```c
#include <stdio.h>

int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.\n");
    return 0;
}
```
C Programming

- Procedural thought process
- No built-in object abstractions
  - Data separate from methods/functions
- Low memory overhead compared to Java
  - No overhead of classes
  - Relatively fast
- Heap memory management manual
- Pointers to manipulate shared data
The C runtime environment
Steps in program translation

**Code Time**

Program in C

Helloworld.c

**Compile Time**

Compiler

**Run Time**

Hardware

**Program:**
Text file stored on computers hard disk or some secondary storage

**Executable:**
- Program in machine code
- Data in binary

```
10001100011000100000000000000000
10001100111100100000000000001000
10101100111100100000000000000000
10101100011000100000000000001000
```

Moved to main memory
What does gcc do?

```
gcc hello.c
```

- Input C file
- Linux command line prompt
- `compilation`
What does gcc do?

% gcc hello.c

Source file

hello.c

gcc

Executable

a.out
What does gcc do?

% gcc hello.c

"Source"

Program in C

```c
#include <stdio.h>
void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = 'a';
    }
}
int main()
{
    func1();
    printf("abc");
}
```
What does gcc do?

% gcc hello.c

“Source”
Program in C

#include <stdio.h>
void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = ‘a’ ;
    }
}
int main()
{
    ....
    func1();
    printf("\abc");
}

“Executable”:
Equivalent program in machine language

Executable:

```
0000 1001 1100 0110
1010 1111 0101 1000
1010 1111 0101 1000
0000 1001 1100 0110
1100 0110 1010 1111
0101 1000 0000 1001
0101 1000 0000 1001
1100 0110 1010 1111
```
What does gcc do?

% gcc hello.c
% ./a.out  (executable loaded in memory and processed)
Also referred to as “running” the C program

“Source”: Program in C

“Executable”: Equivalent program in machine language
How is ‘other’ code included?

- Include Header files (`.h`) that contain function declarations - the function interface
- The corresponding `.c` files contain the actual code

```c
#include <stdio.h>

void func1(int a, char *b)
{
    if(a > 0)
    {
        *b = ‘a’;
    }
}

int main()
{
    ....
    func1();
    printf(“\abc”);
}
```
Manipulating data objects in C
How we manipulate variables depends on their data-type

Data-types

- Basic: int, char, float
- Derived: Pointers, Arrays, Structures, enumerations
In C, a variable is declared by

A. using it in an arithmetic expression

B. preceding it with one or more keywords to indicate its ‘data type’, e.g. char c;

C. initializing it to a value
Basic data object in memory

A region in memory that contains a value and is associated with a name/identifier

```c
int num;
num = 20;
```

Memory address → 102

Identifier → num

Byte level representation

```
0x20
```

4 bytes

```
0x00 0x01 0x02 0x03
```
Basic data object in memory

A region in memory that contains a value and is associated with a name/identifier

Attributes of a Data-object/variable:
- Name/identifier
- Value
- Address: Location in memory
- Size
- A unique data-type
- Lifetime
- Scope
Declarations and definitions

- `char c='a';`  /* 1 byte */
- `short s;`  /* 2 bytes */
- `int a;`  /* usually 4 bytes - signed */
- `unsigned int a=0;`  /* usually 4 bytes*/
- `float f;`  /* usually 4 bytes use sizeof(float)*/
Can we change the location of a variable after it is declared?

A. Yes

B. No

\( \& i \) is a value, should be used on the right side of the equality (assignment) operator.

