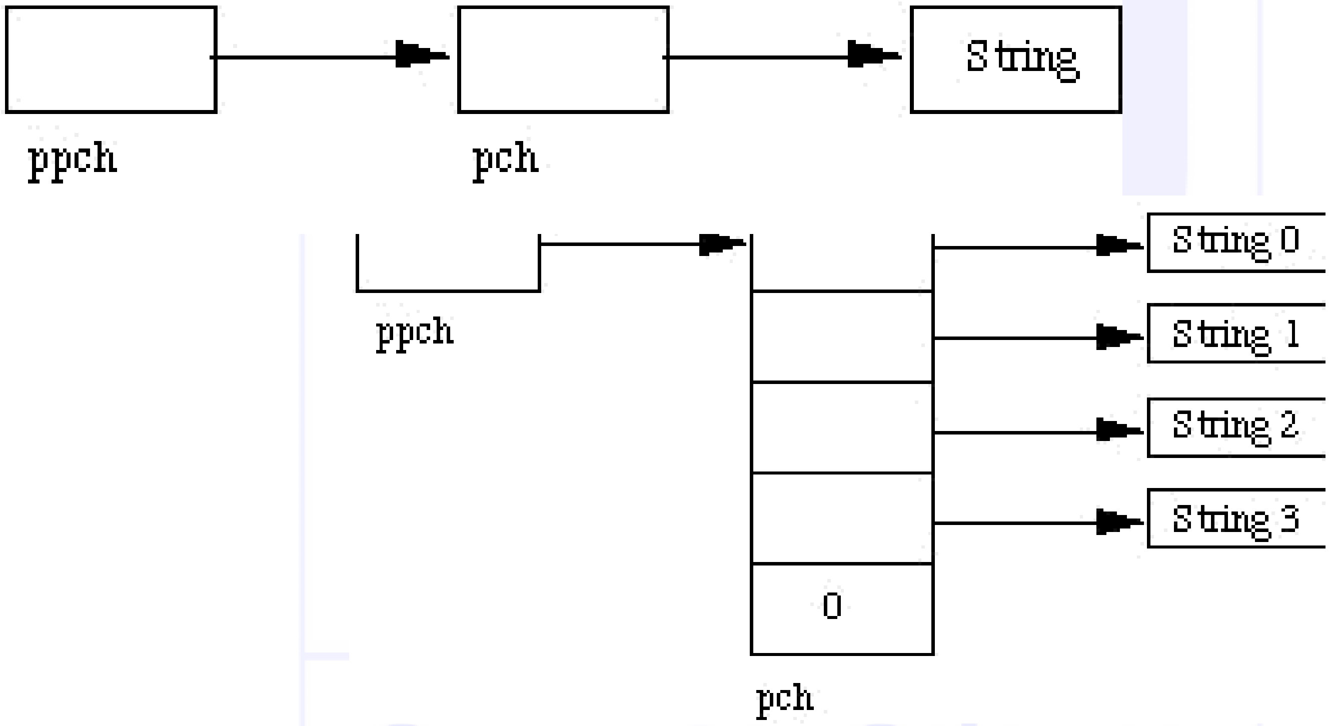
```
char ch = 'z'; // a character
char *pch; // a pointer to a character
char **ppch; // a pointer to a pointer to a character
pch = &ch; ppch = &pch;
```



Recall that `char *` refers to a null terminated string

TECHNICAL UNIVERSITY OF CLUJ-NAPOCA Computer Science

Pointer to String



- We can refer to individual strings by `ppch[0]`, `ppch[1]`, ... Thus this is identical to declaring `char *ppch[]`.

Pointers used as strings

- A normal string in C is nothing but a chunk of memory containing the characters as bytes and ending with a byte of value zero.
- The thing which is used as a string variable is simply a pointer to the first of those bytes.
- Problems with C strings:
  - The string cannot contain the ASCII zero character as that would look like a premature string end.
  - One cannot simply concatenate strings or even copy from one string to another (using '=' would instead make the two pointers point to the same memory rather than a duplicate) without writing routines to process the strings a character at a time or using library functions.
  - It is easy to forget the maximum length one has allocated to a string or forget to keep a zero byte on the end allowing the program to run off the end of the string into arbitrary memory.

