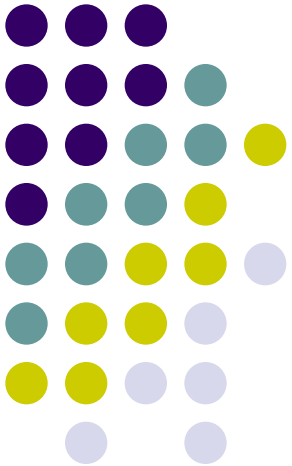
RT - RK

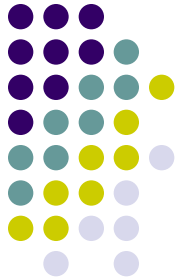
COMPUTER BASED SYSTEMS



Incomplete types in C

RT - RK
COMPUTER BASED SYSTEMS

Incomplete types



- Incomplete type is a type that describes an identifier but do not provide information about identifier size

- All types in C are separated in three groups:
  - function types - functions
  - object types – everything else except when size of object is not known
  - incomplete types

- Incomplete types in C:
  - void type
  - Structure without specified members
  - Union without specified members
  - Array type without specified dimensions

RT - RK
COMPUTER BASED SYSTEMS

VOID type

- Not possible to complete void type
- Type of the result of the function that does not return result – without return statement or return statement without corresponding expression

```
void func1()
{
    ...
    /* return statement without
    expression */
    return
}
```

```
void func2()
{
    ...
    /* no return statement */
}
```

- Only function argument which indicates function that does not take parameters

```
int func2(void);
```

- Not possible to declare object of void type

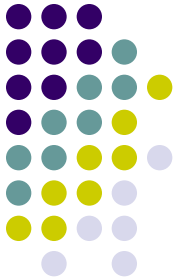
```
void var; /* error */
```

- Pointer variable that points to void type can be declared

```
void *ptr; /* OK */
```

RT - RK
COMPUTER BASED SYSTEMS

struct/union as incomplete types



- Only feature of C that can not be handled without incomplete types is forward reference to structure and unions
- This is necessary when there are two structure declaration that reference each other

```
struct s
{
    ...
    struct q *p; /* at this point struct q is incomplete type */
};

struct q
{
    ...
    struct s *p;
};
```

- In above example compiler is able to generate correct code because struct s contains only pointer to struct q and C standard defines that all pointers to structures have same (during compilation known) size
- This feature can be used for encapsulation – struct q can be declared in some other file and not available in files where pointers to struct q should not be dereferenced

RT - RK
COMPUTER BASED SYSTEMS

Array as incomplete type

- Array declaration without specified dimension
- int array[];
- Used as function parameter declaration

```
int func(array[])
{
    ...
}
```

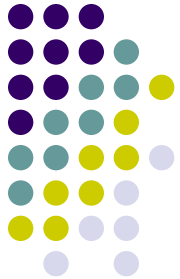
- It should be noted that in this case parameter type is not really incomplete because compiler rewrites the code and replaces it with pointer – which size is known
- To complete incomplete array type same name must be declared within same scope with dimensions specified

```
int array [];

…

int array [21];
```

- It is possible to declare incomplete type in header file and to complete it differently, depending on different needs, in different source files – but it should be used carefully

RT - RK
COMPUTER BASED SYSTEMS

Functions with variable number of arguments 1/2



- Incomplete – because size of actual argument list is not known prior to call
- Example: printf function from standard library

```
printf (const char *format, …);

printf (“Student %s with average grade %f\n”,name, avrg);
```

- This is implemented by introducing new type va_list and corresponding operations on that type
- Operations:
  - **void va_start(va_list ap, parmN)** – initialize argument pointer
  - **type va_arg(va_list ap, type)** – sequentially access arguments
  - **void va_end(va list ap)** – end using argument list

