Embedded C Part -2

- Swarna Prabha Jena

Department of ECE

Embedded C
Functions – What?

An activity that is natural to or the purpose of a person or thing.

"bridges perform the function of providing access across water"

A relation or expression involving one or more variables.

"the function (bx + c)"

Source: Google

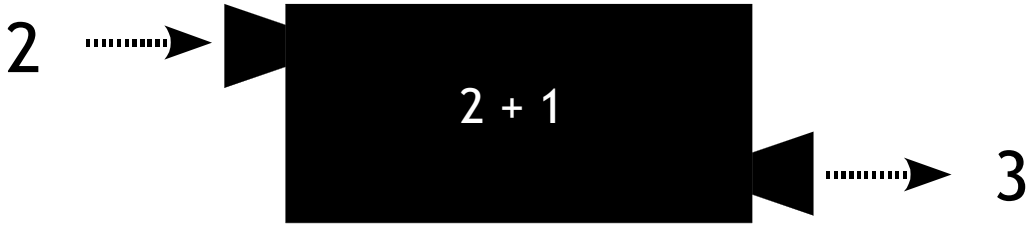
- In programing languages it can something which performs a specific service
- Generally it can have 3 parameters like
  - Input
  - Operation
  - Output

Embedded C

Functions – What?

$f(x) = x + 1$

$x = 2$



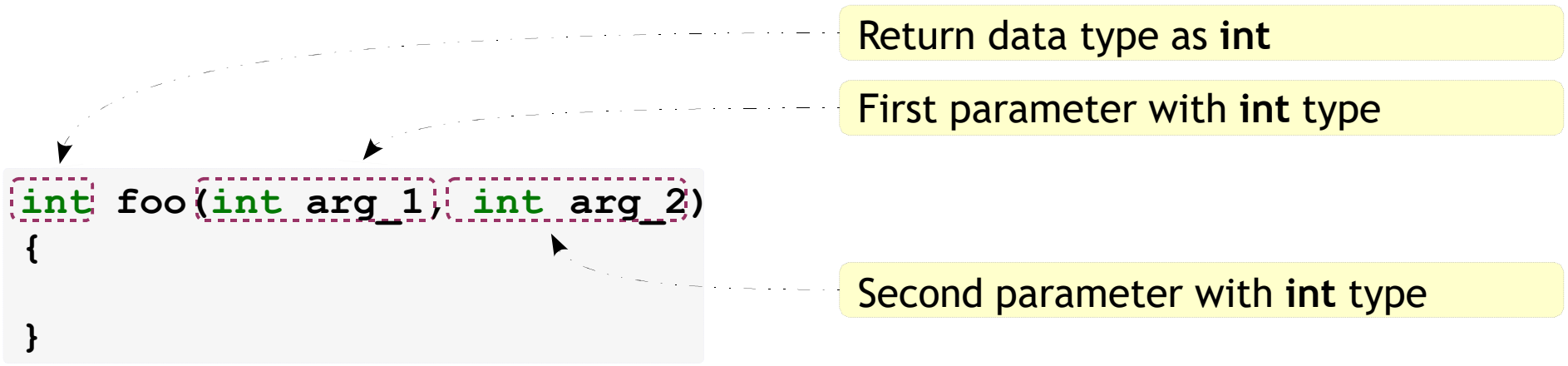
Embedded C

Functions – How to write

Syntax

```
return_data_type function_name(arg_1, arg_2, ..., arg_n)
{
    /* Function Body */
}
```

Example



Embedded C

Functions – How to write

$$y = x + 1$$

Example

```
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

Return variable or argument function

Embedded C

Functions - How to call

Example

```c
#include <stdio.h>

int main()
{
    int x, y;

    x = 2;

    y = foo(x);
    printf("y is %d\n", y);

    return 0;
}
```

The function call

```c
int foo(int x)
{
    int ret = 0;

    ret = x + 1;

    return ret;
}
```

Embedded C

Functions – Why?

- Re usability
  - Functions can be stored in library & re-used
  - When some specific code is to be used more than once, at different places, functions avoids repetition of the code.
- Divide & Conquer
  - A big & difficult problem can be divided into smaller sub-problems
    and solved using divide & conquer technique
- Modularity can be achieved.
- Code can be easily understandable & modifiable. Functions are easy to debug & test.
- One can suppress, how the task is done inside function, the
  which is called Abstraction

Embedded C

Functions – A complete look

Example

```c
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

- The main function
- The function call
- Actual arguments
- Return type
- Formal arguments
- Fournmctaidnarguments
- Roetmuarnlfarogmumfuenctsion

Embedded C

Functions – Ignoring return value

Example

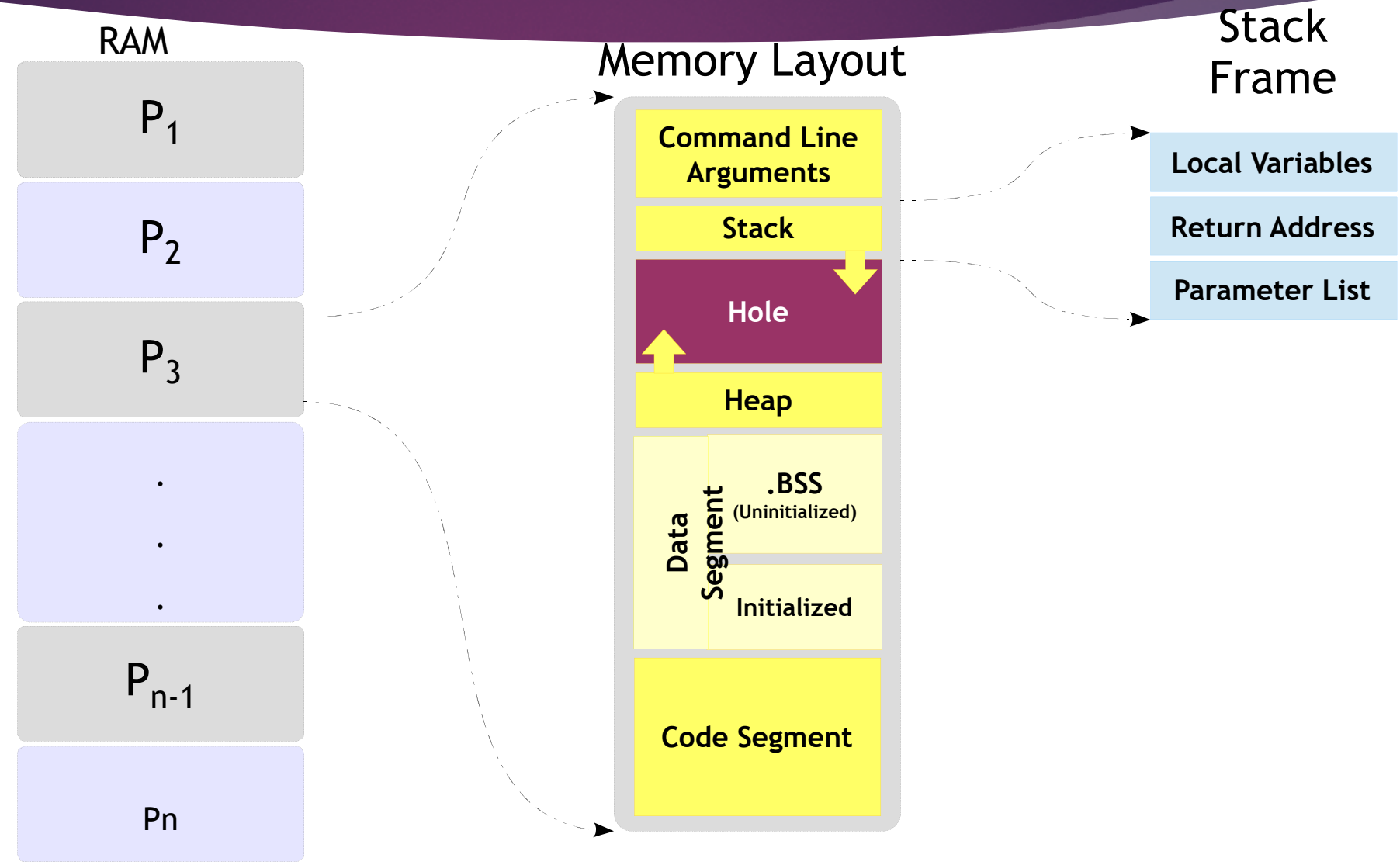
```
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

Ignored the return from function
In C, it is up to the programmer to capture or ignore the return value

```
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Embedded C

Function and the Stack



Embedded C

Functions – Parameter Passing Types

Pass by Value	Pass by reference
• This method copies the actual value of an argument into the formal parameter of the function.	• This method copies the address of an argument into the formal parameter.
• In this case, changes made to the parameter inside the function have no effect on the actual argument.	• Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

Embedded C

Functions - Pass by Value

Example

```c
#include <stdio.h>

int add_numbers(int num1, int num2);

int main()
{
    int num1 = 10, num2 = 20, sum;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```c
int add_numbers(int num1, int num2)
{
    int sum = 0;

    sum = num1 + num2;

    return sum;
}
```

Embedded C

Functions – Pass by Value

Example

```
#include <stdio.h>

void modify(int num1);

int main()
{
    int num1 = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num1);

    modify(num1);

    printf("After Modification\n");
    printf("num1 is %d\n", num1);

    return 0;
}
```

```
void modify(int num1)
{
    num1 = num1 + 1;
}
```

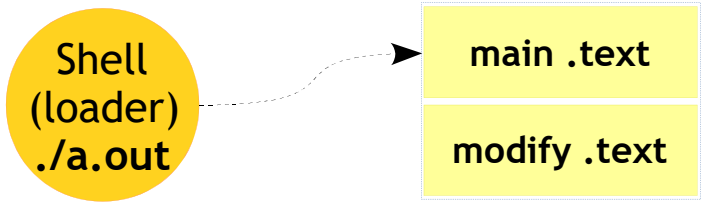
Embedded C

Functions - Pass by Reference

Example

```c
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```c
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```



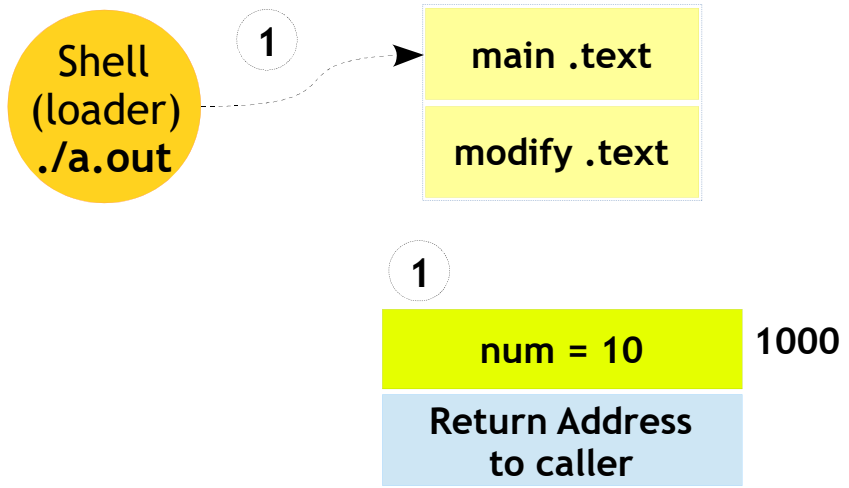
Embedded C

Functions – Pass by Reference

Example

```
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```



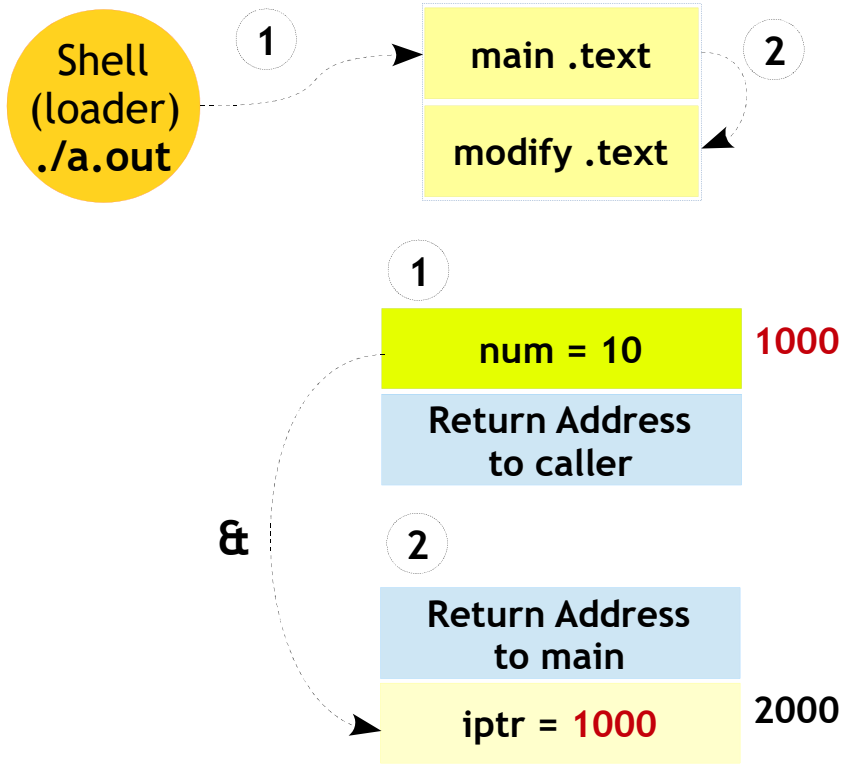
Embedded C

Functions - Pass by Reference

Example

```
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
```

```
void modify(int *iptr)
{
    *iptr = *iptr + 1;
}
```

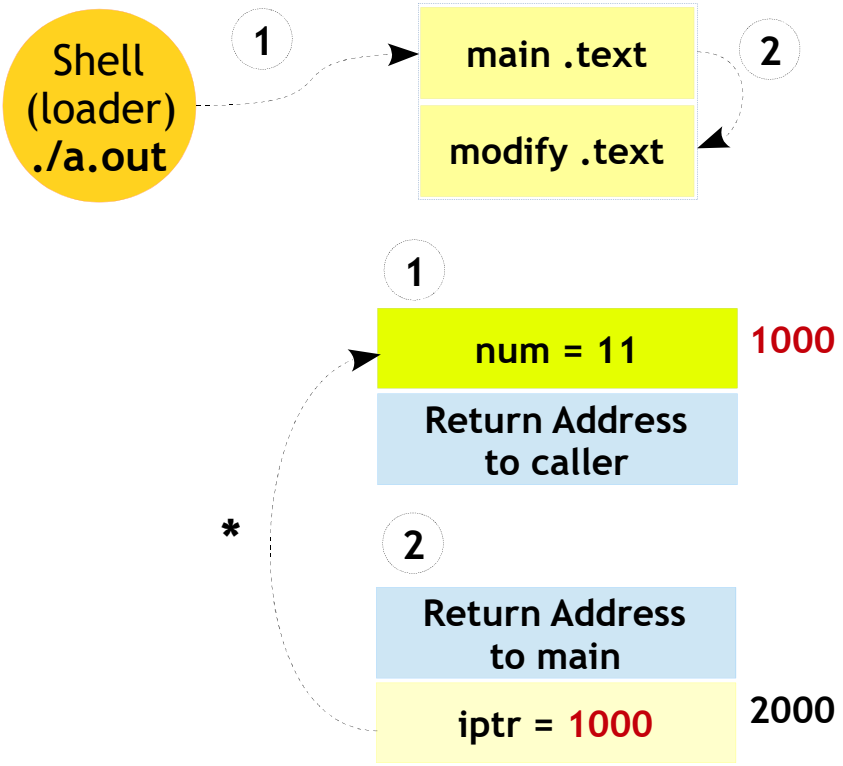


Embedded C

Functions - Pass by Reference

Example

```
#include <stdio.h>

void modify(int *iptr);

int main()
{
    int num = 10;

    printf("Before Modification\n");
    printf("num1 is %d\n", num);

    modify(&num);

    printf("After Modification\n");
    printf("num1 is %d\n", num);

    return 0;
}
void modify(int *iptr)
{
    *iptr = *iptr + 1;
```



Embedded C

Functions – Pass by Reference - Advantages

- Return more than one value from a function
- Copy of the argument is not made, making it fast, even when used with large variables like arrays etc.
- Saving stack space if argument variables are larger (example – user defined data types)

Embedded C

Functions – Passing Array

- As mentioned in previous slide passing an array to function can be faster
- But before you proceed further it is expected you are familiar with some pointer rules
- If you are OK with your concepts proceed further, else please know the rules first

Embedded C

Functions – Passing Array

Example

```
#include <stdio.h>

void print_array(int array[]);

int main()
{
     int array[5] = {10, 20, 30, 40, 50};

     print_array(array);

     return 0;
}
```

```
void print_array(int array[])
{
     int i;

     for (i = 0; i < 5; i++)
     {
          printf("Index %d has Element %d\n", i, array[i]);
     }
}
```

