6. Data Types

6.1. Introduction

Data Type	A data type defines a collection of data values and a set of predefined operations on those values.
Descriptor	A descriptor is the collection of the attributes of a variable. If the attributes are all static, descriptors are required only at compile time. Descriptors are built by the compiler, usually as a part of the symbol table.
Object	An object is an instance of a user-defined abstract data type.

6.2. Primitive Data Types

Primitive Data Type	Data types that are not defined in terms of other types are called primitive data types.

6.2.1. Numeric Types

6.2.1.1. Integer

Definition: An integer is a number without a fractional part. The set of integers is the union of the set of whole numbers and the set of negative counting numbers.

Integers and whole numbers. C++ implements both integers (signed) and whole numbers (unsigned).

Ranges: The range of values a particular integer variable can take on is limited by the number of bits allocated to that variable. The *type-specifiers* **char**, **short**, **int**, and **long** define the relative range of values that a variable of that type can take on.

char ≤ short ≤ int ≤ long

Implementation: Integers are implemented as **two's complement binary integers**. Whole numbers are implemented as unsigned binary integers. Several field widths (*w*) are common including **8**, **16**, **32**, and **64** bits.

Integers: Let $\mathbb{Z}$ be the set of integers and I designate the set of integer types.

$$\boldsymbol{I = \left\{ i \in \mathbb{Z} \middle| -2^{w-1} \leq i \leq 2^{w-1} - 1, w \in \{8, 16, 32, 64\} \right\}}$$

- An 8-bit integer $\boldsymbol{c}$ ranges from $\boldsymbol{-2^7 \leq c \leq 2^7 - 1}$ or $\boldsymbol{-128 \leq c \leq 127}$
- An 16-bit integer $\boldsymbol{s}$ ranges from $\boldsymbol{-2^{15} \leq s \leq 2^{15} - 1}$ or $\boldsymbol{-32,768 \leq s \leq 32,767}$
- 32-bit integer $\boldsymbol{i}$ ranges from $\boldsymbol{-2^{31} \leq i \leq 2^{31} - 1}$ or $\boldsymbol{-2,147,483,648 \leq i \leq 2,147,483,647}$
- A 64-bit integer $\boldsymbol{l}$ ranges from $\boldsymbol{-2^{63} \leq l \leq 2^{63}}$

Let U designate the set of unsigned integer types.

$$\boldsymbol{U = \{u \in \mathbb{Z} | 0 \leq u \leq 2^n - 1, n \in \{8, 16, 32, 64\}\}}$$

- An 8-bit whole number $\boldsymbol{c}$ ranges from $\boldsymbol{0 \leq c \leq 2^8 - 1}$ or $\boldsymbol{0 \leq c \leq 255}$.
- A 16-bit integer $\boldsymbol{s}$ ranges from $\boldsymbol{0 \leq s \leq 2^{16} - 1}$ or $\boldsymbol{0 \leq s \leq 65,535}$.
- A 32-bit integer $\boldsymbol{i}$ ranges from $\boldsymbol{0 \leq i \leq 2^{32} - 1}$ or $\boldsymbol{0 \leq i \leq 4,294,967,296}$.
- A 64-bit integer $\boldsymbol{l}$ ranges from $\boldsymbol{0 \leq l \leq 2^{64} - 1}$.

Integers and whole numbers. The relationship between integers and whole numbers for a given size is shown in Figure 1.

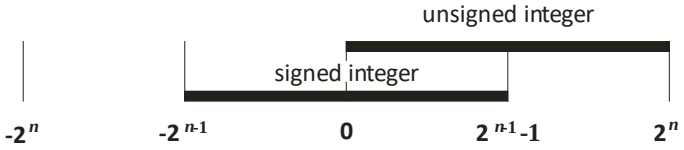


Figure 1. Signed and unsigned integer values

Representation: Implementation: Integers are implemented as two's complement binary integers. Whole numbers are implemented as unsigned binary integers. Several field widths (*w*) are common including 8, 16, and 32 bits.

