Fundamentals of Computing: Lecture 22

Piyush P Kurur
Office no: 224
Dept. of Comp. Sci. and Engg.
IIT Kanpur

September 18, 2009

Dynamic memory allocation

1. Allocating memory as required.
2. Making programs more flexible, array bounds that are runtime dependent.

Arbitrary sized array

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

int main()
{
  int *a, int n;
  printf("enter the size of the list: ");
  scanf("%d",&n);
  a = (int *) malloc(n * sizeof(int))
  if( a == NULL)
  {
    printf("too bad not enough memory");
    return 1;
  }
  /* a is now a variable sized array */
  sort(a,n);
}
```

General idiom for variable size arrays

```
T *a;
a = (T *) malloc( n * sizeof(T))
if( a == NULL)
{
  /* not enough memory */
}
/* use a */
free(a);
```

General idiom for variable size arrays

```
T *a;
a = (T *) malloc( n * sizeof(T))
if( a == NULL)
{
  /* not enough memory */
}
/* use a */
free(a);
```

- The function malloc allocates the required amount of space,

General idiom for variable size arrays

```
T *a;
a = (T *) malloc( n * sizeof(T))
if( a == NULL)
{
  /* not enough memory */
}
/* use a */
free(a);
```

- The function malloc allocates the required amount of space,
- `malloc` returns the pointer to the allocated memory if possible or `NULL` otherwise,

General idiom for variable size arrays

```
T *a;
a = (T *) malloc( n * sizeof(T))
if( a == NULL)
{
  /* not enough memory */
}
/* use a */
free(a);
```

- The function malloc allocates the required amount of space,
- `malloc` returns the pointer to the allocated memory if possible or `NULL` otherwise,
- The expression sizeof(T) gives the "size" of the type T,

The `sizeof` operator

- `sizeof(T)` is the memory required to store a value of type `T`.

The `sizeof` operator

- `sizeof(T)` is the memory required to store a value of type `T`.
- `sizeof(a)` where a is of type `T` is same as size of `sizeof(T)`,

The `sizeof` operator

- `sizeof(T)` is the memory required to store a value of type `T`.
- `sizeof(a)` where a is of type `T` is same as size of `sizeof(T)`,
- The value of `sizeof` operation is of type `size_t`.

```
#include<stdio.h>

int main()
{
  int a[100];
  int *ptr;
  int b;
  int c[] = {100,200,3,4,5,6};
  printf("sizes of:\n");
  printf("\t a is   %lud\n",sizeof(a));
  printf("\t b is   %lud\n",sizeof(b));
  printf("\t c is   %lud\n",sizeof(c));
  printf("\t ptr is %lud\n", sizeof(ptr));
  return 0;
}
```

The `sizeof` of an array

The size of an array of type `T` and length `n` is `n` times the size of `T`

The `sizeof` of an array

The size of an array of type `T` and length `n` is `n` times the size of `T`

Idiom to find the lenght

```
int a[] = {100,200,3,4,5,6}
int len = sizeof(a)/sizeof(a[0]);
int len = sizeof(a)/sizeof(int);
```

What is the type of the function `malloc`?

What is the type of the function `malloc`?

```
void *malloc(size_t size);
```

What is the type of the function `malloc`?

```
void *malloc(size_t size);
```

Pointer to void

- Any pointer can be cast to a void pointer. eg.

```
int *ptr;
void *p;
p = (void *) ptr;
```

What is the type of the function `malloc`?

```
void *malloc(size_t size);
```

Pointer to void

- Any pointer can be cast to a void pointer. eg.

```
int *ptr;
void *p;
p = (void *) ptr;
```

- If `p` is a `void` pointer which was assigned a pointer to `T` then `p` can be cast back to `T`.

What is the type of the function `malloc`?

```
void *malloc(size_t size);
```

Pointer to void

- Any pointer can be cast to a void pointer. eg.

```
int *ptr;
void *p;
p = (void *) ptr;
```

- If `p` is a `void` pointer which was assigned a pointer to `T` then `p` can be cast back to `T`.
- A void pointer cannot be dereferenced

What is the type of the function `malloc`?

```
void *malloc(size_t size);
```

Pointer to void

- Any pointer can be cast to a void pointer. eg.

```
int *ptr;
void *p;
p = (void *) ptr;
```

- If `p` is a `void` pointer which was assigned a pointer to `T` then `p` can be cast back to `T`.
- A void pointer cannot be dereferenced
- No pointer arithmetic is allowed on void pointer.

Why `void` pointers?

Why `void` pointers?

1. Often one wants generic functions like `malloc` that really does not care of the pointer type that it is manipulating.

Why `void` pointers?

1. Often one wants generic functions like `malloc` that really does not care of the pointer type that it is manipulating.
2. The hardware may not support arbitrary conversion due to alignment restriction.

Why `void` pointers?

1. Often one wants generic functions like `malloc` that really does not care of the pointer type that it is manipulating.
2. The hardware may not support arbitrary conversion due to alignment restriction.