Embedded C

Functions – Passing Array

Example

```c
#include <stdio.h>

void print_array(int *array);

int main()
{
    int array[5] = {10, 20, 30, 40, 50};

    print_array(array);

    return 0;
}
```

```c
void print_array(int *array)
{
    int i;

    for (i = 0; i < 5; i++)
    {
        printf("Index %d has Element %d\n", i, *array);
        array++;
    }
}
```

Embedded C

Functions – Passing Array

Example

```
#include <stdio.h>

void print_array(int *array, int size);

int main()
{
     int array[5] = {10, 20, 30, 40, 50};

     print_array(array, 5);

     return 0;
}
```

```
void print_array(int *array, int size)
{
     int i;

     for (i = 0; i < size; i++)
     {
          printf("Index %d has Element %d\n", i, *array++);
     }
}
```

Embedded C

Functions – Returning Array

Example

```c
#include <stdio.h>

int *modify_array(int *array, int size);
void print_array(int array[], int size);

int main()
{
     int array[5] = {10, 20, 30, 40, 50};
     int *iptr;

     iptr = modify_array(array, 5);
     print_array(iptr, 5);

     return 0;
}
```

```c
int *modify_array(int *array, int size)
{
     int  i;

     for (i = 0; i < size; i++)
     {
          *(array + i) += 10;
     }

     return array;
}
```

```c
void print_array(int array[], int size)
{
     int i;

     for (i = 0; i < size; i++)
     {
          printf("Index %d has Element %d\n", i, array[iter]);
     }
}
```

Embedded C

Functions – Returning Array

Example

```c
#include <stdio.h>

int *return_array(void);
void print_array(int *array, int size);

int main()
{
     int *array_val;

     array_val = return_array();
     print_array(array_val, 5);

     return 0;
}
```

```c
int *return_array(void)
{
     static int array[5] = {10, 20, 30, 40, 50};

     return array;
}
```

```c
void print_array(int *array, int size)
{
     int i;

     for (i = 0; i < size; i++)
     {
          printf("Index %d has Element %d\n", i, array[i]);
     }
}
```

Embedded C

Functions – Recursive

- Recursion is the process of repeating items in a self-similar way
- In programming a function calling itself is called as recursive function
- Two steps

  Step 1: Identification of base case

  Step 2: Writing a recursive case

Embedded C

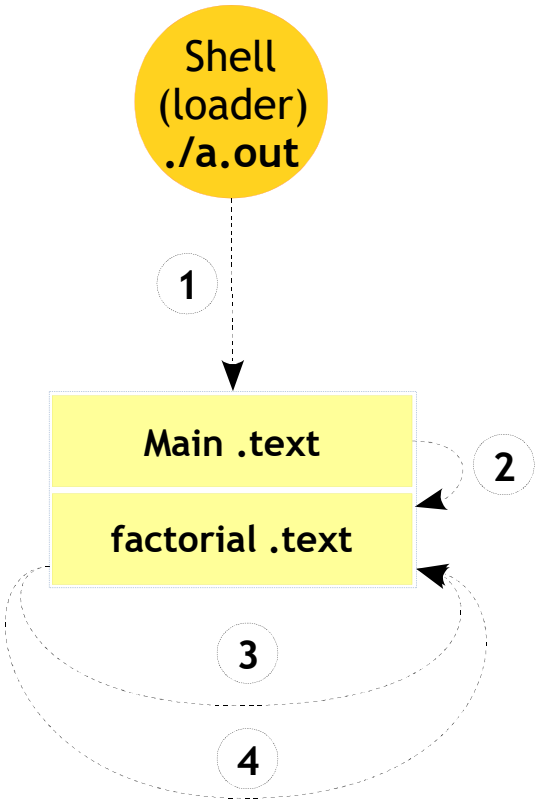
Functions – Recursive – Example

Example

```c
#include <stdio.h>

/* Factorial of 3 numbers */

int factorial(int number)
{
    if (number <= 1)
    {
        return 1;
    }
    else
    {
        return number * factorial(number - 1);
    }
}

int main()
{
    int result;

    result = factorial(3);
    printf("Factorial of 3 is %d\n", result);

    return 0;
}
```

Embedded C

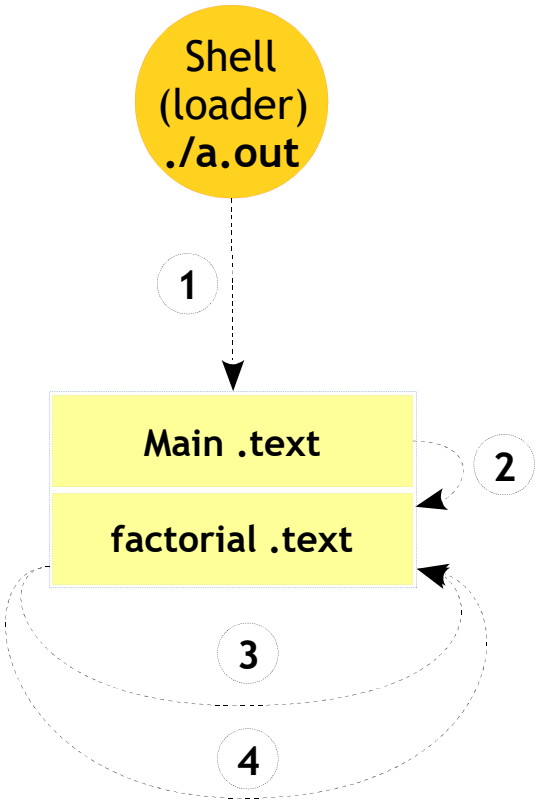
Functions – Recursive – Example Flow



	Stack Frames	Value with calls
1	result Return Address to caller	factorial(3)
2	Return Address to main number = 3	number != 1 number * factorial(number - 1) 3 * factorial(3 - 1)
3	Return Address to factorial number = 2	number != 1 number * factorial(number - 1) 2 * factorial(2 - 1)
4	Return Address to factorial number = 1	number == 1

Embedded C

Functions – Recursive – Example Flow



	Stack Frames	Results with return
1	result Return Address to caller	Gets 6 a value
2	~~Return Address to main~~ ~~number = 3~~	Returns 3 * 2 to the caller
3	~~Return Address to factorial~~ ~~number = 2~~	Returns 2 * 1 to the caller
4	~~Return Address to factorial~~ ~~number = 1~~	returns 1 to the caller

Embedded C
Functions – Function Pointers

- A variable that stores the pointer to a function.
- Chunk of code that can be called independently and is standalone
- “Registering” a piece of code and calling it later when required

Syntax

```
return_datatype (*foo)(datatype, ...);
```

Embedded C

Functions – Function Pointers

Example

```c
#include <stdio.h>

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int (*function)(int, int);

    function = add;
    printf("%d\n", function(2, 4));

    return 0;
}
```

Example

```c
#include <stdio.h>

void run_at_exit(void)
{
    printf("Exiting\n");
}

int main()
{
    printf("In Main Function\n");

    atexit(run_at_exit);

    return 0;
}
```

Embedded C
Functions – Variadic

- Variadic functions can be called with any number of trailing arguments
- For example,

  printf(), scanf() are common variadic functions
- Variadic functions can be called in the usual way with individual arguments

Syntax

```
return_data_type function_name(parameter list, ...);
```

Embedded C

Functions – Variadic – Definition & Usage

- Defining and using a variadic function involves three steps:

Step 1: Variadic functions are defined using an ellipsis ('…') in the argument list, and using special macros to access the variable arguments.

Example

```
int foo(int a, ...)
{
    /* Function Body */
}
```

Step 2: Declare the function as variadic, using a prototype with an ellipsis ('…'), in all the files which call it.

Step 3: Call the function by writing the fixed arguments followed by the additional variable arguments.

Embedded C

Functions – Variadic – Argument access macros

- Descriptions of the macros used to retrieve variable arguments
- These macros are defined in the header file stdarg.h

Type/Macros	Description
va_list	The type va_list is used for argument pointer variables
va_start	This macro initializes the argument pointer variable ap to point to the first of the optional arguments of the current function; last-required must be the last required argument to the function
va_arg	The va_arg macro returns the value of the next optional argument, and modifies the value of ap to point to the subsequent argument. Thus, successive uses of va_arg return successive optional arguments
va_end	This ends the use of ap

Embedded C

Functions – Variadic – Example

Example

```c
#include <stdio.h>

int main()
{
    int ret;

    ret = add(3, 2, 4, 4);
    printf("Sum is %d\n", ret);

    ret = add(5, 3, 3, 4, 5, 10);
    printf("Sum is %d\n", ret);

    return 0;
}
```

```c
int add(int count, ...)
{
    va_list ap;
    int iter, sum;

    /* Initialize the arg list */
    va_start(ap, count);

    sum = 0;
    for (iter = 0; iter < count; iter++)
    {
        /* Extract args */
        sum += va_arg(ap, int);
    }

    /* Cleanup */
    va_end(ap);

    return sum;
}
```

Embedded C

Pointers – Pitfalls – Segmentation Fault

- A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed.

Example

```c
#include <stdio.h>

int main()
{
    int num = 0;

    printf("Enter the number\n");
    scanf("%d", num);

    return 0;
}
```

Example

```c
#include <stdio.h>

int main()
{
    int *num = 0;

    printf("num = %d\n", *num);

    return 0;
}
```

Embedded C

Pointers – Pitfalls – Dangling Pointer

- A dangling pointer is something which does not point to a valid location any more.

Example

```c
#include <stdio.h>

int main()
{
    int *iptr;

    iptr = malloc(4);
    free(iptr);

    *iptr = 100;

    return 0;
}
```

Example

```c
#include <stdio.h>

int *foo(void)
{
    int num = 5;

    return &num_ptr;
}

int main()
{
    int *iptr;

    iptr = foo();

    return 0;
}
```

Embedded C

Pointers – Pitfalls – Wild Pointer

- An uninitialized pointer pointing to a invalid location can be called as an wild pointer.

Example

```
#include <stdio.h>

int main()
{
    int *iptr_1; /* Wild Pointer */
    static int *iptr_2; / Not a wild pointer */


    return 0;
}
```

Embedded C

Pointers – Pitfall - Memory Leak

- Improper usage of the memory allocation will lead to memory leaks
- Failing to deallocating memory which is no longer needed is one of most common issue.
- Can exhaust available system memory as an application runs longer.

Embedded C

Pointers – Pitfall - Memory Leak

Example

```
#include <stdio.h>

int main()
{
     int *num_array, sum = 0, no_of_elements, i;

     while (1)
     {
          printf("Enter the number of elements: \n");
          scanf("%d", &no_of_elements);
          num_array = malloc(no_of_elements);

          sum = 0;
          for (i = 0; i < no_of_elements; i++)
          {
               scanf("%d", &num_array[i]);
               sum += num_array[i];
          }

          printf("The sum of array elements are %d\n", sum);
          /* Forgot to free!! */
     }
     return 0;
}
```

Embedded C

Pointers – Pitfalls – Bus Error

- A bus error is a fault raised by hardware, notifying an operating system (OS) that, a process is trying to access memory that the CPU cannot physically address: an invalid address for the address bus, hence the name.

Example

```c
#include <stdio.h>

int main()
{
    char array[sizeof(int) + 1];
    int *ptr1, *ptr2;

    ptr1 = &array[0];
    ptr2 = &array[1];

    scanf("%d %d", ptr1, ptr2);

    return 0;
}
```

Embedded C
Pointers - Const Qualifier

Example

```
#include <stdio.h>

int main()
{
    int const *num = NULL;

    return 0;
}
```

The location, its pointing to is constant

Example

```
#include <stdio.h>

int main()
{
    int * const num = NULL;

    return 0;
}
```

The pointer is constant

Embedded C

Pointers – Multilevel

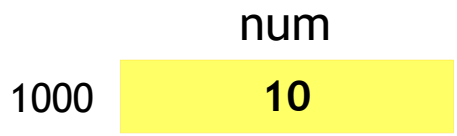
- A pointer, pointing to another pointer which can be pointing to others pointers and so on is know as multilevel pointers.
- We can have any level of pointers.
- As the depth of the level increase we have to bit careful while dealing with it.