0	0	0	0	0	1	0	1
7	6	5	4	3	2	1	0

Bit Position

Figure 2. 8-bit whole number representation **unsigned char** *uc***=5;**

1	1	1	1	1	0	1	1
7	6	5	4	3	2	1	0

Bit Position

Figure 3. 8-bit integer representation **char** *sc***=-5;**

6.2.1.2. Floating-Point

Definition:

Real types simulate real numbers. Real types are discrete whereas the set of real numbers is continuous. Real types are called floating-point numbers. The density of floating-point numbers is shown on a real number line in Figure 1.

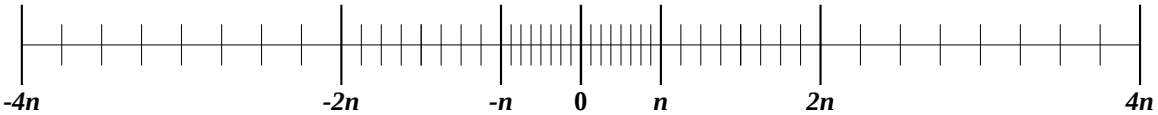
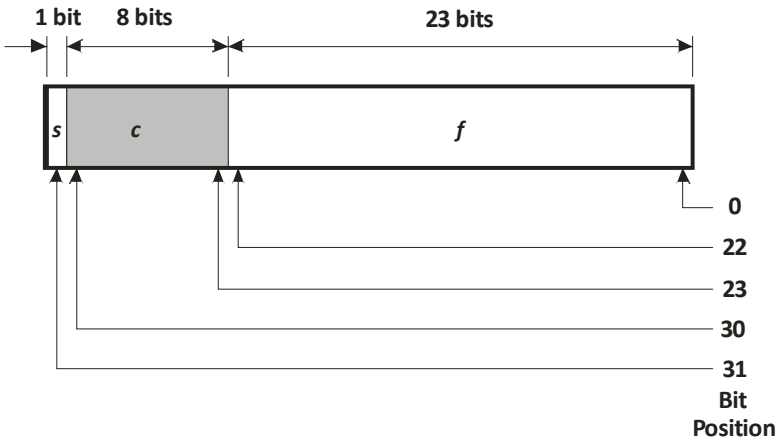


Figure 4. Density of floating-point numbers.

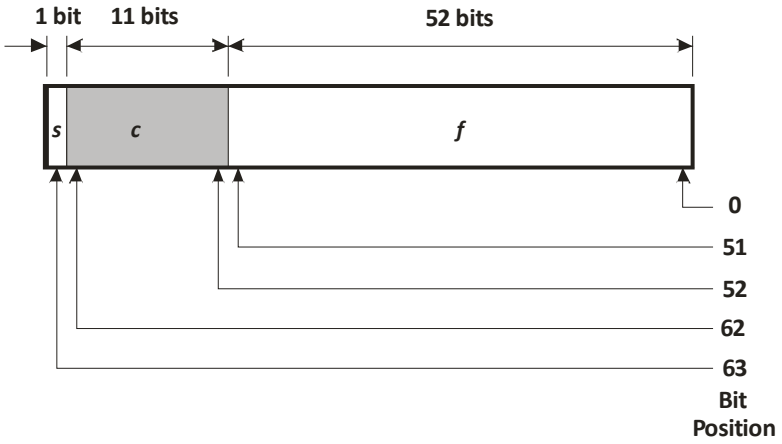
Sets: Each set is dependent on its representation.



Legend: *s*: sign
c: characteristic
f: fraction
b: bias

Figure 5. IEEE-754 single binary floating-point representation used to implement type **float**.

$$R = \{r \in \mathbb{R} | -1^s \times 2^{c-b} \times 1.f, s \in \{0,1\}, 1 \leq c \leq 254, b = 127, f = \sum_{k=1}^{23} f_k \times 2^{-k}, f_k \in \{0,1\}\}$$



Legend: *s*: sign
c: characteristic
f: fraction
b: bias

Figure 6. IEEE-754 double binary floating-point representation used to implement type **double**.

$$R = \{r \in \mathbb{R} | -1^s \times 2^{c-b} \times 1.f, s \in \{0,1\}, 1 \leq c \leq 2047, b = 1023, f = \sum_{k=1}^{52} f_k \times 2^{-k}, f_k \in \{0,1\}\}$$

6.2.1.3. Complex

Fortran and Python support a primitive type of complex. A value of type complex is an ordered pair of floating-point values. The example below is from Python
(7+3j)

6.2.1.4. Decimal

Computers designed for business systems applications often have support for decimal data types. COBOL, C#, and Visual Basic have decimal types.

