

Introduction to C, C++, and Unix/Linux

CS 60

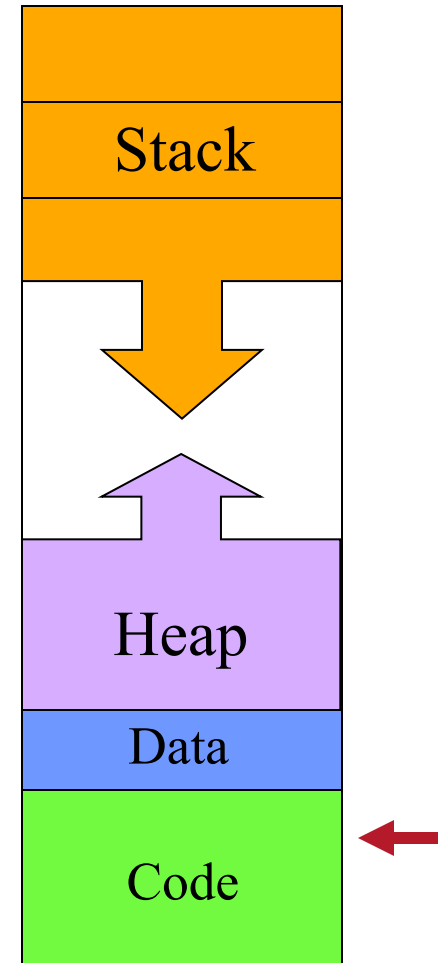
More on Functions

Today

- Finish C!
- Reading [KR] Chapters 1-7

Function pointers

- Functions (including **main**) are instructions that reside in the memory space of the application
- Function names are symbols (like variable names) that represent a memory address
- We can create pointers to functions and call functions through pointers



```
int func(int b)
{
    return (b+b) ;
}
```

```
int main(void)
{
    printf("Address of\n"
        "main() : %p\n"
        "func() : %p\n"
        "printf() : %p\n",
        &main, &func, &printf) ;
    return 0 ;
}
```

By the way...

`"abc" "def" → "abcdef"`

`%p` – prints pointer address

`return 0 → return(0)`

or `"main, func, printf) ;"`

Oddly, both work

```
int f1 (double) ;
```

```
int *f2 (double) ;
```

```
int (*f3) (double) ;
```

```
f3 = f1 ;
```

```
/* f3 = f2 ; */
```

```
x = (*f3) (3.14) ;
```

```
x = f3 (3.14) ;
```

f1 is a function that takes a double argument and returns an int

f2 is a function that takes a double argument and returns a pointer to an int

f3 is a pointer to a function that takes a double argument and returns an int

Assigns the pointer to a function

Error – type mismatch

Legal call to the function **f1**

Oddly, this is legal too!

```
int varfun(int index, int arg)
{
    int (*pf)(int);
    int val;

    pf = funlist(index);
    val = (*pf)(arg);

    return val;
}
```

pf is a pointer to a function that takes an integer argument and returns an int

funlist() is a function that returns a pointer to a function

```

int (*funlist(int index))(int)
{
    int (*pf)(int);
    switch (index) {
        case 1: pf = f1;
                break;
        case 2: pf = f2;
                break;
        default: pf = f3;
                break;
    }
    return pf;
}

int varfun(int i, int arg)
{
    int (*pf)(int);
    int val;

    pf = funlist(i);
    val = (*pf)(arg);

    return val;
}

```

Casting a function pointer

```
int x, (*f3) (double) ;  
unsigned int addr = 0x8ff324 ;  
  
f3 = addr; /* Warning */  
  
f3 = (int (*) (double)) addr ;  
  
x = (*f3) (3.14159) ;  
x = f3 (3.14159) ;      /* or this way */
```

Recursive call to main

```
int main(void)
{
    typedef int (*PM) (void);
    ...
    PM fp = main; /* or &main */
    (*fp) ();
    ...
}
```

or just call **main()**

To review...