Embedded C

Pointers – Multilevel

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

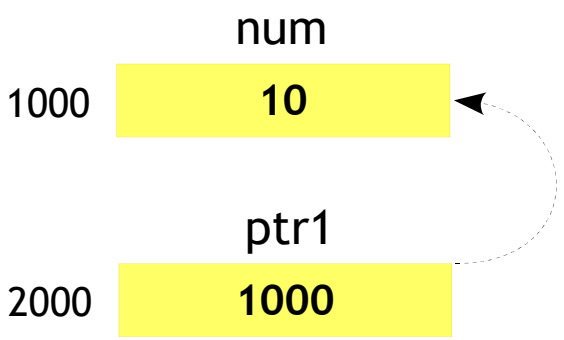


Embedded C

Pointers – Multilevel

Example

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

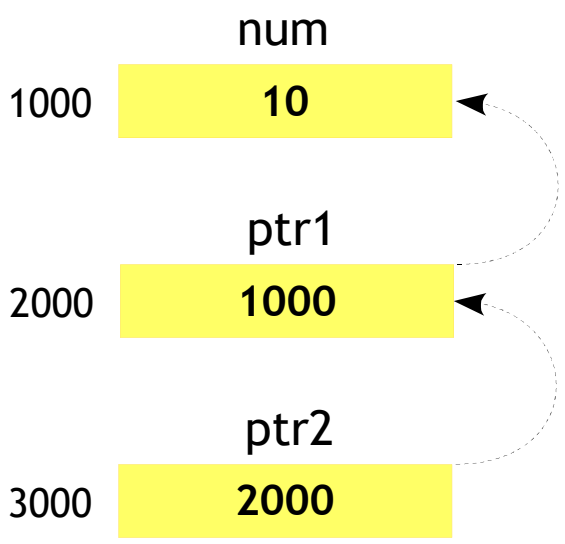


Embedded C

Pointers – Multilevel

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

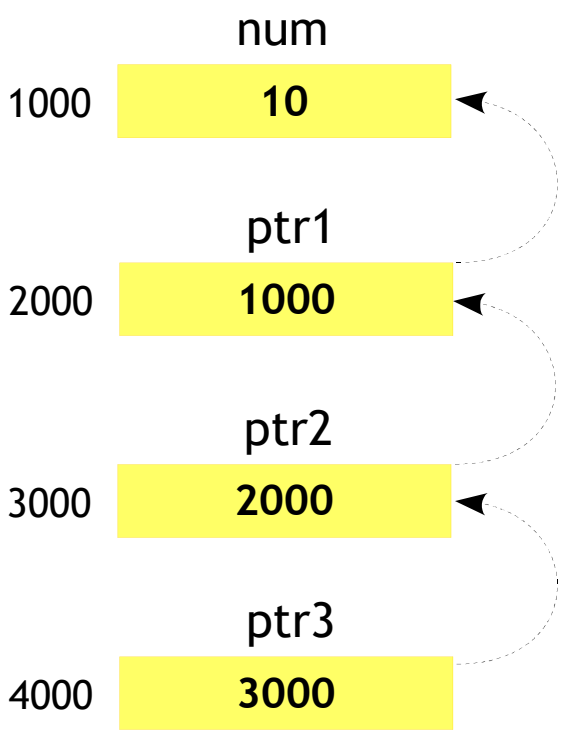


Embedded C

Pointers – Multilevel

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```

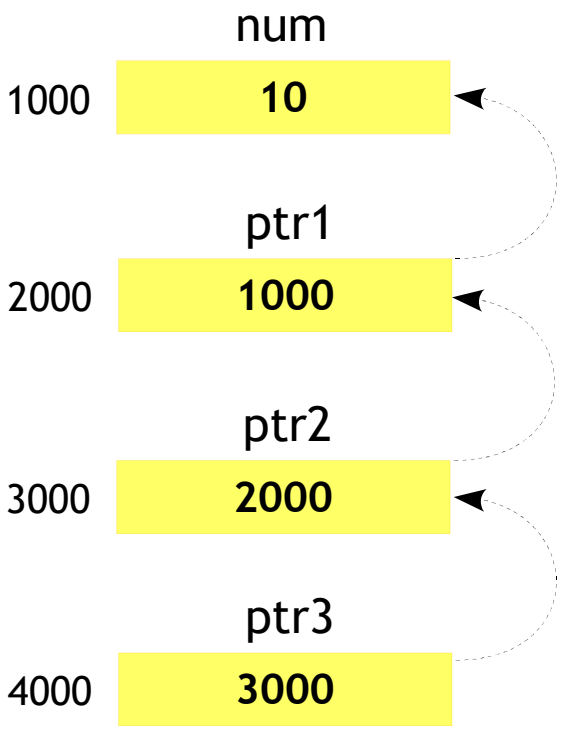


Embedded C

Pointers – Multilevel

Example

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



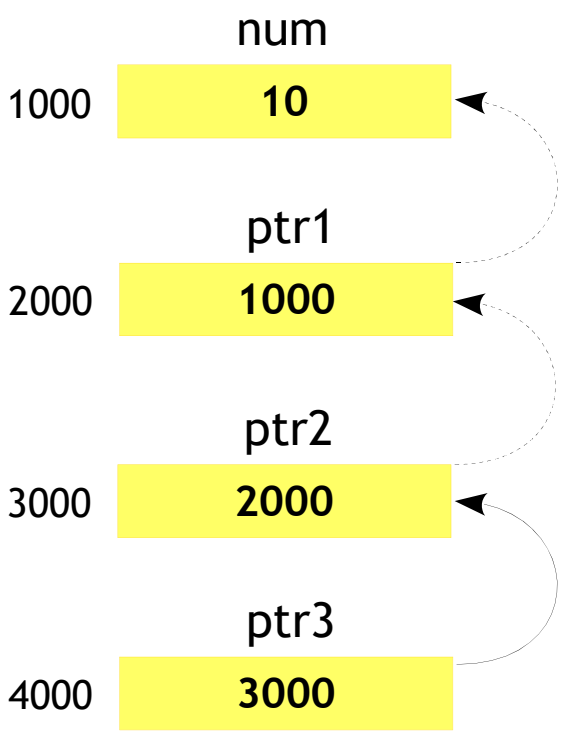
Output → 3000

Embedded C

Pointers – Multilevel

Example

```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



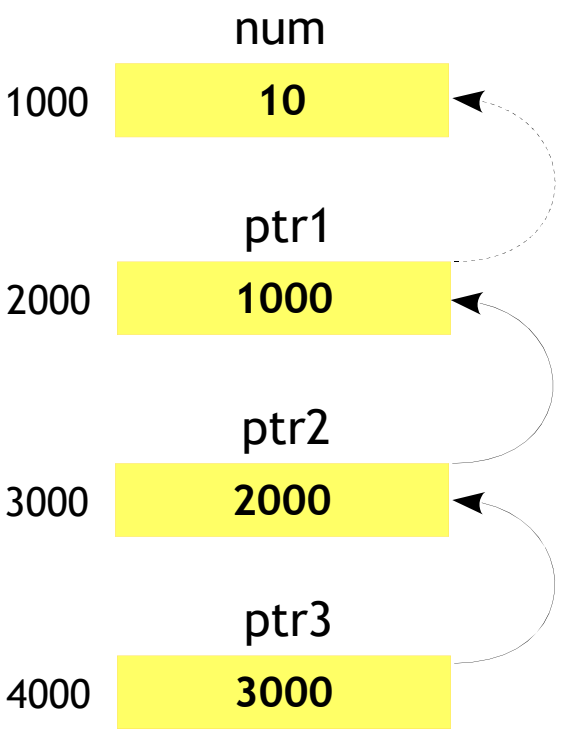
Output → 2000

Embedded C

Pointers – Multilevel

Example

```
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



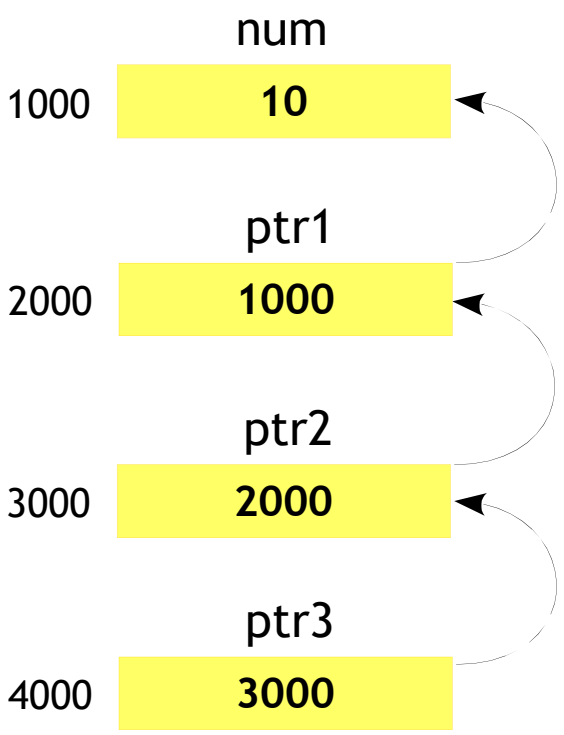
Output → 1000

Embedded C

Pointers – Multilevel

Example

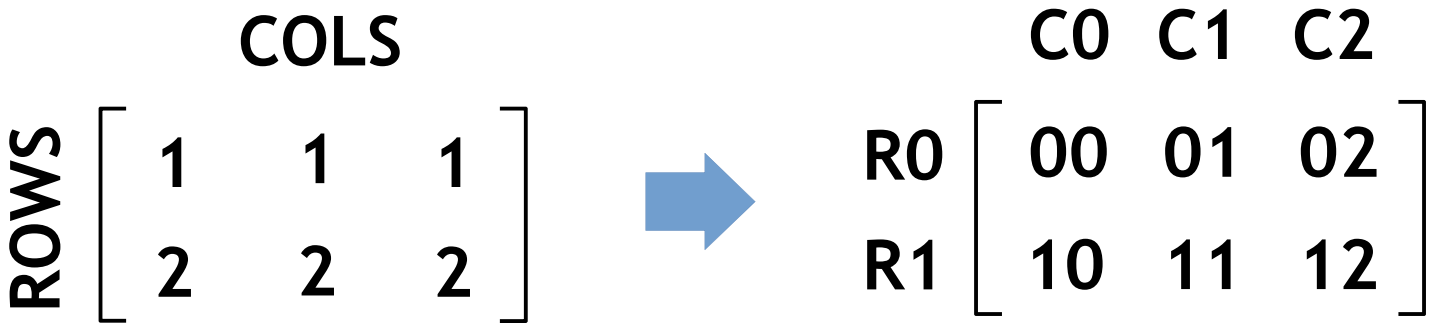
```c
#include <stdio.h>

int main()
{
    int num = 10;
    int *ptr1 = &num;
    int **ptr2 = &ptr1;
    int ***ptr3 = &ptr2;

    printf("%d", ptr3);
    printf("%d", *ptr3);
    printf("%d", **ptr3);
    printf("%d", ***ptr3);

    return 0;
}
```



Output → 10

Embedded C

Pointers – 2D Array



Embedded C

Pointers - 2D Array

Example

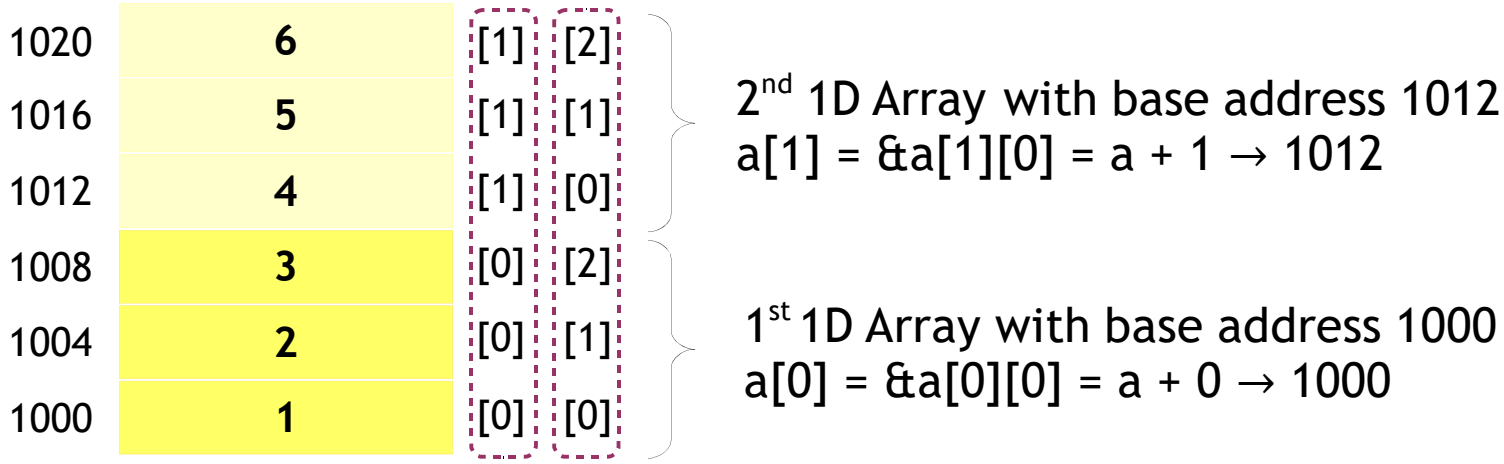
```c
#include <stdio.h>

int main()
{
    int a[2][3] = {1, 2, 3, 4, 5, 6};

    return 0;
}
```

```
Total Memory: ROWS * COLS * sizeof(datatype) Bytes
```

Embedded C

Pointers – 2D Array - Referencing

2 * 1D array linearly placed in memory



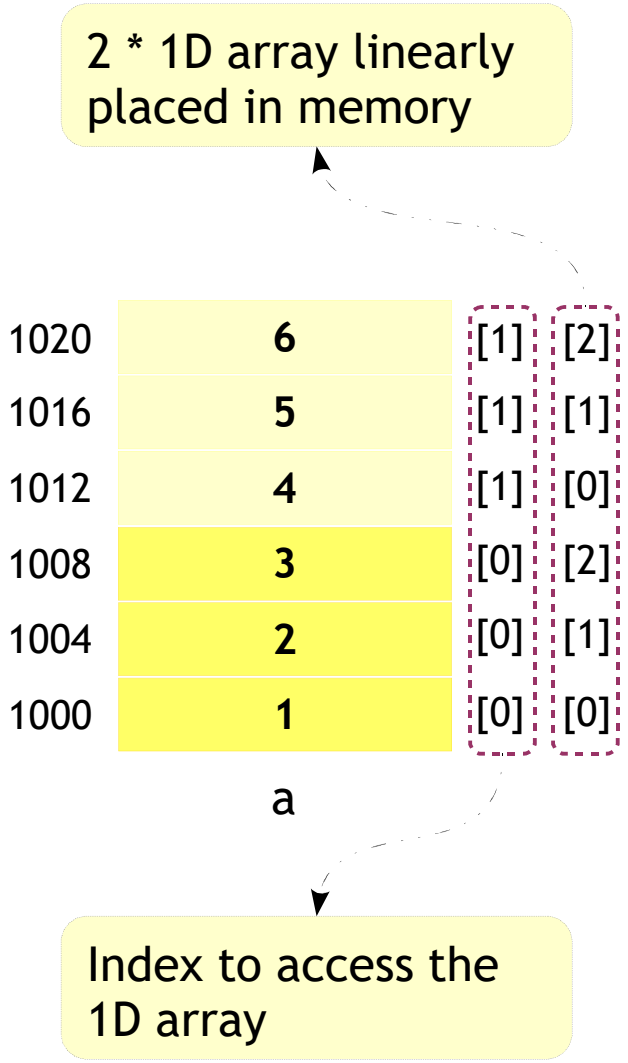
2^{nd} 1D Array with base address 1012
a[1] = &a[1][0] = a + 1 → 1012

1^{st} 1D Array with base address 1000
a[0] = &a[0][0] = a + 0 → 1000

Index to access the 1D array

Embedded C

Pointers – 2D Array - Dereferencing



Example 1: Say **a[0][1]** is to be accessed, then decomposition happens like,

a[0][1] = *(a[0] + 1 * sizeof(type))
= *(*(a + 0 * sizeof(1D array)) + 1 * sizeof(type))
= *(*(1000 + 0 * 12) + 1 * 4)
= *(*(1000 + 0) + 4)
= *(*(1004))
= 2

Example 2: Say **a[1][2]** is to be accessed, then decomposition happens like,

a[1][2] = *(a[1] + 2 * sizeof(type))
= *(*(a + 1 * sizeof(1D array)) + 2 * sizeof(type))
= *(*(1000 + 1 * 12) + 2 * 4)
= *(*(1000 + 12) + 8)
= *(*(1020))
= 6

Embedded C

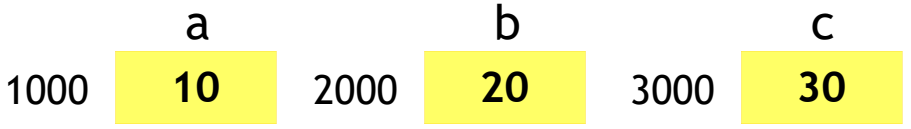
Pointers - Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

Example

```
int a = 10;
int b = 20;
int c = 30;

int *ptr[3];

ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;
```



Embedded C

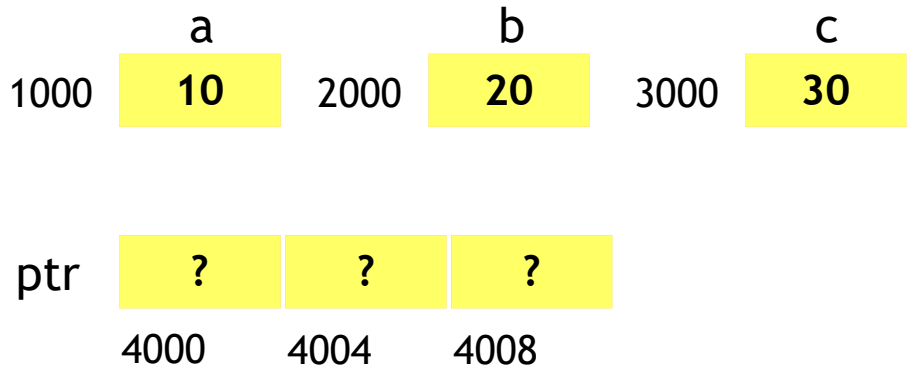
Pointers – Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

Example

```
int a = 10;
int b = 20;
int c = 30;

int *ptr[3];

ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;
```



Embedded C

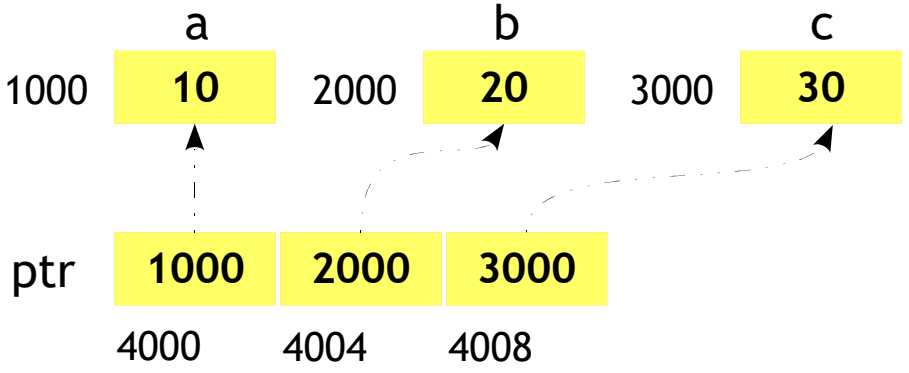
Pointers – Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

Example

```
int a = 10;
int b = 20;
int c = 30;

int *ptr[3];

ptr[0] = &a;
ptr[1] = &b;
ptr[2] = &c;
```



Embedded C

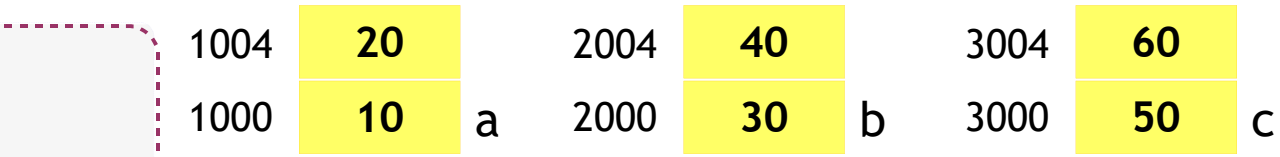
Pointers – Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

Example

```
int a[2] = {10, 20};
int b[2] = {30, 40};
int c[2] = {50, 60};

int *ptr[3];

ptr[0] = a;
ptr[1] = b;
ptr[2] = c;
```



