### Pointer Declarations

```c
#type-name
*pointerVar
```

The `*` and `->` operators change a pointer's value by accessing the operand that the pointer points to. It is crucial to ensure proper usage to avoid runtime errors.

**WARNING:** C does not perform any automatic run-time checking for proper pointer usage.

In C, pointers are used to access the addresses of variables. The address-of operator (`&`) is used to create addresses of ordinary variables.

```c
void *P; /* P is a pointer to an integer*/
```

The `&` operator is used to create the address of any ordinary variable.

**WARNING:** It is an ERROR to apply the `*` operator to an uninitialized pointer variable.

The pointer dereferencing operator (`*`) is used to access the variable that a pointer variable is pointing at.

```c
int variable;
void *P = &variable; /* P is a pointer to the variable */
```

The `++` and `--` operators change a pointer variable by the size of the object that the pointer points at. This feature should only be used to access consecutive elements of arrays.

**WARNING:** It is an ERROR to apply the `*` operator to a non-constant pointer variable.

In C, all pointers hold the addresses of variables of one specific type.

### Poliing in C

During the execution of a program, all variables in the program are stored in the memory of the computer. Each variable has two attributes:

- **Value:** The content of the variable
- **Address:** The location of the variable in memory

In C, almost all pointers hold the addresses of variables of one specific type. A pointer is a special kind of variable whose value is the address of other variables.

### Pointer Declarations

```c
#type-name
*pointerVar
```

In C, a pointer is a special kind of variable whose value is the address of other variables. The location of a variable in memory is called the address of the variable. The memory of the computer is divided into blocks of memory, all variables in the program are stored in these blocks.
It is extremely poor programming practice to do anything with a pointer that has an incompatible address. Such usage is non-portable and inherently error prone.

Pointers variables of type `void*` are used to store pointers that do have a declared type.

There are NO operations defined for the data type `void`, so directly dereferencing a `void` pointer is an ERROR.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type. Such usage is non-portable and inherently error prone.

It is extremely poor programming practice to do anything with a pointer that has an incompatible address. Such usage is non-portable and inherently error prone.

**Pointers** are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.

**Pointers** variables of type `void*` are used to store pointers that do have a declared type.

Any use of a `void` pointer will require a type cast to make it legal.
HOW TO Use Pointers in Array Loops

It is often more efficient or more convenient to use a pointer rather than an index to process an array. If the type of the array element is type-name, use a pointer variable with type type-name.*

For an array A, the address of the first element in the array is &A[0]. If ASIZE is the size of the array, the address of the last element in the array is &A[ASIZE]. The address of the first element that is not in the array is &A[ASIZE].

For example:

```c
int X[400], Y[400];
double XP, *YP;
...
for (I = 0; I < 400; I++)
for (XP = &X[0], YP = &Y[0]; XP <= &X[400]; XP++)
  X[K] = Y[K];
```

**Pointer Arithmetic and Arrays**

C supports the following pointer arithmetic:

- `*p`: Dereferences a pointer to the first element of the array.
- `p++`: Points to the next element in the array.
- `++p`: Points to the next element in the array.

There is an equivalence between array subscripting and pointer arithmetic. For example:

```
A[K] is the same as *(A + K) or *(A + K)
```

In C, if the address of an element is less than the address of the next element, `++p` is undefined. Otherwise, it points to the next element in the array.

**Arrays and Pointers**

- `p = &X[K]; /*assigns to A[K] and sets p to points at A[K+1]. Example:
  p = &X[0];
  while (p <= &X[100])
    X[K] = Y[K];
```

**Good Style:** NEVER use pointer arithmetic to address outside of an array.

**WARNING:** Do not check addresses computed using pointer arithmetic.

- In C, if the address of an array is an array of pointers, then the address of an element is the address of the first element in the array.
**Pointers, Parameters and Arguments**

For any one-dimensional array &name, the expression &name[ ] is a pointer to the first element.

For any two-dimensional array &name[ ][ ] of type type-name, the expression &name[ ][ ] is a pointer to the first element of the array.

**Multidimensional Arrays and Pointers**

For an array of type &name[ ][ ] of type type-name, the expression &name[ ][ ] is a pointer to the first element of the array.

**Multi-dimensional Arrays and Pointers**

For any one-dimensional array &name[ ] of type type-name, the expression &name[ ] is a pointer to the first element.
Function Pointer Examples

```c
float bisection(float (*funcPtr)(float), float x0, float x1);

float function1(float x);
float function2(float x);

... y = bisection(function1, 0.0, 2.0);
... y = bisection(function2, -10.0, 10.0);
```

Pointers to Functions