Decimal types are stored very much like character strings, using binary codes for the decimal digits.

BCD Code

BCD codes were conceived to perform decimal arithmetic and are particularly useful where monetary values are represented. Banks and other financial institutions require that all monetary values be resolved to the nearest penny.

Four bits are used to represent each decimal digit leaving six unused codes as shown below.

Decimal Symbol	BCD Digit
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Each decimal digit is represented by a group of four (4) bits. For example,
$(185)_{10} = (0001\ 1000\ 0101)_{BCD}$

6.2.2. Boolean Types

The set B of Boolean values consist of two values $B = \{\mathbf{false}, \mathbf{true}\}$

Representation

0	0	0	0	0	0	0	1
7	6	5	4	3	2	1	0

Bit Position

Figure 7. Boolean data representation for constant **true**.

6.2.3. Character Types

Representation:

A variable of type **char** can store a single member from the set of ASCII (American Standard Code for Information Interchange).

A variable of type **wchar_t** can store a wide character occupying 16 bits.

Characters are integer codes. Let C_8 be the set of all characters represented by type **char**. $C_8 = \{c \in C_8 | 0 \leq c \leq 2^8 - 1\}$

Let C_{16} be the set of all characters represented by the type **wchar_t**.

$$C_{16} = \{c \in C_{16} | 0 \leq c \leq 2^{16} - 1\}$$

A character is distinguished from an integer only when it is printed or displayed. Instead of printing or displaying the integer code, the character face is printed or displayed.

Decimal Code	ASCII Character	Decimal Code	ASCII Character	Decimal Code	ASCII Character	Decimal Code	ASCII Character
0	NUL	32	SP	64	**@**	96	**`**
1	SOH	33	**!**	65	**A**	97	**a**
2	STX	34	**“**	66	**B**	98	**b**
3	ETX	35	**#**	67	**C**	99	**c**
4	EOT	36	**$**	68	**D**	100	**d**
5	ENQ	37	**%**	69	**E**	101	**e**
6	ACK	38	**&**	70	**F**	102	**f**
7	BEL	39	**‘**	71	**G**	103	**g**
8	BS	40	**(**	72	**H**	104	**h**
9	HT	41	**)**	73	**I**	105	**i**
10	LF	42	*****	74	**J**	106	**j**
11	VT	43	**+**	75	**K**	107	**k**
12	FF	44	**,**	76	**L**	108	**l**
13	CR	45	**-**	77	**M**	109	**m**
14	SO	46	**.**	78	**N**	110	**n**
15	SI	47	**/**	79	**O**	111	**o**
16	DLE	48	**0**	80	**P**	112	**p**
17	DC1	49	**1**	81	**Q**	113	**q**
18	DC2	50	**2**	82	**R**	114	**r**
19	DC3	51	**3**	83	**S**	115	**s**
20	DC4	52	**4**	84	**T**	116	**t**
21	NAK	53	**5**	85	**U**	117	**u**
22	SYN	54	**6**	86	**V**	118	**v**
23	ETB	55	**7**	87	**W**	119	**w**
24	CAN	56	**8**	88	**X**	120	**x**
25	EM	57	**9**	89	**Y**	121	**y**
26	SUB	58	**:**	90	**Z**	122	**z**
27	ESC	59	**;**	91	**[**	123	**{**
28	FS	60	**<**	92	****	124	**|**
29	GS	61	**=**	93	**]**	125	**}**
30	RS	62	**>**	94	**^**	126	**~**
31	US	63	**?**	95	**_**	127	DEL

Figure 8. ASCII Character Set

Coding	Description
ASCII	American Standard Code for Information Interchange 8-bit code 0 to 255
ISO 8859-1	International Standards Organization 8-bit code 0 to 255 Used by Ada 95
UCS-2	Unicode Consortium 16-bit code published in 1991 First 128 characters are identical to ASCII Java, JavaScript, Python, Perl, C#

6.3. Character String Types

Character String Type	A character string type is one in which the values consist of sequences of characters.