\( \& i = \) This is incorrect

You can only get the address using \( \& \).
Accessing value, Lvalue and Rvalue

• To access/change the value of a basic type:

\[
y = x; \\
x = 10; \\
y = \begin{cases} 
  x & \text{if } x > y \\
  y & \text{else}
\end{cases};
\]

102

20
Accessing location

To access the location/address, use the address operator ‘&’

&x (is 102)

7

102

address of x

20

x
• **Pointer**: A variable that contains the **address** of a variable

```c
int *x;
```
Pointers

• **Pointer:** A variable that contains the **address** of a variable

```c
int *x, y;
y = 3;
x = &y;
```

![Diagram showing the use of pointers with variables x and y, and the allocation of memory addresses 102 and 120.](image-url)
• **Pointer**: A variable that contains the **address** of a variable.

```c
int *x, y;
y = 3;
x = &y;
```

```
102  120
x y  
```

```
102 120
x y  
```

```
102 120
x y  
```

```
120  3
x y  
```

```
120  3
x y  
```

```
120  3
x y  
```

- `x` points to `y`

```
120  3
x y  
```

```c
sizeof(x) = 
```

```c
y  
```

```c
120
x y  
```

```c
x points to y
```
Pointer Diagrams

• Short hand diagram for the following scenario
Using pointers to change the value of a variable

Use dereference * operator to left of pointer name

\[ \text{x} \quad \text{y} \quad 3 \]

\[ \text{x} \quad \text{y} \quad 3 \]

\[ \text{x} = 5; \]
Two ways of changing the value of any variable
Why this is useful will be clear when we discuss functions and pointers

\[ \textit{x} \rightarrow 3 \]

\[ \textit{y} \]

\[ \textit{x} \] \[ \textit{y} \rightarrow 5 \]

\[ \textit{x} = 5; \]
Pointers and Pointees

```
int *p1, *p2, x;
p1 = &x;
p2 = p1;
```

Q: Which of the following pointer diagrams best represents the outcome of the above code?

A. 

B. 

C. Neither, the code is incorrect
Pointer and Pointee data types

Q: This code gives a warning at compile time. Why?

```c
char *p;
int y = 10000;
p = &y;
```

A. The pointer ‘p’ is made to point to a variable of incompatible type

B. *p does not contain a valid value because y was not initialized
Q: Does the following code give an error at run time?

\[\text{int } *p; \]
\[*p = 5; \]

A. Yes
B. No

\(\)
Q: What is the output of this code?

```c
int *p, x = 5;
printf("%d",(*p)++);
```

A. The value pointed to by p, which is 5
B. The value pointed to by p plus one, which is 6
C. Undefined
Two important facts about Pointers

1) A pointer can only point to one type – (basic or derived) such as int, char, a struct, another pointer, etc.

2) After declaring a pointer: int *ptr;
   ptr doesn’t actually point to anything yet. We can either:
   - make it point to something that already exists, or
   - allocate room in memory for something new that it will point to… (next lecture)
Array Basics

- int ar[5];  // declares a 5-element integer array

  - address of the 0th element = base address
  - ar has this base address
  - ar points to ar[0]
Array Basics

int ar[5];  // declares a 5-element integer array
int ar[] = {795, 635};  // declares and fills a 2-element integer array.
Accessing elements:

\[ \text{ar}[i]; // returns the } i^{\text{th}} \text{ element} \]

- How are arrays in C different from Java?
- Pointers and arrays are very similar

\[ * (\text{ar} + i) \text{ same as } \text{ar}[i] \]
Arrays and Pointers

- \( \text{ar} \) is a pointer to the first element
- \( \text{ar}[0] \) is the same as \( *\text{ar} \)
- \( \text{ar}[2] \) is the same as \( *(\text{ar}+2) \)
Arrays and Pointers

- Use pointers to pass arrays to functions
- Use *pointer arithmetic* to access arrays more conveniently
Since a pointer is just a memory address, we can add to it to traverse an array.

\( \text{ptr} + 1 \) will return a pointer to the next array element.
Pointer Arithmetic

- *ptr+1 = ?
- *ptr++ = ?
- *(ptr+1) = ?

A. 21, 20, 40
B. 21, 21, 40
C. 21, 40, 40
Arrays: Fast data access

- Using pointer arithmetic, easy to compute the address of any array element in memory
- How are array elements accessed on an ARM?