Embedded C

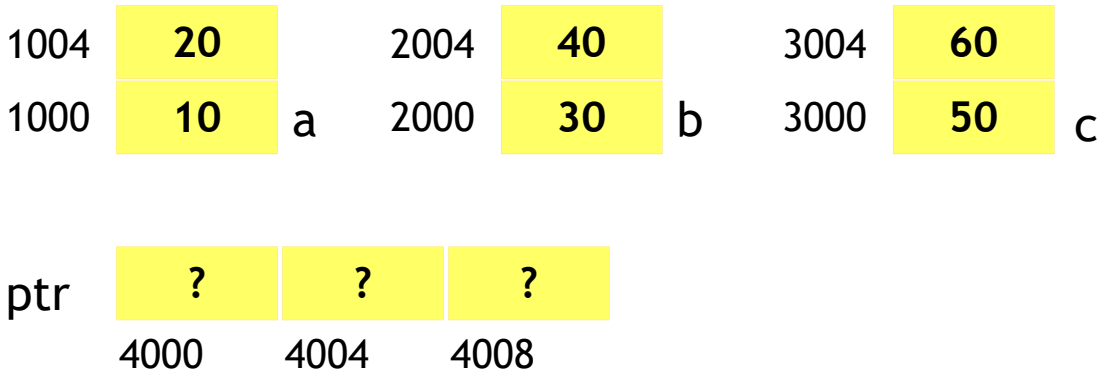
Pointers – Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

Example

```
int a[2] = {10, 20};
int b[2] = {30, 40};
int c[2] = {50, 60};

int *ptr[3];

ptr[0] = a;
ptr[1] = b;
ptr[2] = c;
```



Embedded C

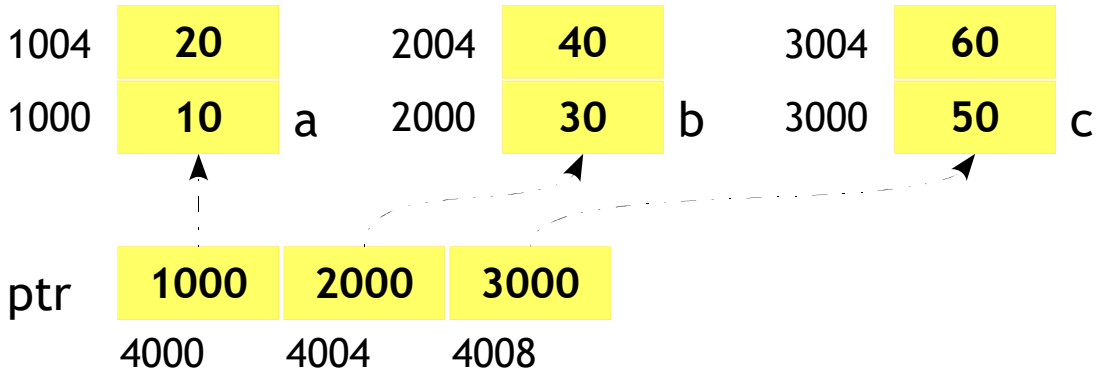
Pointers - Array of pointers

Syntax

```
datatype *ptr_name[SIZE]
```

Example

```
int a[2] = {10, 20};
int b[2] = {30, 40};
int c[2] = {50, 60};

int *ptr[3];

ptr[0] = a;
ptr[1] = b;
ptr[2] = c;
```

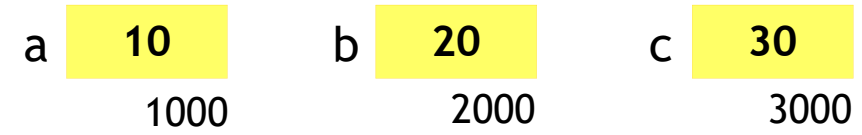


Embedded C

Pointers – Array of pointers

Example

```
#include <stdio.h>

void print_array(int **p)
{
     int i;

     for (i = 0; i < 3; i++)
     {
          printf("%d ", *p[i]);
          printf("at %p\n", p[i]);
     }
}

int main()
{
     int a = 10;
     int b = 20;
     int c = 30;
     int *ptr[3] = {&a, &b, &c};

     print_array(ptr);

     return 0;
}
```

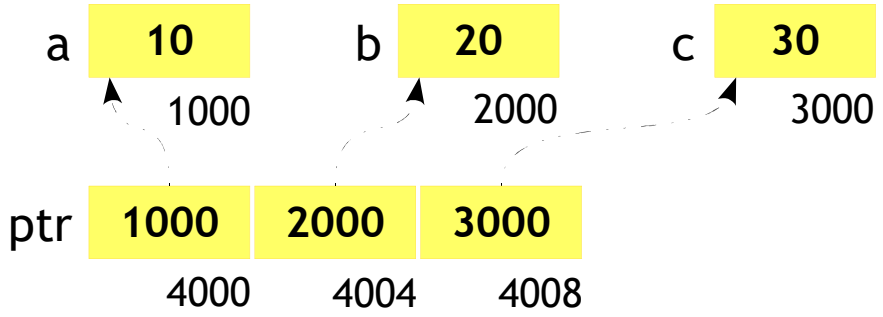


Embedded C

Pointers - Array of pointers

Example

```
#include <stdio.h>

void print_array(int **p)
{
     int i;

     for (i = 0; i < 3; i++)
     {
          printf("%d ", *p[i]);
          printf("at %p\n", p[i]);
     }
}

int main()
{
     int a = 10;
     int b = 20;
     int c = 30;
     int *ptr[3] = {&a, &b, &c};

     print_array(ptr);

     return 0;
}
```

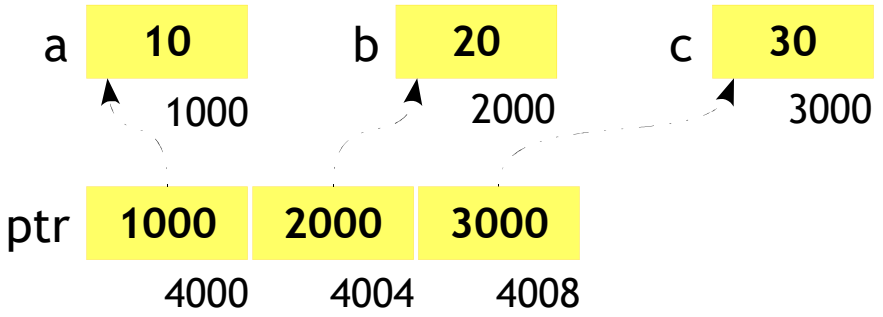


Embedded C

Pointers – Array of pointers

Example

```
#include <stdio.h>

void print_array(int **p)
{
     int i;

     for (i = 0; i < 3; i++)
     {
          printf("%d ", *p[i]);
          printf("at %p\n", p[i]);
     }
}

int main()
{
     int a = 10;
     int b = 20;
     int c = 30;
     int *ptr[3] = {&a, &b, &c};

     print_array(ptr);

     return 0;
}
```

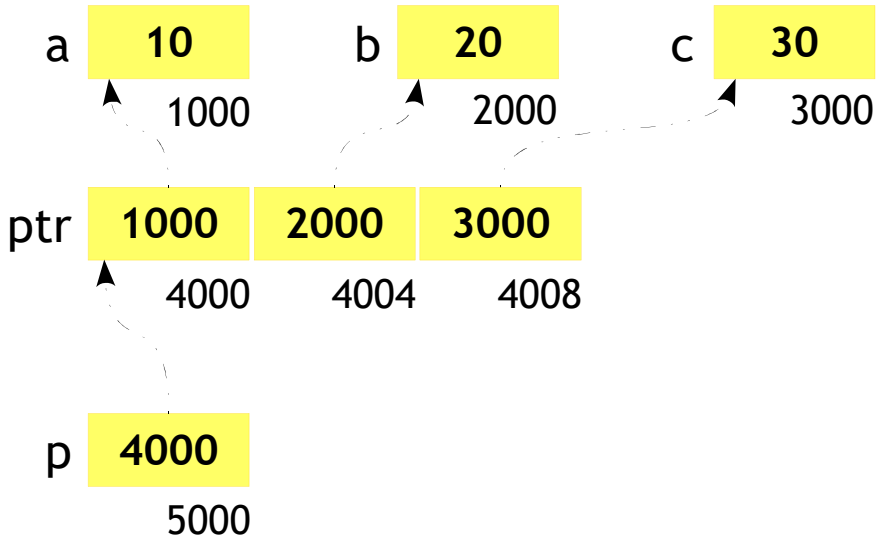


Embedded C

Pointers – Array of pointers

Example

```
#include <stdio.h>

void print_array(int **p)
{
     int i;

     for (i = 0; i < 3; i++)
     {
          printf("%d ", *p[i]);
          printf("at %p\n", p[i]);
     }
}

int main()
{
     int a = 10;
     int b = 20;
     int c = 30;
     int *ptr[3] = {&a, &b, &c};

     print_array(ptr);

     return 0;
}
```

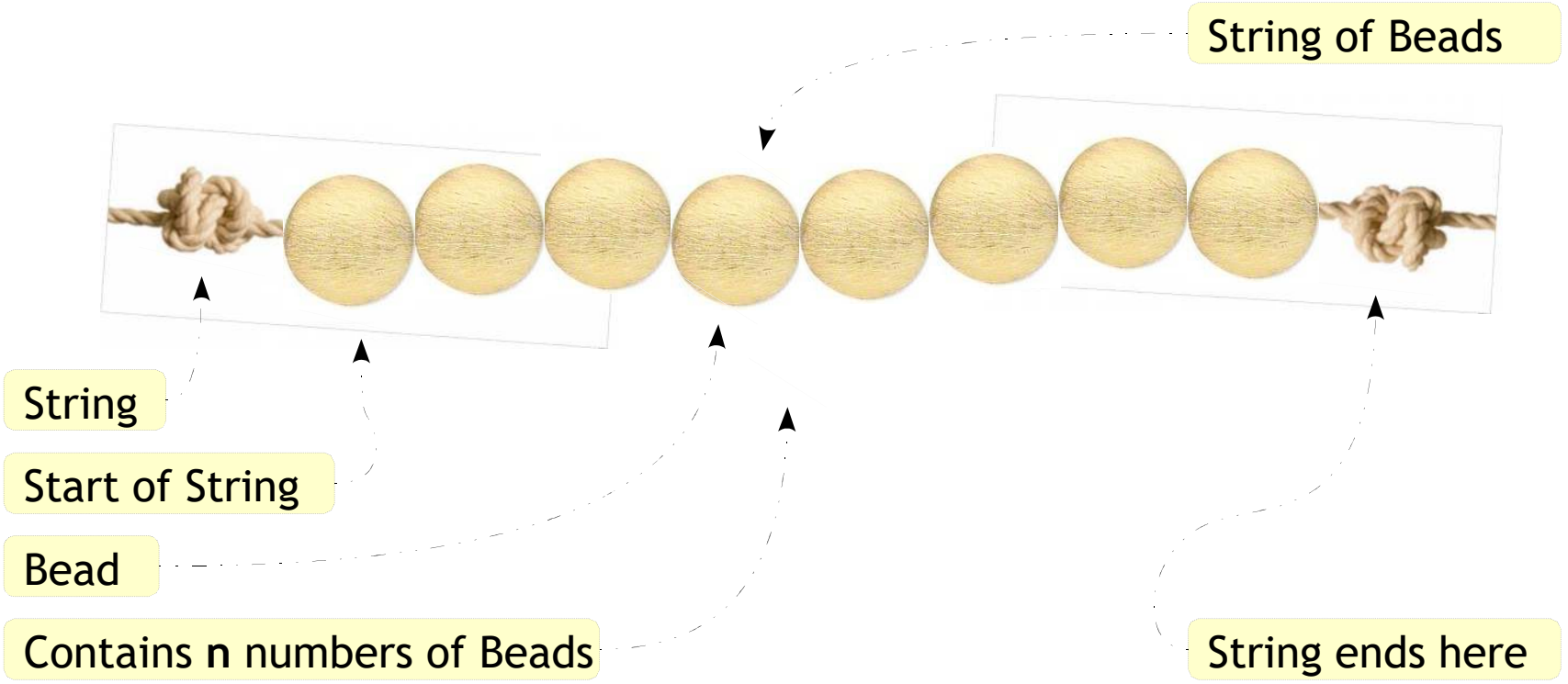


Embedded C

Strings

A set of things tied or threaded together on a thin cord.

Source: Google



Embedded C
Strings

- Contiguous sequence of characters
- Easily stores ASCII and its extensions
- End of the string is marked with a special character, the null character '\0'
- '\0' is implicit in strings enclosed with “”
- Example

“You know, now this is what a string is!”

Embedded C

Strings - Initializations

Examples

```
char char_array[5] = {'H', 'E', 'L', 'L', 'O'};
```
← Character Array

```
char str[6] = {'H', 'E', 'L', 'L', 'O', '\0'};
```
← String

```
char str[] = {'H', 'E', 'L', 'L', 'O', '\0'};
```
← Valid

```
char str[6] = {“H”, “E”, “L”, “L”, “O”};
```
← Invalid

```
char str[6] = {“H” “E” “L” “L” “O”};
```
← Valid

```
char str[6] = {“HELLO”};
```
← Valid

```
char str[6] = “HELLO”;
```
← Valid

```
char str[] = “HELLO”;
```
← Valid

```
char *str = “HELLO”;
```
← Valid

Embedded C
Strings - Size

Examples

```
#include <stdio.h>

int main()
{
    char char_array_1[5] = {'H', 'E', 'L', 'L', 'O'};
    char char_array_2[] = "Hello";

    sizeof(char_array_1);
    sizeof(char_array_2);

    return 0;
}
```

The size of the array Is calculated So,

5, 6

is

```
int main()
{
    char *str = "Hello";

    sizeof(str);

    return 0;
}
```

The size of pointer always constant so,

4 (32 Bit Sys)

Embedded C

Strings - Size

Example

```
#include <stdio.h>

int main()
{
    if (sizeof("Hello" "World") == sizeof("Hello") + sizeof("World"))
    {
        printf("WoW\n");
    }
    else
    {
        printf("HuH\n");
    }


    return 0;
}
```

Embedded C

Strings - Manipulations

Examples

```
char str1[6] = "Hello";
char str2[6];

str2 = "World";
```

Not possible to assign a string to a array since its a constant pointer

```
char *str3 = "Hello";
char *str4;

str4 = "World";
```

Possible to assign a string to a pointer since its variable

```
str1[0] = 'h';
```

Valid. str1 contains "hello"

```
str3[0] = 'w';
```

Invalid. str3 might be stored in read only section. Undefined behaviour

Embedded C

Strings - Sharing

Example

```
#include <stdio.h>

int main()
{
    char *str1 = "Hello";
    char *str2 = "Hello";

    if (str1 == str2)
    {
        printf("Hoo. They share same space\n");
    }
    else
    {
        printf("No. They are in different space\n");
    }

    return 0;
}
```

Embedded C

Strings – Library Functions

Purpose	Prototype	Return Values
Length	**size_t strlen(const char *str)**	String Length
Compare	**int strcmp(const char *str1, const char *str2)**	str1 < str2 → < 0 str1 > str2 → > 0 str1 = str2 → = 0
Copy	**char *strcpy(char *dest, const char *src)**	Pointer to dest
Check String	**char *strstr(const char *haystack, const char *needle)**	Pointer to the beginning of substring
Check Character	**char *strchr(const char *s, int c)**	Pointer to the matched char else NULL
Merge	**char *strcat(char *dest, const char *src)**	Pointer to dest

Embedded C
Memory Segments

The Linux OS is divided into two major sections
- User Space
- Kernel Space

The user programs cannot access the kernel space. If done will lead to segmentation violation

Let us concentrate on the user space section here

Embedded C
Memory Segments

User Space

P_1

P_2

P_3

.
.
.