6.3.1. Design Issues

- Should strings be simply a special kind of character array or a primitive type?
- Should strings have static or dynamic length?

6.3.2. Strings and Their Operations

Operations

- assignment
- catenation
- substring reference
- comparison
- pattern matching

A substring reference is a reference to a substring of a given string.

Assignment and comparison operations on character strings are complicated by the possibility of string operands of different lengths. For example, what happens when a longer string is assigned to a shorter string or vice versa?

C uses **char** arrays to store character strings.

Representation:

A C-string is an array of characters terminated by a null character. For example, the string **"toy"** is represented as shown in figure 1.

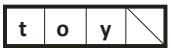


Figure 9. C-string **"toy"**.

Declaration:
A C-string is declared as an array of characters. Examples are given below.

char *s***[10];**	//*s* is an array of 10 characters having elements *s***[0] ..** *s***[9].**
	//*s* can hold up to 9 characters
char *t***[]="one";**	**//***t* is a string initialized to the characters **'o'**, **'n'**, **'e'**, **'\0'**
char *u***[3]="one";**	**//** *u* is initialized to the characters **'o'**, **'n'**, **'e'**
	//*u* is ***not*** a string.
char *v***[]={'o','n','e','\0'};**	**//***v* is a string having four (4) characters. Each character is initialized.
char *e***[]="";**	**//***e* is a string having a single character, the null terminator.
	//*e* is the empty string

Strings and pointers to strings:

1. Strings are referenced by pointers to the actual string. For example, variable *t*, is used to reference string *t* declared as **char** *t***[]="toy";**
2. When the name of an array appears without a subscripting operator **[]**, the type of the array name is changed to a pointer to the element type. For example, *t* has type **char*** because elements of *t* have type **char** and *t* is an array.
3. String pointers can be declared directly. For example, **char*** *s***;**. Variable ***s*** can be assigned to point to a string but no such assignment has been made yet. Variable *s* is said to be undefined. References to *s* will likely cause an execution-time error.
4. A string pointer can be initialized. For example **char*** *s***="toy";** Storage for string **"toy"** is allocated in the constant area of the program. The string **"toy"** cannot be changed. String *s*, however, can be reassigned. Refer to figure 2.

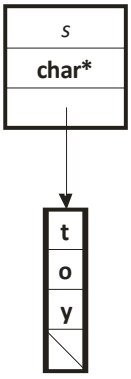


Figure 10. char* *s***="toy";**

Operations:

#include <*string***>**

Declaration	Description	Example
int *strlen***(char*** *s***);**	Function *strlen* returns the number of characters in the string referenced by parameter *s*. The terminating character is excluded from the count returned by function *strlen*.	**char** *s***[]="one";** **int** *c***=strlen(***s***);** *cout* **<<** *c***;** Output 3
char* *strcpy***(char*** *d***,char*** *s***);**	Function *strcpy* copies the contents of the string *s* to the string d, overwriting the contents of *d*. The entire contests of *s* are copied, plus the terminating null character even if *s* is longer than *d*. The argument d is returned.	**char** *d***[]="destinataion";** **char** *s***[]="source";** **char*** *t***=strcpy(***d***,***s***);** *cout* **<<** *d***;** Output source
char* *strcat***(char*** *d***,char*** *s***);**	Function *strcat* appends the contents of string *s* to string *d*. A pointer to string *d* is returned. The null character that terminates *d* (and perhaps other characters following it in memory) is overwritten with characters from *s* and a new terminating null character. Characters are copied from *s* until a null character is encountered in *s*. The memory beginning with *d* is assumed to be large enough to hold both strings.	**char** *d***[10]="One";** **char** *s***[]=", two";** **char*** *t***=***strcat***(***d***,***s***);** Ouput One, two
int strcmp(char* u,char* v);	Function *strcmp* lexicographically compares the contents of the null-terminated string *u* with the contents of the null-terminated string *v*. It returns a value of type **int** that is less than zero if $u < v$; equal zero if $u = v$; and greater than zero if $u > v$.	**char u[]="ted";** **char v[]="tom";** **int c=strcmp(u,v);** *cout* **<<** *c***;** Ouput -1

Table 1. Selected functions in library **cstring** (**#include <c***string***>**) continued

C++ strings

Representation:

The representation of C++ string is hidden.