```c
(type-name (*) (parameterList));
```

This declaration declares a variable (funcPointer) that is a pointer to a function.

The function constant assigned to a function pointer should always be compatible with the declaration for the function pointer variable.

The function constant passed as an argument to a function pointer parameter should always be compatible with the corresponding parameter declaration.

Compatible means:
- The type returned by the function is the same.
- Corresponding parameters are of the same type.
- The type returned by the function is the same.

If these rules are not followed, chaos will ensue.

Pointer Returning Functions

A function may be declared to return a pointer to some type of object. The following rules apply to the value of the pointer returned by such a function:

- It is an ERROR to return a pointer to any of the function's local variables. Those variables cease to exist when the function returns.
- If a function returns a pointer, a pointer to any of the function's local variables. Those variables cease to exist when the function returns.
- A function cannot return a pointer to a variable stored by the function.
- A function cannot return a pointer to a variable that is not declared by its declaration.
- The function constant may be a pointer to an object of some type of object. The following rules apply to the value of the pointer returned by such a function:

WARNING: A pointer pointing to a local (automatically) stored by the function:
- It is an ERROR to return a pointer to a variable stored by the function.
- The function constant may be a pointer to an object of the constant.
Structures and Unions

A structure is a mechanism that allows several data items of arbitrary types to be treated as a single entity.

Examples:
- name, address, telephone number
- X coordinate, Y coordinate, Z coordinate
- student name, student number, assignment marks

A union is a mechanism for saving space when several mutually exclusive data items need to be stored in the same space.

Examples:
- x coordinate, y coordinate, z coordinate
- name, address, telephone number

Structures are preferred when some block of logically related information needs to be processed as a group.

Example: A student structure

```
struct student {
    char name[25];
    char label[33];
    int price;
    float value;
}
```

Good style: Always use a typedef to create a single point of definition for any structure or union that has a significant use in a program.

Example:
```
typedef struct exStruct {
    char name[25];
    char label[33];
    int price;
    float value;
} myStruct;
```

Chapter 12: Structures and Unions

K.N. King Chapter 16, 18

Supplementary Reading

Chapter 12: Structures and Unions

K.N. King Chapter 16, 18

Reading Assignment

K.N. King Chapter 16, 18
Union Types

The optional identifier can be a list of union variables that are being declared at
the same time as the union.

A union is a structure that packs several data items into one structure.
The field alternatives is a list of mutually exclusive field alternatives.
The field alternatives is a list of mutually exclusive field alternatives.
The unionTag is the name of the union type.

union unionTag
  fieldAlternativesList;

The fieldAlternativesList is a list of union variables that are being declared at
the same time as the union.

union unionTag
  fieldAlternativesList;

The unionTag is the name of the union type.

使用结构来打包一组相关变量。

Each unionTag is the name of the union type.

Each fieldAlternativesList is a list of mutually exclusive field alternatives.

Each fieldAlternatives is a single data declaration.

Use a structure to pack several data items into one alternative.

The optional identifier can be used as a field of a structure including another.

Use a structure to pack several data items into one alternative.

Each fieldAlternativesList is a list of mutually exclusive field alternatives.

Each fieldAlternatives is a single data declaration.

Use a structure to pack several data items into one alternative.

The optional identifier can be used as a field of a structure including another.

Use a structure to pack several data items into one alternative.

Each fieldAlternativesList is a list of mutually exclusive field alternatives.

Each fieldAlternatives is a single data declaration.

Use a structure to pack several data items into one alternative.

The optional identifier can be used as a field of a structure including another.

Use a structure to pack several data items into one alternative.

Each fieldAlternativesList is a list of mutually exclusive field alternatives.

Each fieldAlternatives is a single data declaration.

Use a structure to pack several data items into one alternative.

The optional identifier can be used as a field of a structure including another.

Use a structure to pack several data items into one alternative.

Each fieldAlternativesList is a list of mutually exclusive field alternatives.

Each fieldAlternatives is a single data declaration.
Unions are a mechanism for saving space when several mutually exclusive alternatives sets of data need to be stored and treated like a single object.

The size of a union is approximately the size of the largest field alternative in the union. Use sizeof to get the exact size of a union.

C does NOT run-time checking for proper use of unions.

The programmer must provide some way of indicating which field alternative is active at any instant in time.

Example of self-identifying structure/union.

```
typedef enum
point, square, circle, triangle
uType;

struct uStruct
{ uType unionKind;
  double side;
  /*square*/
  int radius;
  /*circle*/
  float sideA, sideB, sideC;
  /*triangle*/
  value;
  double Xcoordinate, Ycoordinate;
};
```

Example of union declaration.

```
typedef union a myflag
{ double x, y, z; /*point*/
  float value; /*double*/
  char *pointer; /*void pointer*/
};
```