![Diagram showing memory access with pointers and registers.](image)
Restrictions on memory access

- Not all of memory is accessible by your program

![Diagram showing memory access and pointers](image)
Q: Which of the assignment statements produces an error at compilation. Why?

int *p, ar[5];  //Declaration

i) p=ar+5;      ar     100 104 108 112 116

ii) ar=p+1;

A. p=ar+5;  ok but don't do it in your program
B. ar=p+1;  cannot change base address.
C. Both statements result in error at compile time
D. Neither results in a compilation error
Q: What happens when the following code is executed?

```c
int *p, ar[5]; //Declaration
p=ar-5;
*p=0;
```

A. Always results in a segmentation fault because a pointer cannot be used to change the value of an array element
B. Always results in a segmentation fault because the array element being accessed is out of bounds
C. Likely to result in a segmentation fault because the memory location being accessed may not be a valid address
D. It results in a compilation error
Arrays

- Pitfall: An array in C does not know its own length, & bounds not checked!
  - Consequence: We can accidentally access off the end of an array.
  - Consequence: We must pass the array and its size to a procedure which is going to traverse it.
- Segmentation faults and bus errors:
  - These are VERY difficult to find, so be careful.
What if we have an array of large structs (objects)?

C takes care of it: In reality, \( p + 1 \) doesn’t add 1 to the memory address, but rather adds the size of the array element.

C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes: 1 byte for a char, 4 bytes for an int, etc.
How many of the following are invalid?

I. `pointer + integer (ptr+1)`
II. `integer + pointer (1+ptr)`
III. `pointer + pointer (ptr + ptr)`
IV. `pointer – integer (ptr – 1)`
V. `integer – pointer (1 – ptr)`
VI. `pointer – pointer (ptr – ptr)`
VII. `compare pointer to pointer (ptr == ptr)`
VIII. `compare pointer to integer (1 == ptr)`
IX. `compare pointer to 0 (ptr == NULL)`
X. `compare pointer to NULL (ptr == NULL)`

#invalid

A: 1
B: 2
C: 3
D: 4
E: 5
How many of the following are invalid?

I. pointer + integer (ptr+1)
II. integer + pointer (1+ptr)
III. pointer + pointer (ptr + ptr)
IV. pointer – integer (ptr – 1)
V. integer – pointer (1 – ptr)
VI. pointer – pointer (ptr – ptr)
VII. compare pointer to pointer (ptr == ptr)
VIII. compare pointer to integer (1 == ptr)
IX. compare pointer to 0 (ptr == NULL)
X. compare pointer to NULL (ptr == NULL)

#invalid
A: 1
B: 2
C: 3
D: 4
E: 5
Functions:
Breaking large computing tasks to smaller ones
C Programming

- Procedural thought process

```c
main () /* High level Outline */
{
    ...
    get_input(arg1) /*Comment: Step 1 */
    perform_step_2(arg2);
    perform_step_3();
    store_result(); /* Print output or store in a file */
}
```
Overview of Functions