Declaration:

Include file **#include <***string***>**. Use type name *string*. Review declarations below.

```
string s;          //s is a string.
string t="one";    //string t is initialized to the string "one"
string u("two");   //string u is initialized to the string "two"
string e("");      //string e is initialized to the empty string
```

Examples:

Program **p01** illustrates how to find the length of a string

```
#include <iostream>
#include <string>
using namespace std;
int main()
{   string s="toy";
    cout << "length(" << s << ")=" << s.length();
    cout << endl;
    return 0;
}
```

Figure 11. Program **p01**.

Program **p01** output.

```
length(toy)=3
```

Program **p02** illustrates string assignment.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{   string s="source";
    string d="destination";
    d=s;
    cout << "d=" << d;
    cout << endl;
    return 0;
}
```

Figure 12. Program **p02**.

Program **p02** output.

```
d=source
```

Program **p03** illustrates string concatenation.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string s("");
   for (;;) {
      cout << endl;
      cout << "Enter a string. ";
      string t;
      cin >> t;
      if (cin.eof()) break;
      s=s+" ";
      s=s+t;
      cout << s << " ";
   }
   cout << endl;
```

Figure 13. Program **p03**.

Sample program **p03** dialog.
[tt@cs L21]$ **p03**

Enter a string. **One,**
One,
Enter a string. **two,**
One, two,
Enter a string. **buckle**
One, two, buckle
Enter a string. **my**
One, two, buckle my
Enter a string. **shoe.**
One, two, buckle my shoe.
Enter a string.**^D**
[tt@cs L21]$

Pattern matching

Regular expressions used to match patterns
Examples

/[A-Za-z][A-Za-z\d]+/	The first character must be a letter The second and subsequent characters must be either a letter or a digit.
/\d+\.?\d*|\.\d+/	One or more digits followed by Optionally by a decimal point followed by zero or more digits OR a decimal point followed by one or more digits

6.3.3. String Length Options

static length string	**C, Python** The length of the string is fixed during compilation like an array bound.
limited dynamic length string	Strings vary up to maximum length that is defined during compilation.
dynamic length string	Varying length strings with no maximum similar to the C++ string defined by class string in the standard C++ library.

6.3.4. Evaluation

Strings that are implemented as arrays are more cumbersome than strings implemented as a type. When strings are implemented as an array, an assignment must be accomplished via a loop whereas strings implemented as a type have the advantage of assignment implemented via the assignment operator.

6.3.5. Implementation of Character String Types

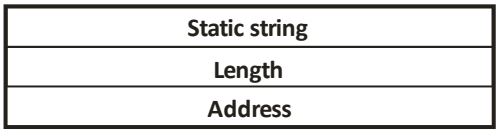


Figure 14. Compile-time descriptor for static strings.

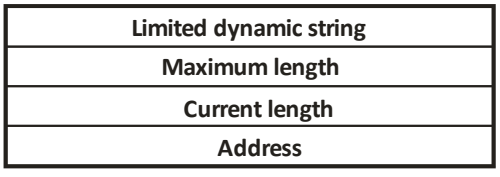


Figure 15. Compile-time descriptor for static strings.

6.4. User-Defined Ordinal Types

- An **ordinal type** is one in which the range of possible values can easily be associated with the set of integers or non-negative integers.
- Primitive ordinal types include
  - **integer**
  - **char**
  - **Boolean**
- User-defined ordinal types include
  - **enumeration**
  - **subrange**

6.4.1. Enumeration Types

From C#

enum *days* **{***sun***,***mon***,***tue***,***wed***,***thu***,***fri***,***sat***};**

Enumeration constants *sun*, *mon*, …, *sat* are names for integer values 0, 1, …, 6. Enumerated type days defines the set of value *sun*, *mon*, …, *sat* and variables of type *days* can take on those values.