P_{n-1}

P_n

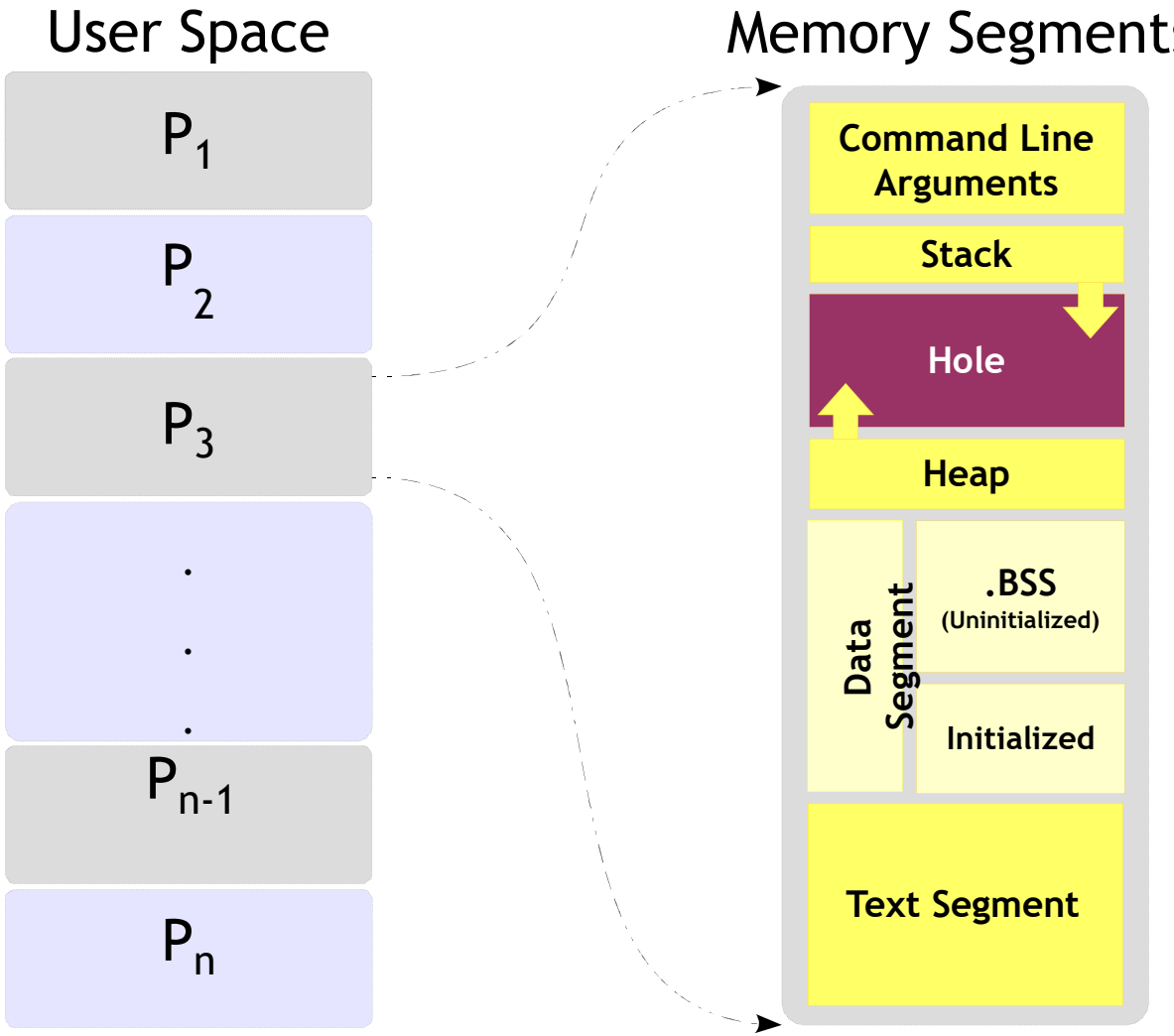
The User space contains many processes

Every process will be scheduled by the kernel

Each process will have its memory layout discussed in next slide

Embedded C

Memory Segments



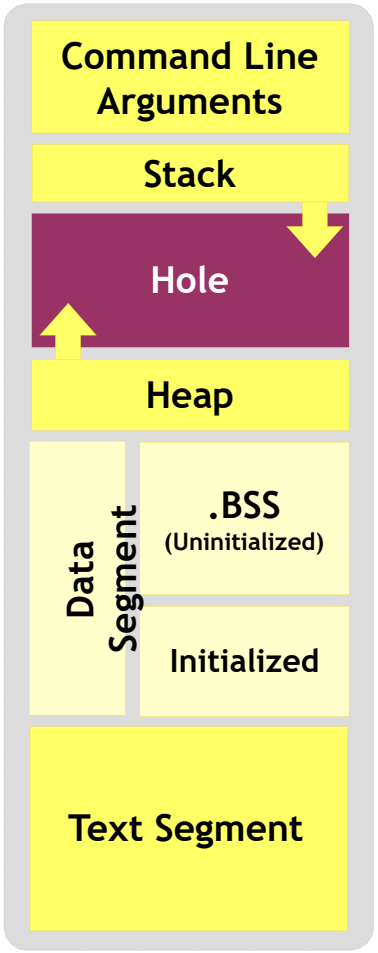
The memory segment of a program contains four major areas.

- Text Segment
- Stack
- Data Segment
- Heap

Embedded C

Memory Segments – Text Segment

Memory Segments

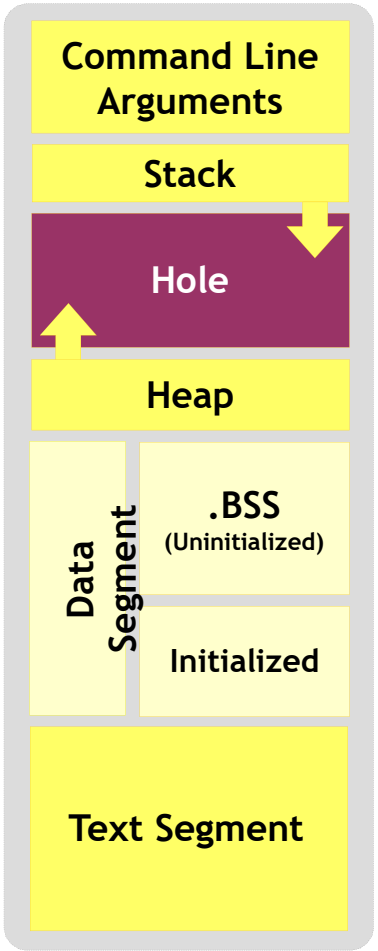


- Also referred as Code Segment
- Holds one of the section of program in object file or memory
- In memory, this is place below the heap or stack to prevent getting over written
- Is a read only section and size is fixed

Embedded C

Memory Segments – Data Segment

Memory Segments

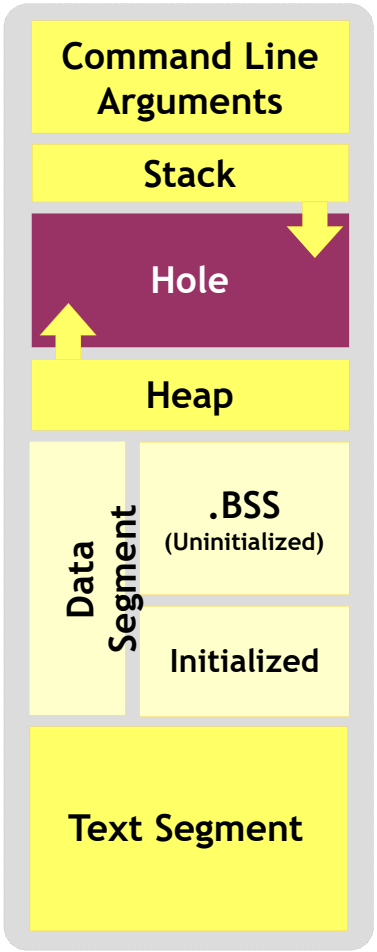


- Contains 2 sections as initialized and uninitialized data segments
- Initialized section is generally called as Data Segment
- Uninitialized section is referred as BSS (Block Started by Symbol) usually filled with 0s

Embedded C

Memory Segments – Data Segment

Memory Segments



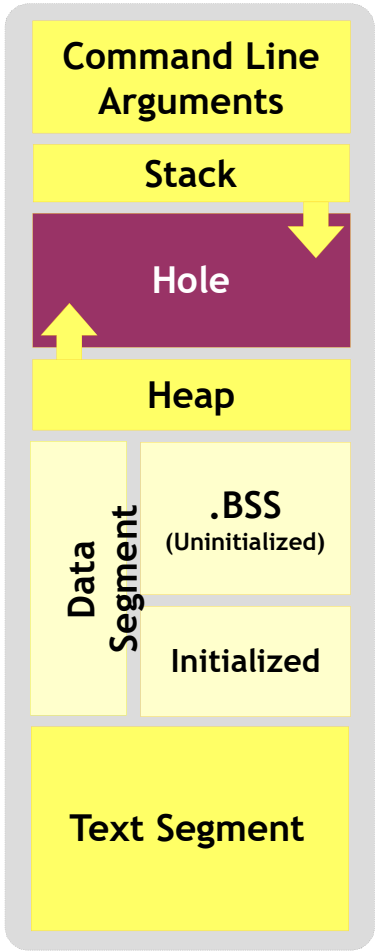
Dynamic memory allocation takes place here

Begins at the end of BSS and grows upward from there

Embedded C

Memory Segments – Stack Segment

Memory Segments

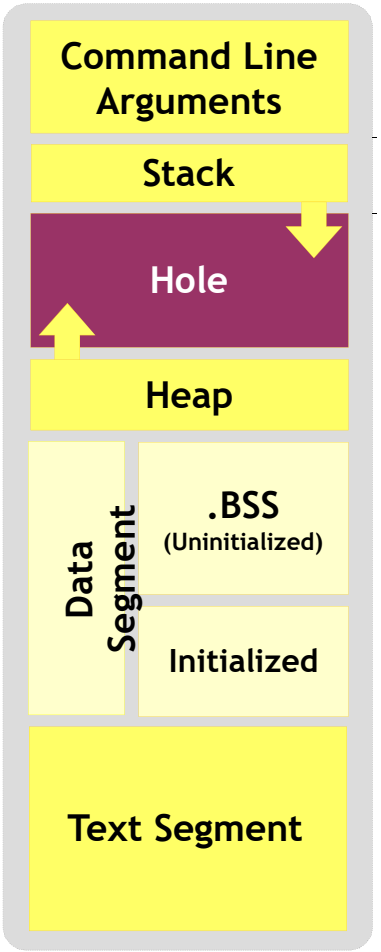


- Adjoins the heap area and grow in opposite area of heap when stack and heap pointer meet (Memory Exhausted)
- Typicallly loaded at the higher part of memory memory
- A “stack pointer” register tracks the top of the stack; it is adjusted each time a value is “pushed” onto the stack
- The set of values pushed for one function call is termed a “stack frame”

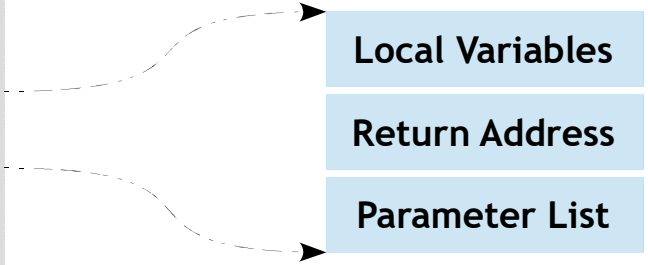
Embedded C

Memory Segments – Stack Segment

Memory Segments



Stack Frame



A stack frame contain at least of a return address

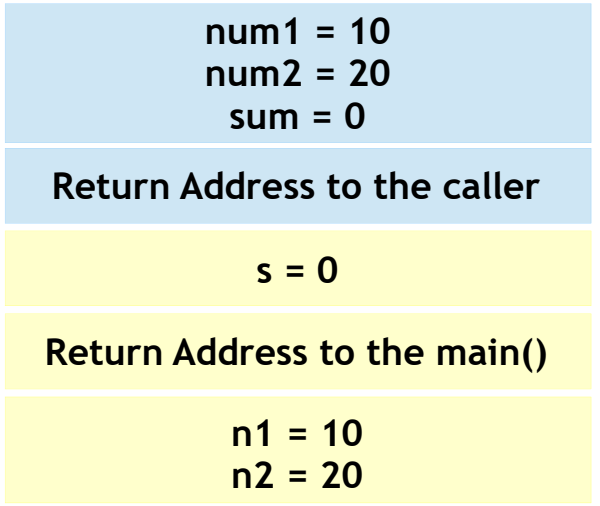
Embedded C

Memory Segments – Stack Frame

```c
#include <stdio.h>

int main()
{
    int num1 = 10, num2 = 20;
    int sum = 0;

    sum = add_numbers(num1, num2);
    printf("Sum is %d\n", sum);

    return 0;
}
```

```c
int add_numbers(int n1, int n2)
{
    int s = 0;

    s = n1 + n2;

    return s;
}
```

Stack Frame



Embedded C
Memory Segments - Runtime

- Run-time memory includes four (or more) segments
  - Text area: program text
  - Global data area: global & static variables
  - Allocated during whole run-time
- Stack: local variables & parameters
  - A stack entry for a functions
  - Allocated (pushed) - When entering a function
  - De-allocated (popped) - When the function returns
- Heap
  - Dynamic memory
  - Allocated by malloc()
  - De-allocated by free()

Embedded C

Storage Classes

Storage Class	Scope	Lifetime	Memory Allocation
auto	Within the block / Function	Till the end of the block / function	Stack
register	Within the block / Function	Till the end of the block / function	Register
static local	Within the block / Function	Till the end of the program	Data Segment
static global	File	Till the end of the program	Data segment
extern	Program	Till the end of the program	Data segment

Embedded C
Storage Classes

Example

```c
#include <stdio.h>

int global_1;
int global_2 = 10;

static int global_3;
static int global_4 = 10;

int main()
{
    int local_1;
    static int local_1;
    static int local_2 = 20;

    register int i;
    for (i = 0; i < 0; i++)
    {
        /* Do Something */
    }

    return 0;
}
```

Variable	Storage Class	Memory Allocation
global_1	No	.BSS
global_2	No	Initialized data segment
global_3	Static global	.BSS
global_4	Static global	Initialized data segment
local_1	auto	stack
local_2	Static local	.BSS
local_3	Static local	Initialized data segment
iter	Register	Registers

Embedded C

Declaration

```
extern int num1;
extern int num1;

int main();

int main()
{
    int num1, num2;
    char short_opt;

    ...
}
```

Declaration specifies type to the variables

Its like an announcement and hence can be made 1 or more times

Declaration about num1

Declaration about num1 yet again!!

Declaration about main function

Embedded C

Storage Classes - extern

file1.c

```c
#include <stdio.h>

int num;

int main()
{
    while (1)
    {
        num++;
        func_1();
        sleep(1);
        func_2();
    }

    return 0;
}
```

file2.c

```c
#include <stdio.h>

extern int num;

int func_1()
{
    printf("num is %d from file2\n", num);

    return 0;
}
```

file3.c

```c
#include <stdio.h>

extern int num;

int func_2()
{
    printf("num is %d from file3\n", num);

    return 0;
}
```

Embedded C
Preprocessor

- One of the step performed before compilation
- Is a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation
- Instructions given to preprocessor are called preprocessor directives and they begin with “#” symbol
- Few advantages of using preprocessor directives would be,
  - Easy Development
  - Readability
  - Portability

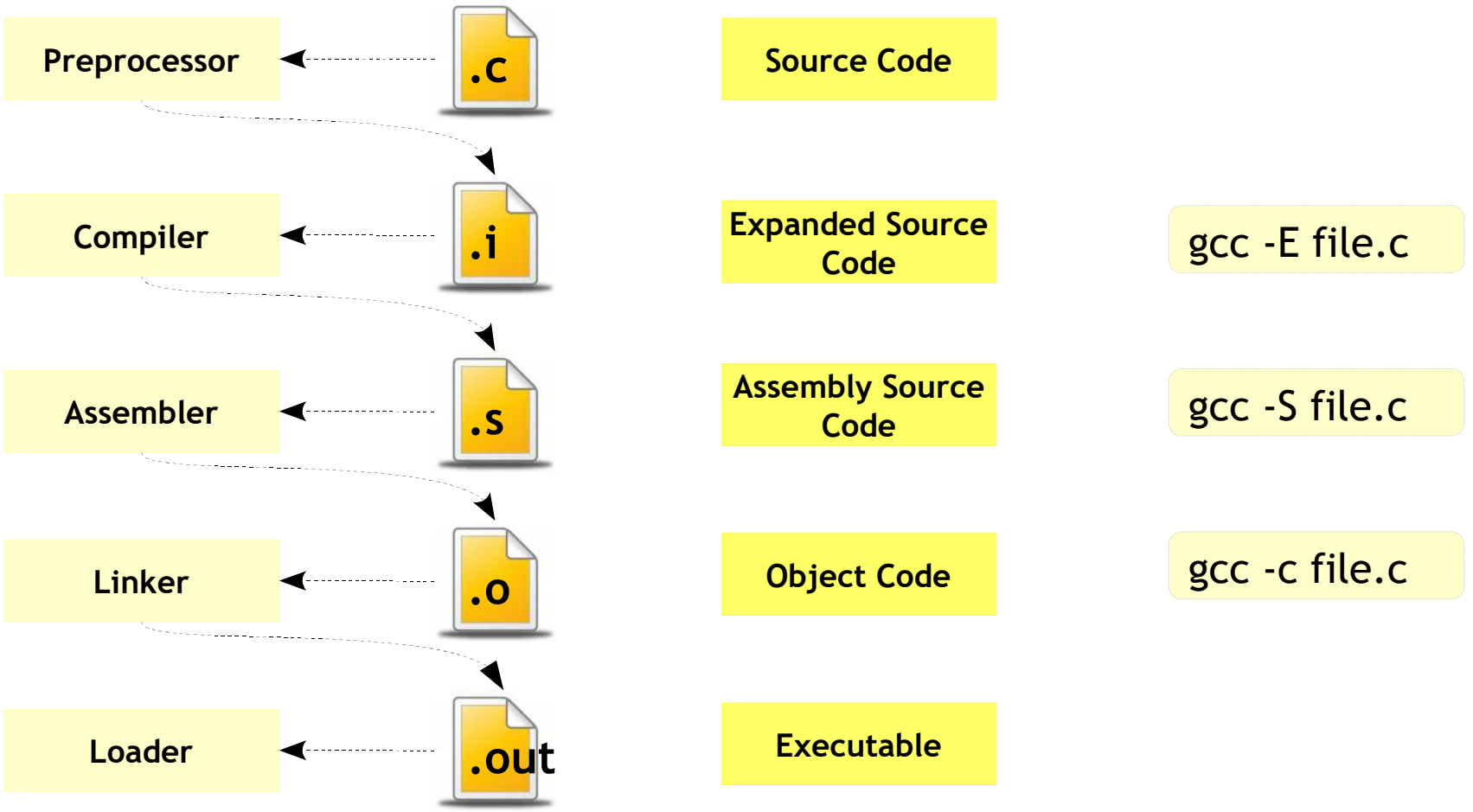
Embedded C

Preprocessor – Compilation Stages

- Before we proceed with preprocessor directive let's try to understand the stages involved in compilation
- Some major steps involved in compilation are
  - Pre-processing
  - Compilation
  - Assembly
  - Linking
- The next slide provide the flow of these stages

Embedded C

Preprocessor – Compilation Stages



gcc -save-temps file.c would generate all intermediate files

Embedded C

Preprocessor – Directives

#include
#define
#undef
#ifdef
#ifndef
#else
#endif
#if
#else
#elif
#error
#warning
#line
#pragma
#
##

Embedded C
Preprocessor – Header Files

- A header file is a file containing C declarations and macro definitions to be shared between several source files.
- Has to be included using C preprocessing directive '**#include**'
- Header files serve two purposes.
  - Declare the interfaces to parts of the operating system by supplying the definitions and declarations you need to invoke system calls and libraries.
  - Your own header files contain declarations for interfaces between the source files of your program.