Pointer Rules Summary

- A pointer stores a reference to its pointee. The pointee, in turn, stores something useful.
- The dereference operation on a pointer accesses its pointee. A pointer may only be dereferenced after it has been assigned to refer to a pointee. Most pointer bugs involve violating this one rule.
- Allocating a pointer does not automatically assign it to refer to a pointee.
- Assigning the pointer to refer to a specific pointee is a separate operation which is easy to forget.
- Assignment between two pointers makes them refer to the same pointee which introduces sharing.

Useful Info on Pointers

- Ted Jensen's pointer tutorial
  - http://pw1.netcom.com/~tjensen/ptr/pointers.htm
- Practical C programming Chapter
  - http://www.oreilly.com/catalog/pcp3/chapter/ch13.html
- Pointers and memory
  - http://cslibrary.stanford.edu/102/PointersAndMemory.pdf
- Advanced pointer topics
  - http://www.cs.cf.ac.uk/Dave/C/node12.html
- C Pointers and Memory Allocation
  - http://www.openismus.com/documents/cplusplus/cpointers.shtml

Memory allocation/deallocation

- A **heap** is a predefined area of memory which can be accessed by the program to store data and variables.
- The data and variables are allocated on the heap by the system as calls to **malloc()** are made. The system keeps track of where the data is stored.
- Data and variables can be deallocated as desired, leading to holes in the heap. The system knows where the holes are and will use them for additional data storage as more **malloc()** calls are made.
- The structure of the heap is therefore a very dynamic entity, changing constantly.

Memory allocation/deallocation. Heap advantages/disadvantages

- Advantages
  - *Lifetime.* Because the programmer now controls exactly when memory is allocated and deallocated, it is possible to build a data structure in memory, and return that data structure to the caller.
  - *Size*. The size of allocated memory can be controlled with more detail. For example, a string buffer can be allocated at run-time which is exactly the right size to hold a particular string.
- Disadvantages
  - *More Work*. Heap allocation needs to arranged explicitly in the code which is just more work.
  - *More Bugs*. Because it's now done explicitly in the code, realistically on occasion the allocation will be done incorrectly leading to memory bugs.
  - Local memory is constrained, but at least it's never **wrong**.

Memory allocation/deallocation support

- C provides access to the heap features through library functions which any C code can call. The functions are:
- Request a contiguous block of memory of the given size in the heap.
- Prototype: `void* malloc(size_t size);`
  - **malloc()** returns a pointer to the heap block or NULL if the request could not be satisfied.
  - The type **size_t** is essentially an unsigned long which indicates how large a block the caller would like measured in bytes.
  - Because the block pointer returned by **malloc()** is a **void***, a cast is required when storing the void* pointer into a regular typed pointer.

Memory allocation/deallocation support

- The mirror image of **malloc(), free** takes a pointer to a heap block earlier allocated by `malloc()` and returns that block to the heap for re-use.
- Prototype: `void free(void* block);`
  - After the `free()`, the client should not access any part of the block or assume that the block is valid memory.
  - *The block should not be freed a second time*.

Memory allocation/deallocation support

- Take an existing heap block and try to relocate it to a heap block of the given size which may be larger or smaller than the original size of the block.
- Prototype:

```
void* realloc(void* block, size_t size);
```

  - Returns a pointer to the new block, or NULL if the relocation was unsuccessful. Remember to catch and examine the return value of `realloc()` – it is a common error to continue to use the old block pointer.
  - `realloc()` takes care of moving the bytes from the old block to the new block.
  - `realloc()` exists because it can be implemented using low-level features which make it more efficient than C code the client could write.

Memory allocation/deallocation. Example

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char *str1, *str2;
    /* allocate memory for the first string */
    if ((str1=(char*)malloc(100)==NULL)
    {
          puts("Not enough memory");
          exit(1);
    }
    printf("Input a character string:\n");
    fgets(str1, 100, stdin);
    printf("The character string you input is: %s", str1);
    /* allocate memory for the second string */
    if ((str2=(char*)malloc(100)==NULL)
    {
          puts("Not enough memory");
          exit(2);
    }
    printf("Input another character string:\n");
    fgets(str2, 100, stdin);
    printf("The character string you input is: %s", str2);
    free(str1);
    free(str2);
    return 0;
}
```

Dynamic allocation of arrays

- Demos:
  - allocMat
  - array3D
- All demos are included in a zip file stored on the web together with this lecture

Reading

- King, chapters 11 and 12
- Prata, chapter 10
- Deitel, chapters 7, 12 section 3

Summary

- Pointers
  - Declaration
  - Operations
- Memory Management
- Pointers and arrays