Design issues are:

- Is an enumeration constant allowed to appear in more than one type definition, and if so, how is the type of an occurrence of that constant in the program checked?
- Are enumeration values coerced to integer?
- Are any other types coerced to an enumeration type?

6.4.1.1. Designs

Enumeration types were first widely used in C and Pascal.

Language	Declaration and use
Pascal	**type** *day* **= (***sun***,***mon***,***tue***,***wed***,***thu***,***fri***,***sat***);** **var** *weekday***:***day***;** *weekday***:=***wed***;** *weekday***:=***succ***(***weekday***);** *weekday***:=***pred***(***tue***);**
C	**enum** *day***{***sun***,***mon***,***tue***,***wed***,***thu***,***fri***,***sat***};** *day weekday***;** *weekday***=***wed***;** *weekday***=***weekday***+1;** *weekday***=***tue***-1;**
C++	**enum** *day***{***sun***,***mon***,***tue***,***wed***,***thu***,***fri***,***sat***};** **day** *weekday***;** *weekday***=***wed***;** *weekday***=(***day***)(***weekday***+1);** *weekday***=(***day***)(***tue***-1);**

6.4.1.2. Evaluation

- Improved **readability** and **reliability**. Named values are easily recognized whereas coded values are not.
- Ada, C#, and Java 5.0 prohibit arithmetic operations on constants and variables having enumeration an enumeration type.
- No enumeration variable can be assigned a value outside its defined range.

6.4.2. Subrange Types

- A **subrange type** is a contiguous subsequence of an ordinal type. For example, 12..14 is a subrange of integer type. Subrange types were introduced by Pascal and are included in Ada.

6.4.2.1. Ada's Design

Language	Declaration and use
Pascal	**type** *day* **= (***sun***,***mon***,***tue***,***wed***,***thu***,***fri***,***sat***);** **var** *sick***:array[***day***] of** *boolean***;** *sick***[***mon***]:=***true***;**
Ada	**type** *Days* **is (***Mon***,** *Tue***,** *We***,** *Thu***,** *Fri***,** *Sat***,** *Sun***);** **subtype** *Weekdays* **is** *Days* **range** *Mon***..***Fri***;** **subtype** *Index* **is Integer range 1..100;**

- The compiler must generate range-checking code for every assignment to a subrange variable.

6.4.2.2. Evaluation

- Subrange types enhance readability by making it clear to readers that variables of subtypes can store only certain ranges of values.

6.4.3. Implementation of User-Defined Ordinal Types

- Enumeration types are usually implemented as integers.
- Subrange types are implemented in exactly the same way as their parent types.
- Range checks must be implicitly included by the compiler in every assignment of a variable or expression to a subrange variable.

6.5. Array Types

- An **array** is an aggregate where all the elements usually have the same type.
- Originally, in Fortran, the syntax of an array was made to model that of mathematical subscripts, for example

Mathematical Representation	Typical Array Representation
a_i	*a***[***i***]**
a_{ij}	*a***[***i***][***j***]** **or** *a***[***I***,***j***]**

- In many languages, such as C, C++, Java, Ada, and C# all of the elements have the same type.
- In other languages, such as JavaScript, Python, and Ruby, variables are typeless references to objects or data values. In these cases, arrays still consist of elements of a single type, but the elements can reference objects or data values of different types.

6.5.1. Design Issues

Design issues include:

- What types are legal for subscripts?
- Are subscripting expressions in element references range checked?
- When are subscript ranges bound?
- When does array allocation take place?
- Are ragged or rectangular multidimensional arrays allowed, or both?
- Can arrays be initialized when they have their storage allocated?
- What kinds of slices are allowed, if any?

6.5.2. Arrays and Indices

- A reference to an element of an array has two parts:
  - The first part is the name of the array.
  - The second part is the **subscript** of **index**.