Embedded C

Preprocessor – Header Files vs Source Files

VS

- Declarations
- Sharable/reusable
  - #defines
  - Datatypes
- Used by more than 1 file

- Function and variable definitions
- Non sharable/reusable
  - #defines
  - Datatypes

Embedded C

Preprocessor – Header Files - Syntax

Syntax

```
#include <file.h>
```

- System header files
- It searches for a file named ***file*** in a standard list of system directories

Syntax

```
#include "file.h"
```

- Local (your) header files
- It searches for a file named ***file*** first in the directory containing the current file, then in the quote directories and then the same directories used for <file>

Embedded C

Preprocessor – Header Files - Operation

file2.c

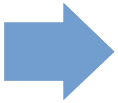
```
char *test(void)
{
    static char *str = "Hello";

    return str;
}
```

file2.h

```
char *test(void);
```

file1.c

```
int num;

#include "file2.h"

int main()
{
    puts(test());

    return 0;
}
```



```
int num;

char *test(void);

int main()
{
    puts(test());

    return 0;
}
```

Embedded C

Preprocessor – Header Files – Search Path

- On a normal Unix system GCC by default will look for headers requested with #include <file> in:
  - /usr/local/include
  - libdir/gcc/target/version/include
  - /usr/target/include
  - /usr/include
- You can add to this list with the -I <*dir*> command-line option

`gcc -print-prog-name=cc1` -v would the search path info

Embedded C

Preprocessor – Header Files – Once-Only

- If a header file happens to be included twice, the compiler will process its contents twice causing an error
- E.g. when the compiler sees the same structure definition twice
- This can be avoided like

Example

```
#ifndef NAME
#define NAME

/* The entire file is protected */

#endif
```

Embedded C

Preprocessor – Macro – Object-Like

- An object-like macro is a simple identifier which will be replaced by a code fragment
- It is called object-like because it looks like a data object in code that uses it.
- They are most commonly used to give symbolic names to numeric constants

Syntax

```
#define SYMBOLIC_NAME      CONSTANTS
```

Example

```
#define BUFFER_SIZE        1024
```

Embedded C

Preprocessor – Macro – Arguments

- Function-like macros can take arguments, just like true functions
- To define a macro that uses arguments, you insert parameters between the pair of parentheses in the macro definition that make the macro function-like

Example

```
#include <stdio.h>

#define SET_BIT(num, pos)      (num | (1 << pos))

int main()
{
    SET_BIT(0, 2);

    return 0;
}
```

Embedded C

Preprocessor – Macro – Multiple Lines

- You may continue the definition onto multiple lines, if necessary, using backslash-newline.
- When the macro is expanded, however, it will all come out on one line

Example

```
#include <stdio.h>

#define SWAP(a, b)                  \
{                                   \
    int temp = a;                   \
    a = b;                          \
    b = temp;                       \
}

int main()
{
    int num1 = 10, num2= 20;

    SWAP(num1, num2);

    printf("%d %d\n", num1, num2);

    return 0;
}
```

Embedded C

Preprocessor – Macro – Stringification

Example

```
#include <stdio.h>

#define WARN_IF(EXP)                                    \
do                                                      \
{                                                       \
    x--;                                                \
    if (EXP)                                            \
    {                                                   \
        fprintf(stderr, "Warning: " #EXP "\n");         \
    }                                                   \
} while (x);

int main()
{
    int x = 5;

    WARN_IF(x == 0);

    return 0;
}
```

- You can convert a macro argument into a string constant by adding #

Embedded C

Preprocessor – Macro – Concatenation

Example

```c
#include <stdio.h>

#define CAT(x)              (x##_val)

int main()
{
    int int_val = 4;
    float float_val = 2.54;

    printf("int val = %d\n", CAT(int));
    printf("float val = %f\n", CAT(float));

    return 0;
}
```

Embedded C

Preprocessor – Macro – Standard Predefined

- Several object-like macros are predefined; you use them without supplying their definitions.
- Standard are specified by the relevant language standards, so they are available with all compilers that implement those standards

Example

```
#include <stdio.h>

int main()
{

    printf("Program: \"%s\" ", __FILE_);
    printf("was compiled on %s at %s. ", __DATE__, __TIME__);
    printf("This print is from Function: \"%s\" at line %d\n", _func_, __LINE_);

    return 0;
}
```

Embedded C

Preprocessor – Conditional Compilation

- A conditional is a directive that instructs the preprocessor to select whether or not to include a chunk of code in the final token stream passed to the compiler
- Preprocessor conditionals can test arithmetic expressions, or whether a name is defined as a macro, or both simultaneously using the special defined operator
- A conditional in the C preprocessor resembles in some ways an if statement in C with the only difference being it happens in compile time
- Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

Embedded C

Preprocessor – Conditional Compilation

- There are three general reasons to use a conditional.
  - A program may need to use different code depending on the machine or operating system it is to run on
  - You may want to be able to compile the same source file into two different programs, like one for debug and other as final
  - A conditional whose condition is always false is one way to exclude code from the program but keep it as a sort of comment for future reference

Embedded C

Preprocessor – Conditional Compilation - ifdef

Syntax

```
#ifdef MACRO

/* Controlled Text */

#endif
```

Embedded C

Preprocessor – Conditional Compilation - defined

Syntax

```
#if defined (ERROR) && (WARNING)

/* Controlled Text */

#endif
```

Embedded C

Preprocessor – Conditional Compilation - if

Syntax

```
#if expression

/* Controlled Text */

#endif
```

Embedded C

Preprocessor – Conditional Compilation - else

Syntax

```
#if expression

/* Controlled Text if true */

#else

/* Controlled Text if false */

#endif
```

Embedded C

Preprocessor – Conditional Compilation - elif

Syntax

```
#if DEBUG_LEVEL == 1

/* Controlled Text*/

#elif  DEBUG_LEVEL == 2

/* Controlled Text */

#else

/* Controlled Text */

#endif
```

Embedded C

Preprocessor – Conditional Com... – Deleted Code

Syntax

```
#if 0

/* Deleted code while compiling */
/* Can be used for nested code comments */
/* Avoid for general comments */

#endif
```

Embedded C

Preprocessor – Diagnostic

- The directive '#error' causes the preprocessor to report a fatal error. The tokens forming the rest of the line following '#error' are used as the error message
- The directive '#warning' is like '#error', but causes the preprocessor to issue a warning and continue preprocessing. The tokens following '#warning' are used as the warning message

Embedded C

User Defined Datatypes (UDT)

- Sometimes it becomes tough to build a whole software that works only with integers, floating values, and characters.
- In circumstances such as these, you can create your own data types which are based on the standard ones
- There are some mechanisms for doing this in C:
  - Structures
  - Unions
  - Typedef
  - Enums
- Hoo!!, let's not forget our old friend **_r_a_** which is a user defined data type too!!.

Embedded C

UDTs - Structures

- A complex data type declaration that defines a physically grouped list of variables to be placed under one name in a block of memory
- Unlike arrays, structures allows different data types in the allocated memory region
- Generally useful whenever a lot of data needs to be grouped together
- Allows us to have OOPS concepts in C,
  - Data first
  - Data centric (Manipulations on
  - data) Better Design

Embedded C

UDTs - Structures

Syntax

```
struct StructureName
{
    /* Group of data types */
};
```

- If we consider the Student as an example, The admin should have at least some important data like name, ID and address.
- So we if create a structure for the above requirement, it would look like,

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150]
};
```

Embedded C

UDTs – Structures – Declartion and definition

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;

    return 0;
}
```

- Name of the **datatype**. Note it's **struct Student** and not Student
- Are called as **fields** or **members** of of the structure
- Declaration **ends** here
- The memory is not yet allocated!!
- **s1** is a **variable** of **type struct Student**
- The memory is allocated now

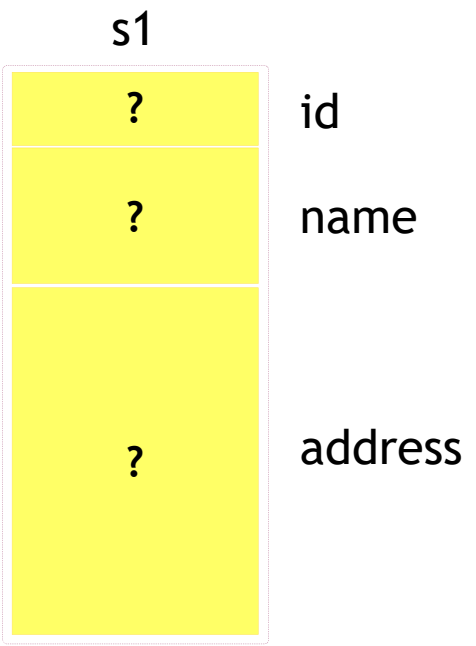
Embedded C

UDTs – Structures – Memory Layout

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;

    return 0;
}
```

- What does s1 contain?
- How can we draw it's memory layout?



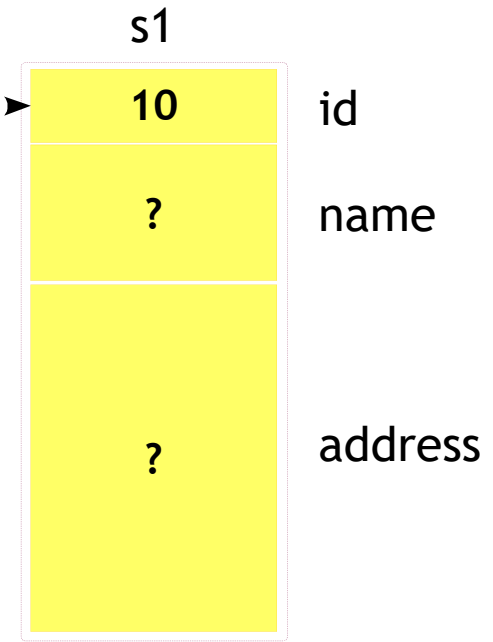
Embedded C

UDTs – Structures - Access

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;

    s1.id = 10;

    return 0;
}
```

- How to write into id now?
- It's by using “.” (Dot) operator



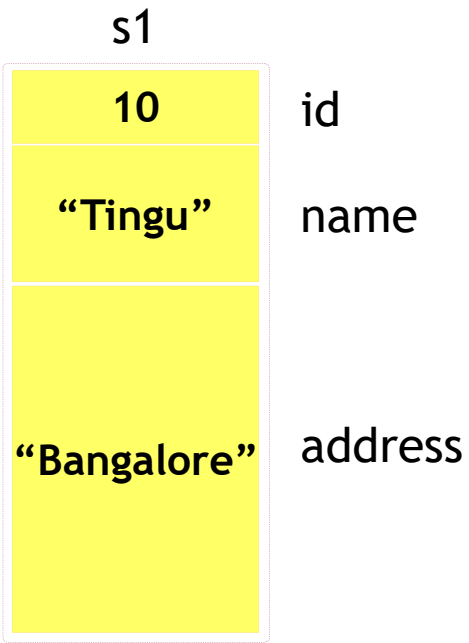
- Now please assign the name for s1

Embedded C

UDTs – Structures - Initialization

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    return 0;
}
```

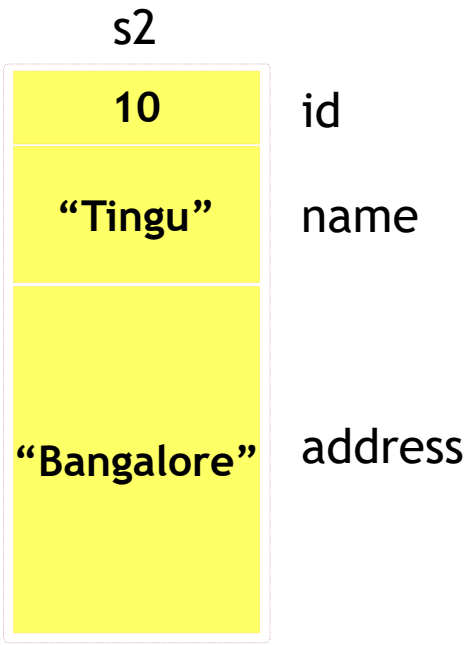


Embedded C

UDTs – Structures - Copy

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};
    struct Student s2;

    s2 = s1;

    return 0;
}
```



Structure name does not represent its address. (No correlation with arrays)

Embedded C

UDTs – Structures - Address

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1 = {10, "Tingu", "Bangalore"};

    printf("Structure starts at %p\n", &s1);
    printf("Member id is at %p\n", &s1.id);
    printf("Member name is at %p\n", &s1.name);
    printf("Member address is at %p\n", &s1.address);

    return 0;
}
```

Embedded C

UDTs – Structures - Pointers

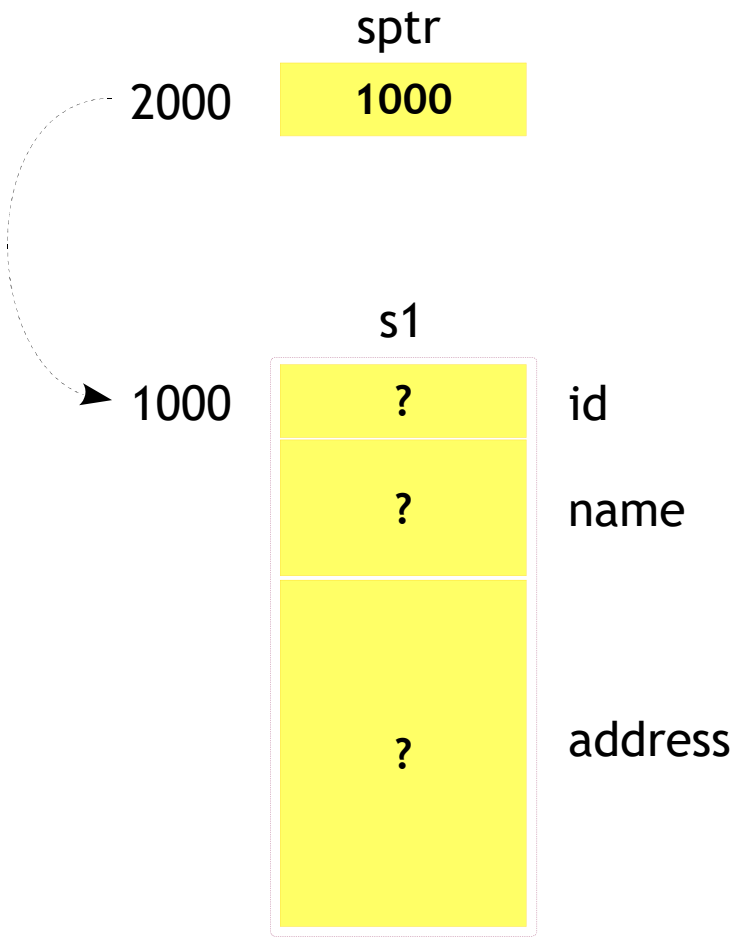
- Pointers!!!. Not again ;). Fine don't worry, not a big deal
- But do you have any idea how to create it?
- Will it be different from defining them like in other data types?

Embedded C

UDTs – Structures - Pointer

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;
    struct Student *sptr = &s1;

    return 0;
}
```