- Variable **x**
 - A symbol (name) representing an address where we store some value of a certain type (**int**, **double**, **char***, ...)
 - Address = **&x**, Value = **x**
- Array variable **a**
 - A symbol (name) representing an address where a block of memory has been reserved for values of a certain type
 - Address = **a**, Values = **a[0]**, **a[1]**, ...
- Function name **func**
 - A symbol representing the address of a piece of code which we can jump to, and designating the parameter and return variable types
 - Address: **&func** or **func**

Passing parameters (pass-by-value)

```
a=1; b=2;
swap(a,b);
// After the call ...
// a has value 1 and b has value 2
// because values are copied
void swap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

Passing arrays

```
a[0]=0; a[1]=1;
swap(a);
// After the call ...
// a[0] has value 1 and a[1] has a 0
// because address of first element of
// x is same as the first element of a.
void swap(int x[])
{
    int temp = x[0];
    x[0] = x[1];
    x[1] = temp;
}
```

Passing structs is expensive

- Huge structs are expensive to pass because they are copied.

Passing Pointer (like List* last class)

```
printit(header) ;
```

```
// The pointer to the header node  
// stored in header is stored in  
// variable x at the time of the call.  
// Both header and x point to same loc.
```

```
void printit(List* x)  
{  
    // prints the List ...  
    ...  
}
```

Creating a list in function (like initialize list)

```
set_union(a_header,b_header,c_header);
```

```
void set_union(List* a_h,b_h,c_h)
{
    // Performs the union of sets a_h
    // and b_h leaving the result in a
    // newly created list c_h
    ...
}
// c_h does not get back when we do
// c_h=(List*)malloc(sizeof(List));
// in set_union
```

Creating a list in function (returning List*)

```
c_header=set_union(a_header,b_header);
```

```
List* set_union(List* a_h,b_h)
{
    // Performs the union of sets a_h
    // and b_h leaving the result in a
    // newly created list c_h which is
    // returned
    return(c_h);
}
// c_h gets back correctly. But if
// before the call c_header had a value
// (a List) it will cause a memory leak
```

Passing parameters (pass-by-address)

```
a=1; b=2;
swap(&a, &b);
// After the call ...
// a has value 2 and b has value 1
// because px becomes the address of a
// and py becomes the address of b
void swap(int *px, int *py)
{
    int temp = *px;
    *px = *py;
    *py = temp;
}
```

Passing parameters (pass-by-reference) in C++

```
a=1; b=2;
swap(a,b);
// After the call ...
// a has value 2 and b has value 1
// because x is the same as a
// and y is the same as b
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

2D arrays: ****data**

```
int **data;  
data = (int **)malloc(12*sizeof(int *));  
for (i=0; i<12; i++)  
    data[i] = (int *)malloc(4*sizeof(int));
```

Creates and allocates a 12x4 array of integers

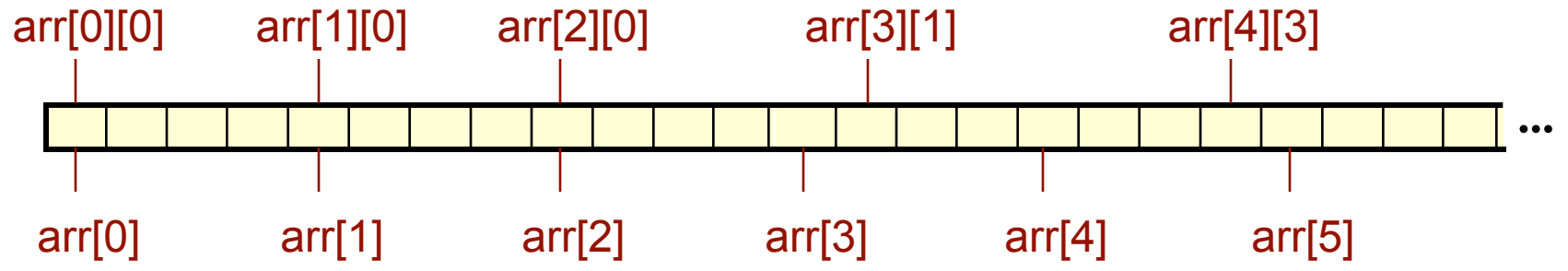
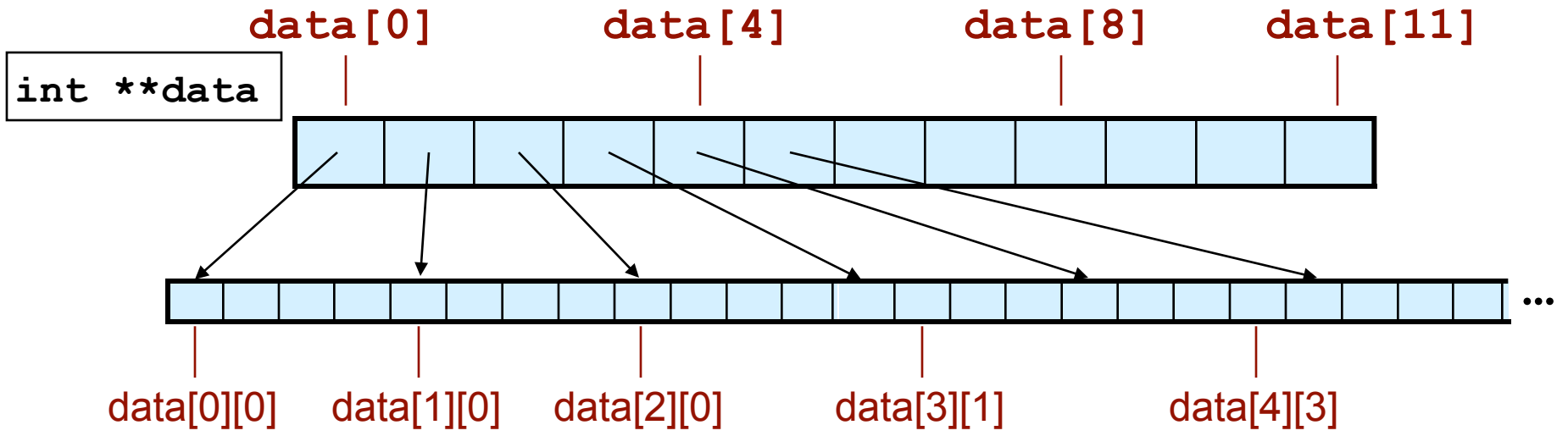
Similar to **data[12][4]**