  Examples:

Language	Declaration(s)	Reference
C++	**double** *A***[9];** **int** *i***=5;**	*A***[***i***]**
Pascal	**var** *A***:array[0..9] of** *real***;** *i***:***integer***;** **…** *i***:=5;**	*A***[***i***]**

- A reference to an element in an array can be thought of as a mapping.
  *array_name***(***subscript_value_list***)**→*element*
- The use of parentheses is deliberate: parentheses denote a function in mathematics and functions are characterized using the mapping notation $f: D \rightarrow R$ meaning function f is a map from the set D to the set R. Specific values of D, $d \in D$ are mapped to set R by function f using the notation $f(d)$.
- Ada retained the mathematical interpretation of arrays and preserved the use of parentheses for arrays. For example:

  *Sum***:=***Sum***+***B***(***I***);**
- Designers of other languages like C, C++, and Java specifically elected to distinguish a reference to an array from a reference to a function. Square brackets are used to enclose subscript values.

  *Sum***:=***Sum***+***B***[***I***];**
- A reference to a multidimensional array differs from language to language also.

Language	Declaration(s)	Reference
C++	**int** *A***[3][5];**	**for (int** *r***=0;***r***<3;r++) {** **for (int** *c***=0;***c***<5;c++) {** *A***[***r***][***c***]=***R***.***Sample***();** **}** **}**

Language	Declaration(s)	Reference

Pascal	**type**
	  imatrix=**array[0..2,0..4] of**
	  integer;
	var *A*:*imatrix*;
		for *r*:=**0 to 2 do**
		begin
		  for *c*:=**0 to 4 do**
		  begin
		  A[*r*,*c*]:=*random*(**100**);
		  end
		end

- Two distinct types are employed to construct an array type.
  - the element type
  - the index type
- The element type can be any type.
- The index type must be an ordinal type that is implemented as an integer.

Language	Declaration and use

Pascal	**program p03;**
	  type
	  day_t*= (sunday*,*monday*,*tuesday*,*wednesday*
	  ,*thursday*,*friday*,*saturday*
	  );
	  activity_t=(*work*,*play*);
	  var
	  day:**array[***day_t***] of** *activity_t*;
	begin{p03}
	  day[*sunday*]:=*play*;
	  day[*monday*]:=*work*
	end.{p03}

- Ada and Pascal for-loops can use any ordinal type variable for counters.
- Most contemporary languages do not specify range checking of subscripts.
- Java, ML, and C# do specify range checks.
- By default, Ada checks the range of all subscripts.

6.5.3. Subscript Bindings and Array Categories

There are five categories of arrays, based on binding to subscript ranges, the binding to storage, and from where the storage is allocated.

Category	Description
static array	An array in which the subscript ranges are statically bound and storage allocation is static (allocated before run time).

	int *A***[10]; //A static array**
	int *main***()**
	{ return 0;
	}

fixed stack-dynamic array	An array in which the subscript ranges are statically bound, but the allocation is done at declaration elaboration time during execution.

	void *f***(void)**
	{ int *A***[10]; //A fixed stack-dymnamic array**
	}
	int *main***()**
	{ *f***();**
	return 0;
	}

stack-dynamic array	An array in which both the subscript ranges and the storage allocation are dynamically bound at elaboration time.

	void *f***(int** *sz***)**
	{ int *A***[***sz***]; //A stack-dymnamic array**
	}
	int *main***()**
	{ *f***(25);**
	return 0;
	}

fixed heap-dynamic array	An array in which the subscript ranges and the storage binding are both fixed after storage is allocated.

	void *f***(int** *sz***)**
	{ int* *A***=new int[***sz***]; //A fixed heap-dymnamic array**
	}
	int *main***()**
	{ *f***(25);**
	return 0;
	}

heap-dynamic array	An array in which the binding of subscript ranges and storage allocation is dynamic and can change any number of times during the array's lifetime.

- C and C++ permit all types of arrays except heap-dynamic arrays.
- Fortran 95 supports fixed heap-dynamic arrays.
- In Java, all arrays are fixed heap-dynamic arrays.
- C# supports heap-dynamic arrays via class ArrayList.