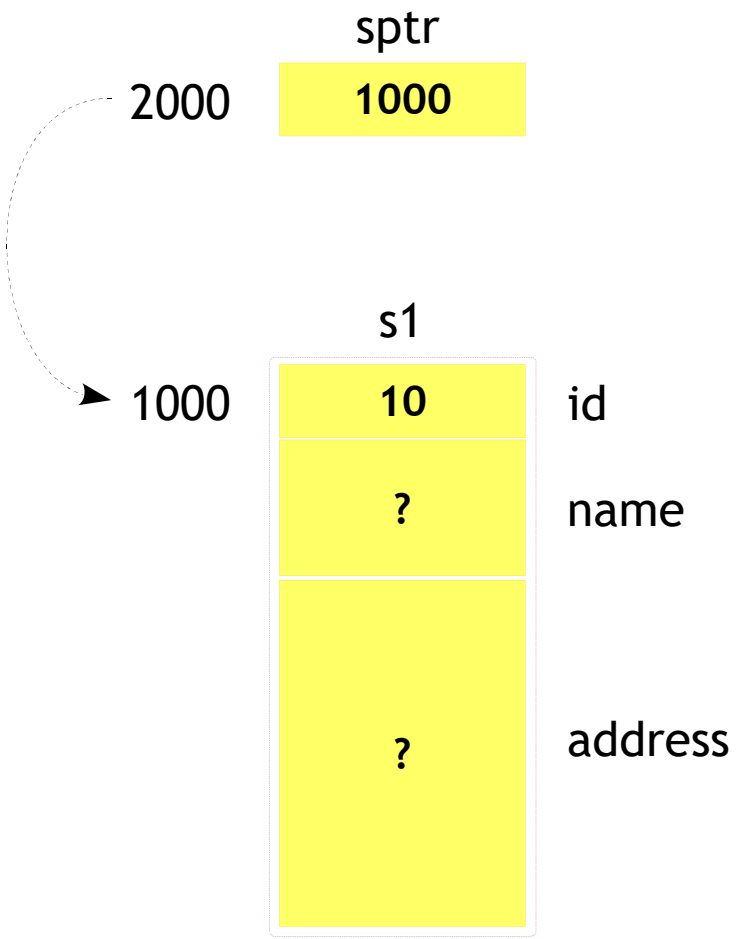


Embedded C

UDTs – Structures – Pointer - Access

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;
    struct Student *sptr = &s1;

    (*sptr).id = 10;

    return 0;
}
```

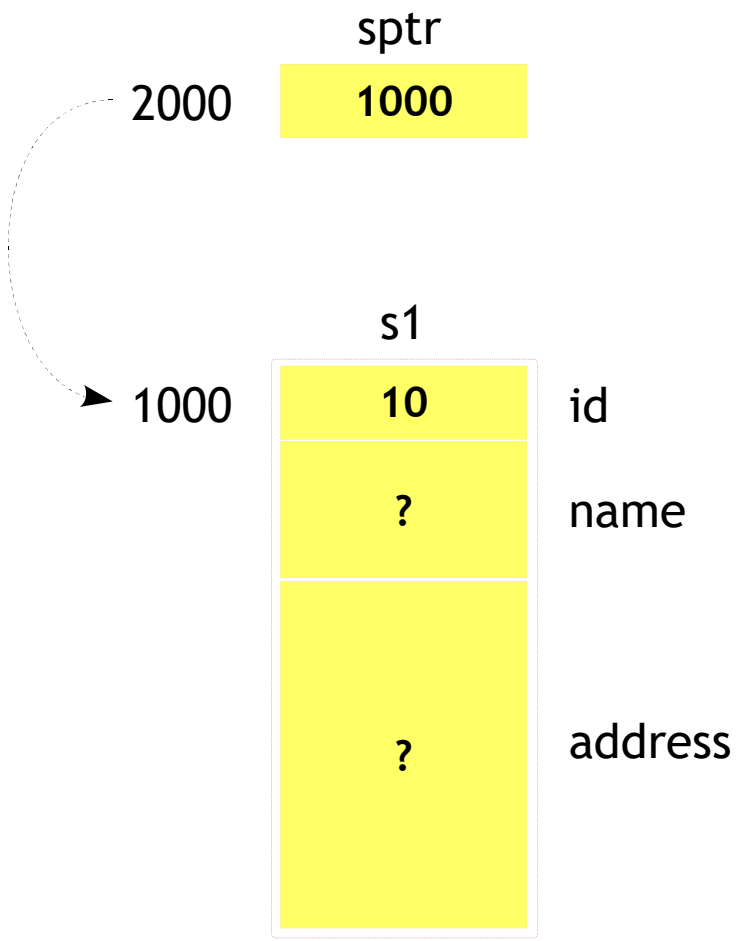


Embedded C

UDTs – Structures – Pointer – Access – Arrow

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

int main()
{
    struct Student s1;
    struct Student *sptr = &s1;

    sptr->id = 10;

    return 0;
}
```



Note: we can access the structure pointer as seen in the previous slide. The Arrow operator is just convenience and frequently used

Embedded C

UDTs – Structures - Functions

- The structures can be passed as parameter and can be returned from a function
- This happens just like normal datatypes.
- The parameter passing can have two methods again as normal
  - Pass by value
  - Pass by reference

Embedded C

UDTs – Structures – Functions – Pass by Value

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150];
};

void data(struct Student s)
{
    s.id = 10;
}

int main()
{
    struct Student s1;

    data(s1);

    return 0;
}
```

Not recommended on larger structures

Embedded C

UDTs – Structures – Functions – Pass by Reference

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150]
};

void data(struct Student *s)
{
    s->id = 10;
}

int main()
{
    struct Student s1;

    data(&s1);

    return 0;
}
```

Recommended on larger structures

Embedded C

UDTs - Structures - Functions - Return

Example

```
struct Student
{
    int id;
    char name[30];
    char address[150]
};

struct Student data(struct Student s)
{
    s.id = 10;

    return s;
}

int main()
{
    struct Student s1;

    s1 = data(s1);

    return 0;
}
```

Embedded C

UDTs – Structures – Padding

- Adding of few extra useless bytes (in fact skip address) in between the address of the members are called structure padding.
- What!!?, wasting extra bytes!!, Why?
- This is done for Data Alignment.
- Now!, what is data alignment and why did this issue suddenly arise?
- No its is not sudden, it is something the compiler would internally while allocating memory.
- So let's understand data alignment in next few slides

Embedded C
Data Alignment

- A way the data is arranged and accessed in computer memory.
- When a modern computer reads from or writes to a memory address, it will do this in word sized chunks (4 bytes in 32 bit system) or larger.
- The main idea is to increase the efficiency of the CPU, while handling the data, by arranging at a memory address equal to some multiple of the word size
- So, Data alignment is an important issue for all programmers who directly use memory.

Embedded C
Data Alignment

- If you don't understand data and its address alignment issues in your software, the following scenarios, in increasing order of severity, are all possible:
  - Your software will run slower.
  - Your application will lock up.
  - Your operating system will crash.
  - Your software will silently fail, yielding incorrect results.

Embedded C
Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

0	'A'
1	78
2	56
3	34
4	12
5	?
6	?
7	?

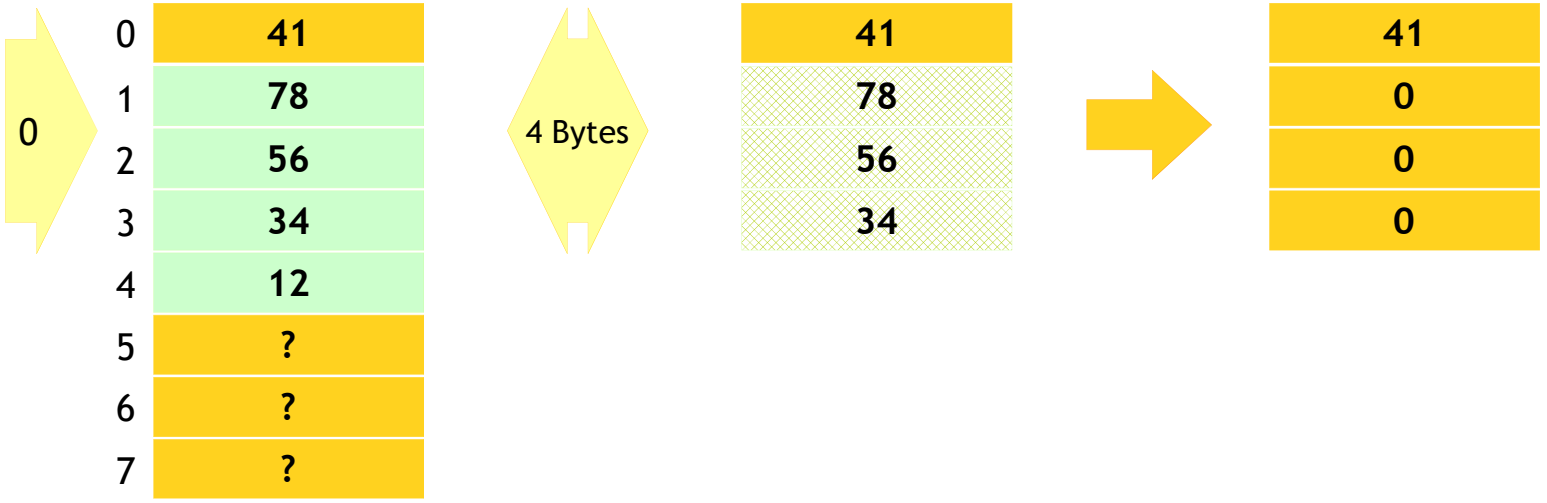
- Lets consider the code as given
- The memory allocation we expect would be like shown in figure
- So lets see how the CPU tries to access these data in next slides

Embedded C
Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the character by the CPU will be like shown below

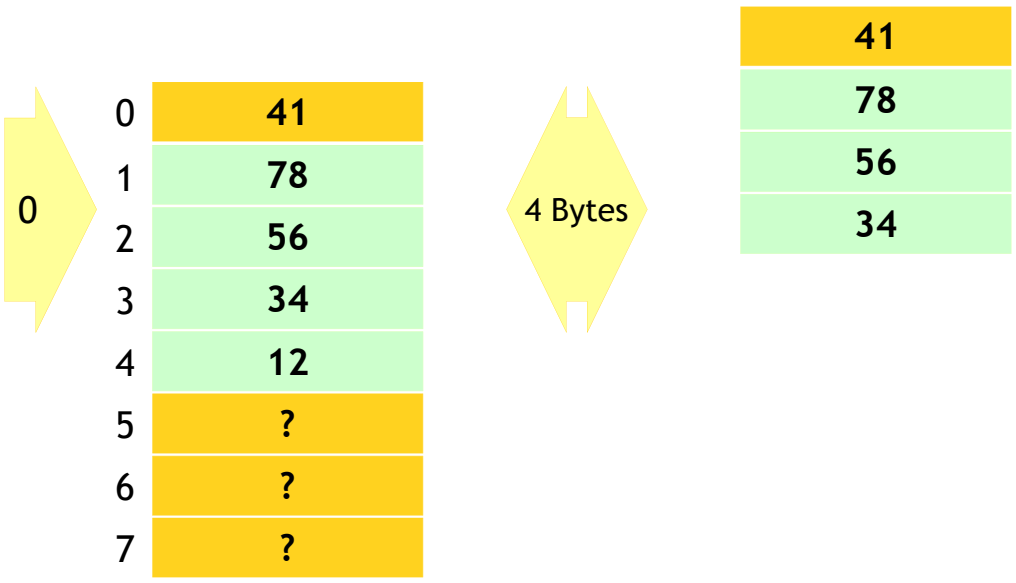


Embedded C
Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below

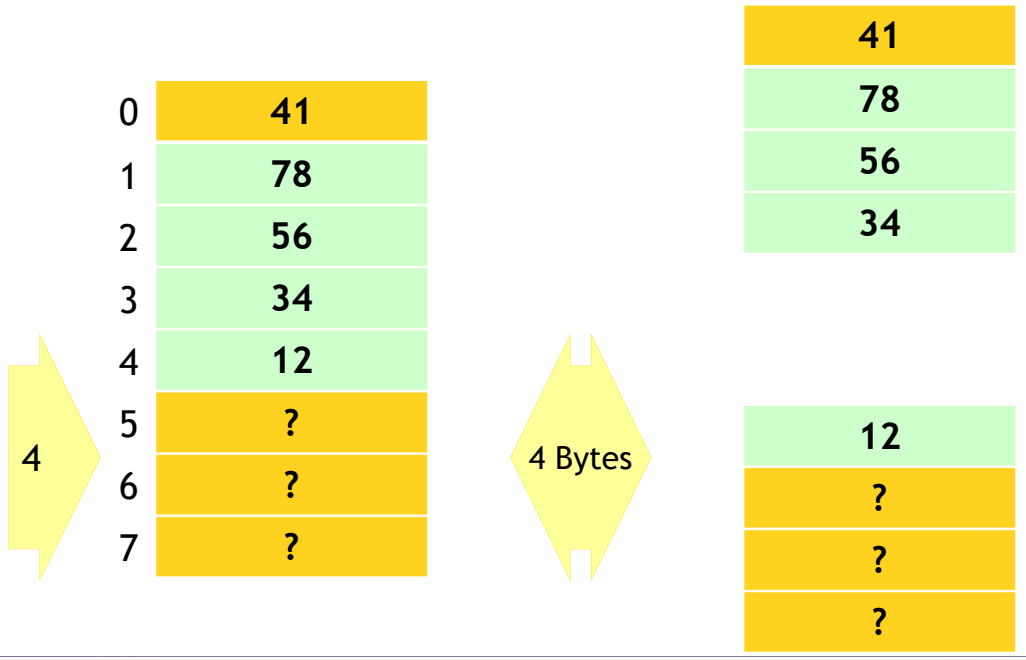


Embedded C
Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below



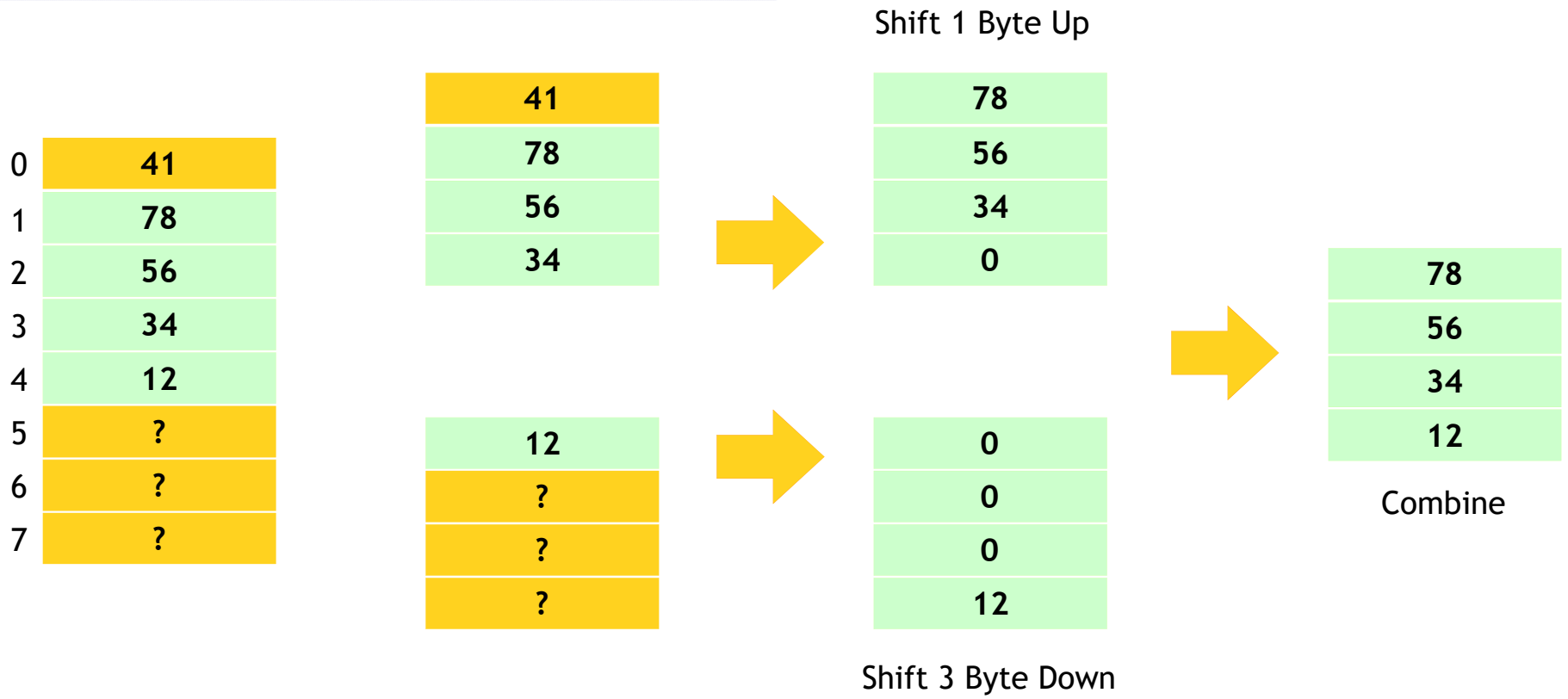
Embedded C

Data Alignment

Example

```
int main()
{
    char ch = 'A';
    int num = 0x12345678;
}
```

- Fetching the integer by the CPU will be like shown below



Embedded C

UDTs – Structures – Data Alignment - Padding

- Because of the data alignment issue, structures uses padding between its members if the don't fall under even address.
- So if we consider the following structure the memory allocation will be like shown in below figure