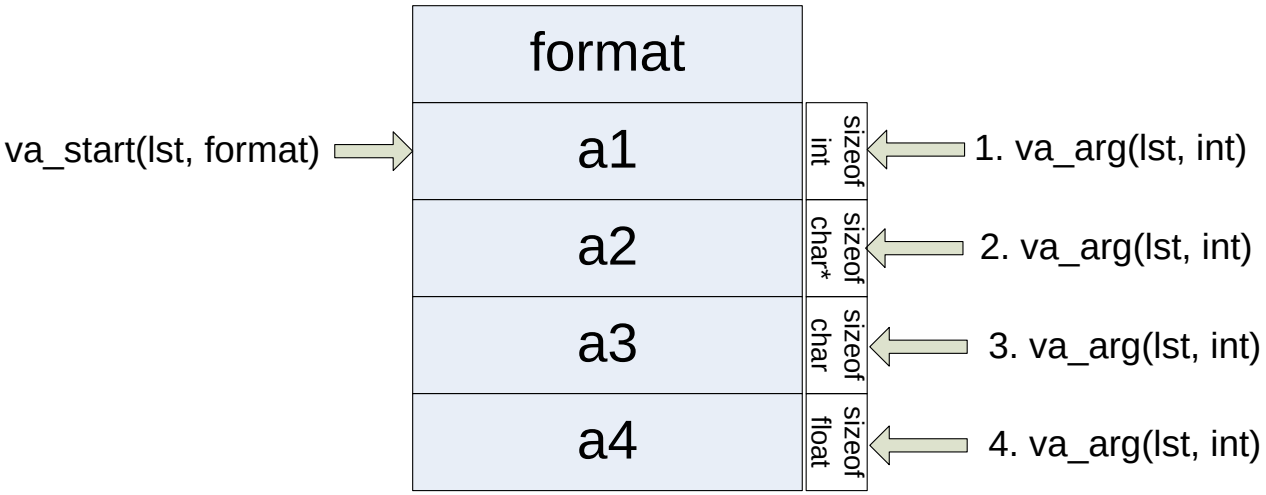
```
void foo(char *fmt, ...) {
    va_list ap;
    int d;
    char c, *s;
    va_start(ap, fmt);
    while (*fmt)
        switch(*fmt++) {
        case 's':              /* string */
            s = va_arg(ap, char *);
            printf("string %s\n", s);
            break;
        case 'd':              /* int */
            d = va_arg(ap, int);
            printf("int %d\n", d);
            break;
        case 'f':              /* char */
            f = (float) va_arg(ap, float);
            printf("float %f\n", f);
            break;
        }
    va_end(ap);
}
```

RT - RK
COMPUTER BASED SYSTEMS

Functions with variable number of arguments 2/2

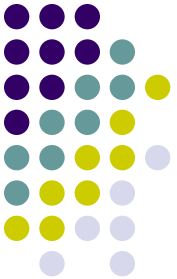
- There must be at least one parameter before …
- printf("%d %s %c %f", a1, a2, a3, a4);

- Stack frame content:



RT - RK
COMPUTER BASED SYSTEMS

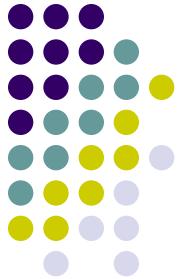
Abstract data types 1/2



- Data types introduced by programmers that consist of set of values and operations that manipulate them
- Examples: List, Queue, Stack, Circular Buffer, …
- When implementing functionality should be separated from implementation
- Abstract type can be used without knowing details about its implementation
- Changes in type implementation should not result in changes that uses that type
- In C++ this is ensured by introducing classes and class interfaces, but in C different mechanism must be used
- In C it is possible to receive pointer from function and pass pointer to function without knowing anything about nature of data to which that pointer points to.
- Based on this it is possible to have:
  - function that create instance of particular data type and return pointer (handle) to created instance
  - functions that manipulate with instance whose handle is passed as parameter
  - function that destroy instance whose handle is passed as parameter
- By using only those functions it is possible to use corresponding data type without any knowledge about its implementation

RT - RK
COMPUTER BASED SYSTEMS

Abstract data types 2/2



- Convention is that header file (.h) provides abstract view of data types without any implementation details and .c file containing implementation
- Files that work with given data type only include corresponding .h file
- Example for List data type:

MyList.h:

```
typedef struct MyListStructType *MyListType;

typedef struct MyListStructElement *MyListElement;

extern ListType InitMyList();
extern int AppendMyList(ListElement);
extern int RemoveMyList(int);
extern int DestroyMyList(ListType);
```

MyApp.c:

```
#include “MyList.h”;

MyListType lst;
...
lst = InitMyList();
AppendMyList(el1);
AppendMyList(el2);
AppendMyList(el3);
...
RemoveMyList(1);
...
DestroyMyList(lst);
...
```

MyList.c:

```
/* to ensure that implementation is
     consistent with declarations in
     .header file */
#include “MyList.h”

ListType InitMyList()
{
     …
};

int AppendMyList(ListElement)
{
     …
};

int RemoveMyList(int)
{
     …
};

int DestroyMyList(ListType)
{
     …
};
```