Functions make code easy to
- Maintain
- Debug
- Reuse
Functions Overview

void foo(int x, int y); //Declaration
Functions Overview

//Definition

```c
void foo(int x, int y) {
    int tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

What does foo do?
Stack Allocation: Function local variables and parameters

- When program execution starts

What if main calls the function func()?
Stack Allocation: Function local variables and parameters

Local variables of main()
Variables declared within a function can be used:

A. By other functions

B. Only within the function
Lifetime of data

- Lifetime: The interval between time of creation and end of existence of data
Possible lifetimes of data

1. Execution time of program
2. Time between explicit creation and explicit deletion
3. Execution time of a function (time between function call and function return)

*Variables whose lifetime is the execution time of function are managed using the stack structure*
Specifying Scope and Lifetime

Scope and lifetime are often implicit but sometimes we have to use specific keywords:

- `static int a=0; /*Defines lifetime*/`
- `extern int a; /*Extends scope to multiple files*/`
Different types of data

What factor differentiates the following three ‘categories’ of data in a C program:

- Global and static variables
- Local variables
- Dynamic variables

A. Scope  
B. Lifetime  
C. Representation
Different types of data

What factor differentiates the following three ‘categories’ of data in a C program:

- Global and static variables
- Local variables
- Dynamic variables

A. Scope
B. Lifetime
C. Representation
Functions: Call by value

main() {
    ... 
    swap(a, b); 
    ... 
}

Q: Are the value of variables ‘a’ and ‘b’ interchanged after swap is called?

A. Yes, because that’s what is implemented by the ‘swap’ routine

B. No, because the inputs to swap are only copies of ‘a’ and ‘b’
Functions: Call by value

main() {
    ... 
    swap(a, b);
    ... }

Q: Are the value of variables ‘a’ and ‘b’ interchanged after swap is called?

A. Yes, because that’s what is implemented by the ‘swap’ routine

B. No, because the inputs to swap are only copies of ‘a’ and ‘b’
Q: Which of the following changes are required to interchange the values in ‘a’ and ‘b’ when swap is called?

A. In swap, return the values of ‘x’ and ‘y’ to the main function after swapping them

B. Declare ‘a’ and ‘b’ as global variables, so that they become accessible to the swap routine

C. Pass the address of ‘a’ and ‘b’ to swap instead of their value

D. Move the implementation in swap to the main function
Q: Which of the following changes are required to interchange the values in ‘a’ and ‘b’ when swap is called?

A. In swap, return the values of ‘x’ and ‘y’ to the main function after swapping them

B. Declare ‘a’ and ‘b’ as global variables, so that they become accessible to the swap routine

C. Pass the address of ‘a’ and ‘b’ to swap instead of their value

D. Move the implementation in swap to the main function
Functions: Call by reference

```c
void swap(int *x, int *y) {
    // code
}
```

Q: What should the modified swap function do?

A. Swap the addresses in ‘x’ and ‘y’

B. Swap the values pointed to by ‘x’ and ‘y’

C. Both the above operations are equivalent
Functions: Call by reference

void swap(int *x, int *y) {
    // ... 
}

Q: What should the modified swap function do?

A. Swap the addresses in ‘x’ and ‘y’

B. Swap the values pointed to by ‘x’ and ‘y’

C. Both the above operations are equivalent
void IncrementPtr(int *p) {
    p = p + 1;
}

Q: What happens when *IncrementPtr*(q) is called in the following code:

```c
int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
```

A. The pointer *q* points to the next element in the array with value 60
B. The pointer *q* points to the first element in the array with value 50
void IncrementPtr(int *p) {
    p = p + 1;  }

Q: What happens when `IncrementPtr(q)` is called in the following code:

```c
int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(q);
```

A. The pointer `q` points to the next element in the array with value 60
B. The pointer `q` points to the first element in the array with value 50
void IncrementPtr(int **p){
    p = p + 1;
}

Q: How should we implement IncrementPtr(), so that ‘q’ moves by one element when the following code executes?

int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);

A. p = p + 1; /*The current one is correct*/
B. &p = &p + 1;
C. *p = *p + 1;
D. *p++;
E. p = &p + 1;
void IncrementPtr(int **p){
    p = p + 1;  }

Q: How should we implement `IncrementPtr()`, so that ‘q’ moves by one element when the following code executes?

```
int A[3] = {50, 60, 70};
int *q = A;
IncrementPtr(&q);
```

A. `p = p + 1;` //The current one is correct
B. `&p = &p + 1;`
C. `*p = *p + 1;`
D. `*p++;`
E. `p = &p + 1;`
What is really in a executable

% gcc hello.c
% ./a.out  (executable loaded in memory and processed)
Also referred to as “running” the C program

“Source”: Program in C

“Executable”: Equivalent program in machine language
What does the executable contain?

- Instructions or “code”
- Data

What is data?

- Any information that instructions operate on (objects in memory)
C Runtime Environment

- "Code" (instructions in machine language)
- "Data" (initialized and uninitialized - static allocated)

Both code and data don’t change in size

- "Heap" (for dynamically allocated data)
- "Stack" (for function local variables)

Heap and stack change in size as the program executes
C Runtime Environment

- Code
- Initialized Data
- Uninitialized Data
- Heap
- Stack