2D arrays: ****data**

`typedef int* pint;`

```
typedef int *pint;  
pint *data;  
data = (pint *)malloc(12*sizeof(pint));  
for (i=0; i<12; i++)  
    data[i] = (int *)malloc(4*sizeof(int));
```

Same as previous slide (typedef makes it clearer)



```
int arr[12][4];
```

- How many bytes will a 4x4 char array take up?

char arr[4][4]; ← 16 bytes

char **arr; ← 32 bytes

|
16 bytes of addresses (4 pointers) + 16 bytes of data

Passing 2D arrays of any size

```
int **a,n,m;
scanf("%d%d",&n,&m);
a = (int **)malloc(n*sizeof(int *));
for (i=0; i<n; i++)
    a[i] = (int *)malloc(m*sizeof(int));
sum(a,n,m);

void sum(int **x,int n,m)
{
    // Use x[i][j] without problems
}
```

Passing 2D arrays of fixed size

```
int a[10][20],n,m;
```

```
sum(a,n,m);
```

```
void sum(int x[][20],int n,m)
{ // 20 must be a constant matching
  // the declaration in calling function.

  // Use x[i][j] without problems
  // but only x[][20]
}
```

- What's the value of **x**?

```
char arr[][2] = {{1, 2}, {3, 4}};  
int x = (int)arr[1,1];
```

x is -1073994722 !!

Because [1,1] evaluates to [1] (remember the comma operator!)
So this is equivalent to

```
int x = (int)arr[1];
```

which sets **x** to the address of **arr[1][0]**

or

```
int main(int argc, char **argv)
```

```
int main(int argc, char *argv[])
```

- General declaration for **main** function
 - **argc** – argument count
 - ◆ The number of command arguments the program was invoked with
 - **argv** – argument vector
 - ◆ A pointer to an array of character strings, one for each argument

```
% myprog -1 32 -p /usr/local
```

```
argc = 5
```

```
argv[0] = "myprog"
```

```
argv[1] = "-1"
```

```
argv[2] = "32"
```

```
argv[3] = "-p"
```

```
argv[4] = "/usr/local"
```

```
% myprog -l 32 -p /usr/local
```

```
for (i=1; i<argc; i++)
{
    if (!strcmp(argv[i], "-h"))
        help = 1;
    else if (!strcmp(argv[i], "-l"))
        num = atoi(argv[++i]);
    else if (!strcmp(argv[i], "-p")) {
        filename = (char *)malloc(128);
        strcpy(filename, argv[++i]);
    }
}
```

Useful string utility functions (B.3 and B.5)

`strcpy`

`atoi`

`strncpy`

`atof`

`strcmp`

`atol`

`strncmp`

`rand`

`strlen`

`exit`

`strchr`

`system`

`strrchr`

`abs`

...and more...

Memory functions

memcpy

memmove

memcmp

memchr

memset

To copy a whole block of memory,
don't copy byte by byte – use
memcpy!

Math functions (B.4)

sin

cos

tan

asin

acos

atan

atan2

exp

log

log10

pow

sqrt

ceil

floor

fabs

...and more...