Example

```
struct Test
{
    char ch1;
    int num;
    char ch2;
}
```

0	ch1
1	pad
2	pad
3	pad
4	num
5	num
6	num
7	num
8	ch2
9	pad
A	pad
B	pad

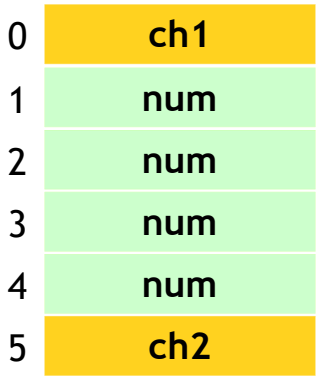
Embedded C

UDTs – Structures – Data Alignment - Padding

- You can instruct the compiler to modify the default padding behavior using #pragma pack directive

Example

```
#pragma pack(1)

struct Test
{
    char ch1;
    int num;
    char ch2;
}
```



Embedded C

UDTs – Structures – Bit Fields

- The compiler generally gives the memory allocation in multiples of bytes, like 1, 2, 4 etc.,
- What if we want to have freedom of having getting allocations in bits?!.
- This can be achieved with bit fields.
- But note that
  - The minimum memory allocation for a bit field member would be a byte that can be broken in max of 8 bits
  - The maximum number of bits assigned to a member would depend on the length modifier
  - The default size is equal to word size

Embedded C

UDTs – Structures – Bit Fields

Example

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};
```

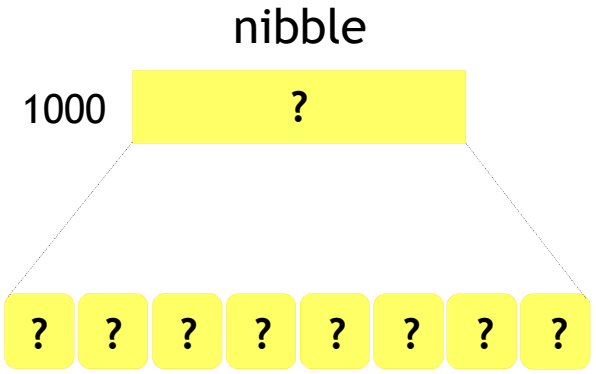
- The above structure divides a char into two nibbles
- We can access these nibbles independently

Embedded C

UDTs - Structures - Bit Fields

Example

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0xA;
    nibble.lower = 0x2;

    return 0;
}
```

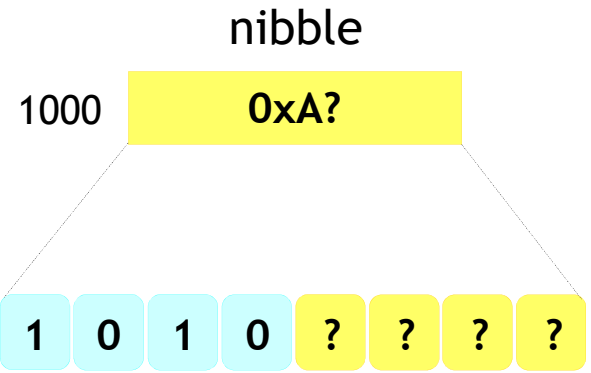


Embedded C

UDTs – Structures – Bit Fields

Example

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0xA;
    nibble.lower = 0x2;

    return 0;
}
```

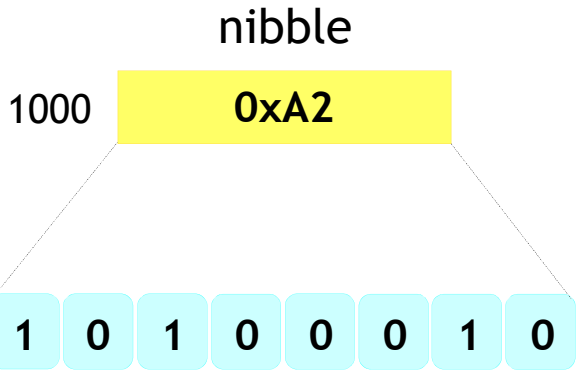


Embedded C

UDTs – Structures – Bit Fields

Example

```
struct Nibble
{
    unsigned char lower    : 4;
    unsigned char upper    : 4;
};

int main()
{
    struct Nibble nibble;

    nibble.upper = 0xA;
    nibble.lower = 0x2;

    return 0;
}
```



Embedded C

UDTs – Unions

- Like structures, unions may have different members with different data types.
- The major difference is, the structure members get different memory allocation, and in case of unions there will be single memory allocation for the biggest data type

Embedded C

UDTs – Unions

Example

```
union Test
{
    char option;
    int id;
    double height;
};
```

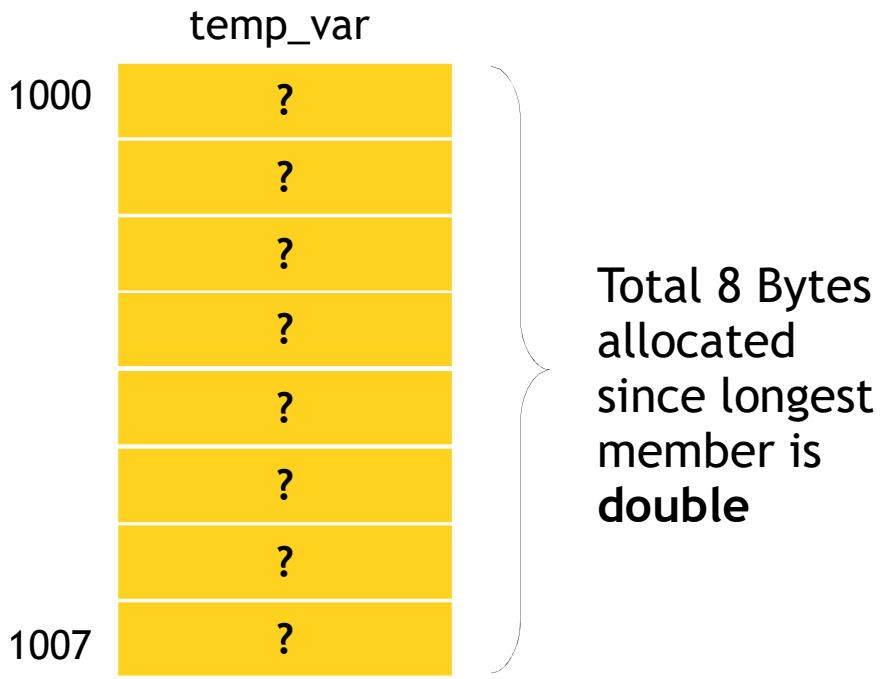
- The above union will get the size allocated for the type double
- The size of the union will be 8 bytes.
- All members will be using the same space when accessed
- The value the union contain would be the latest update
- So as summary a single variable can store different type of data as required

Embedded C

UDTs – Unions

Example

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```

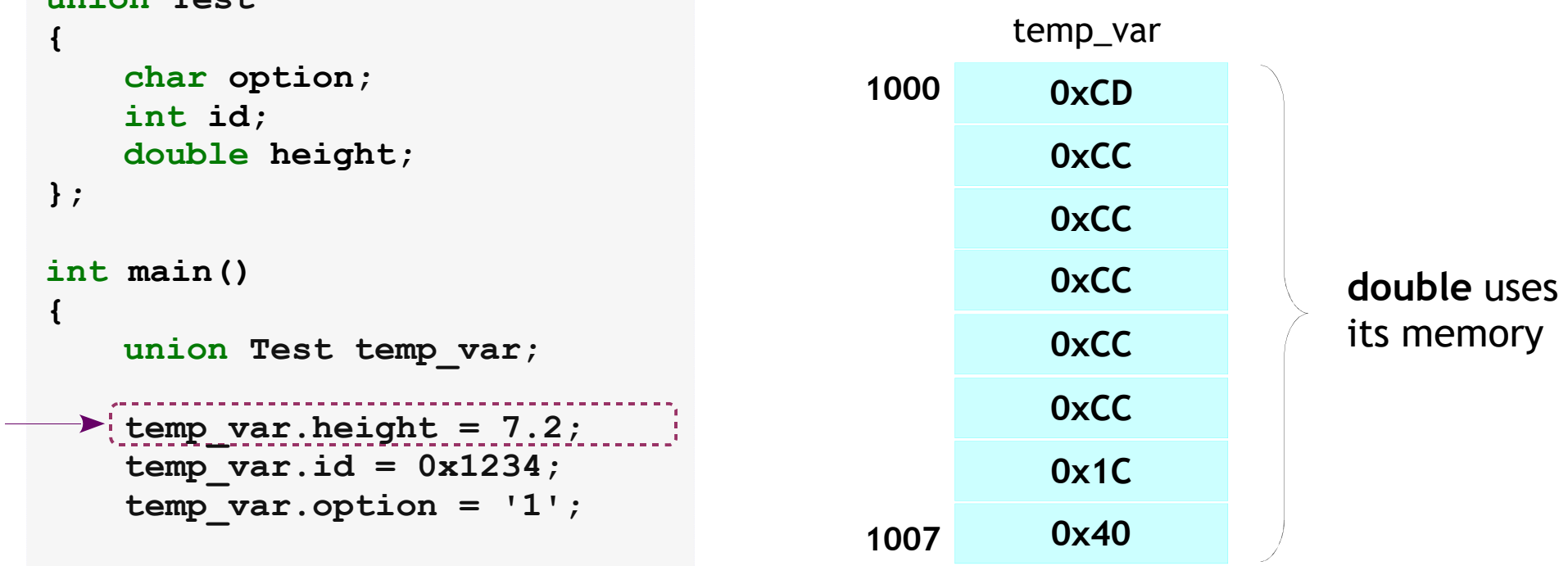


Embedded C

UDTs – Unions

Example

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```

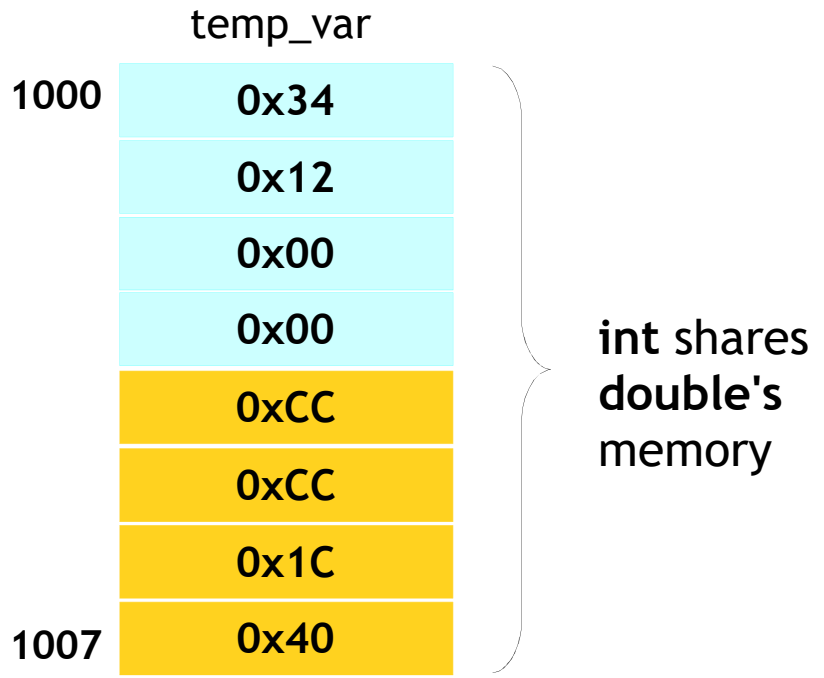


Embedded C

UDTs - Unions

Example

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```

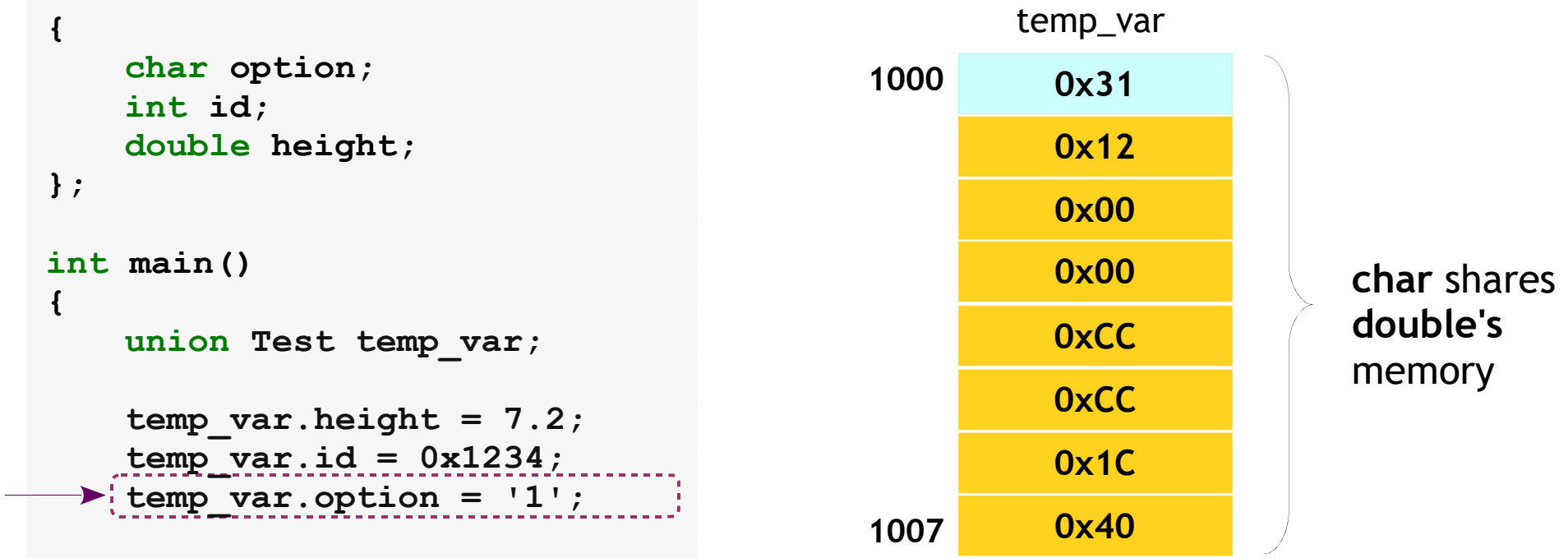


Embedded C

UDTs - Unions

Example

```
union Test
{
    char option;
    int id;
    double height;
};

int main()
{
    union Test temp_var;

    temp_var.height = 7.2;
    temp_var.id = 0x1234;
    temp_var.option = '1';

    return 0;
}
```



Embedded C

UDTs - Typedefs

- Typedef is used to create a new name to the existing types.
- K&R states that there are two reasons for using a typedef.
  - First, it provides a means to make a program more portable. Instead of having to change a type everywhere it appears throughout the program's source files, only a single typedef statement needs to be changed.
  - Second, a typedef can make a complex definition or declaration easier to understand.

Embedded C

UDTs - Typedefs

Example

```
typedef unsigned int uint;

int main()
{
    uint number;

    return 0;
}
```

Example

```
typedef int * intptr;

int main()
{
    intptr ptr1, ptr2, ptr3;

    return 0;
}
```

Example

```
typedef struct _Student
{
    int id;
    char name[30];
    char address[150]
} Student;

void data(Student s)
{
    s.id = 10;
}

int main()
{
    Student s1;

    data(s1);

    return 0;
}
```

Embedded C

UDTs - Typedefs

Example

```c
#include <stdio.h>

typedef int (*fptr)(int, int);

int add(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    fptr function;

    function = add;
    printf("%d\n", function(2, 4));

    return 0;
}
```

Embedded C

UDTs – Enums

- Set of named integral values

Examples

```
enum Boolean
{
    e_false,
    e_true
};
```

```
typedef enum
{
    e_red = 1,
    e_blue = 4,
    e_green
} Color;
```

```
typedef enum
{
    red,
    blue
} Color;

int blue;
```

- The above example has two members with its values starting from 0. i.e, e_false = 0 and e_true = 1.
- The member values can be explicitly initialized
- There is no constraint in values, it can be in any order and same values can be repeated
- Enums does not have name space of its own, so we cannot have same name used again in the same scope.