ArrayList intList **= new** *ArrayList***();**

6.5.4. Array Initialization

Language	Initialization Example
Fortran 94	Integer, Dimension (3) :: List = (/0,5,5/)
C,C++,Java,C#	**int** *list***[]={4,5,7,83};**
C,C++	**char*** *names***[]{"Bob","Jake","Darcie"};**
Java	*String***[]** *names* **= ["Bob","Jake","Darcie"];** **//"Bob","Jake", and "Darcie" are references to String objects**
Ada	*List***: array(1..5) of** *Integer* **:=(1,3,5,7,9);** *Bunch***:array(1..5) of** *Integer***:=(1=>17,3=>34,others=>0);**
Python	**[***x*******x* **for** *x* **in range(12) if** *x***%3==0]** **produces the array** **[0,9,36,81]**

6.5.5. Array Operations

- An array operation is an operation that operates on an array as a unit.
- Common array operations include:
  - assignment
  - catenation
  - equality and inequality comparison
  - slices
- C-based languages do not provide *any* array operations
  - except through the methods of Java, C++, and C#
- Perl supports array assignment
- Pascal provides array assignment for conformant arrays – for arrays that have the same index values.
- Ada supports array assignment, and catenation (&) where both operands have a single dimension
- Python's arrays are called lists. Operations on lists include
  - assignment
  - catenation (+)
  - element membership (**in**)
- Fortran 95 supports array operations called **elemental** because they are between pairs of elements. Elemental operations include:
  - addition (+): The sum of two arrays is an array having the same dimensions where each element in the sum is the sum of corresponding elements in the operands.
  - assignment
  - relational operators
  - arithmetic operators
- Fortran 95 also has libraries having the following operations
  - matrix multiplication
  - matrix transpose
  - vector dot product
- APL: arrays are central to APL

- Addition, subtraction, multiplication, and division are defined for vectors, an array having a single dimension, and matrices.
- Examples
  $A + B$ //adds scalars, vectors, or matrices
  V reverses the elements of a **V**ector.
  ΦM reverses the columns of a **M**atrix.
  ΦM reverses the rows of a **M**atrix.
  ϕM transposes the **M**atrix, ie the rows become columns and vice versa.
  $\div M$ inverts the **M**atrix.
- The . (dot) operator joins selected pairs of operators, for example
  $+.\times$ for two vectors the combined operator is the dot product
  $+.\times$ for two matrices the combined operator is matrix multiplication

6.5.6. Rectangular and Jagged Arrays

Term	Description
rectangular array	A multidimensional array in which all of the rows have the same number elements, all of the columns have the same number of elements, and so forth. Rectangular arrays model rectangular tables exactly.
jagged array	An array is which the lengths of the rows need not be the same. For example, a jagged matrix may consist of three rows, one with five elements, one with seven elements, and one with twelve elements. This also applies to the columns and higher dimensions.

- C, C++ and Java support jagged arrays but not rectangular arrays (Your instructor believes, contrary to the text, that C, C++, and Java do support rectangular arrays and that rectangular arrays appear many times more often than jagged arrays.

```
//A jagged array, I think
int** A=new int*[3];
A[0]=new int[3];
A[1]=new int[5];
A[2]=new int[12];

//A rectangular array
int A[3][12];
```

- Languages that support jagged arrays employ a separate pair of brackets for each dimension

```
A[2][7]
```

- Languages that support rectangular arrays define the subscript list to be enclosed in a single pair of square brackets where each subscript is separated by a comma. (Your instructor has an example in Pascal where a reference to an element in a matrix can be appears in both forms, ***A*[2][7]** and ***A*[2,7]**. Further, the element reference has no bearing on whether the array is jagged or rectangular.)

*A***[2,7]**

6.5.7. Slices

A slice of an array is some substructure of that array. Examples

Example	**Explanation**
int *A***[10];**	
*A***[2..7]**	Element 2, 3, 4, 5, 6, and 7 of array A.
int *A***[10][10];**	
*A***[2][5..9]**	Elements 5 – 9 of row 2.

- Python supports slices, including entire rows or columns, or consecutively numbered parts of rows or columns. Python also supports more complex slices of arrays including a slice that references every other element of vector.
- Fortran 95 supports complex slices including columns.
- Perl supports slices in two forms, a list of specific subscripts or a range of subscripts.
- Ruby supports slices with the slice method of its Array object.
- Ada supports slices of single dimensioned arrays with consecutive indexes.

6.5.8. Evaluation

- Arrays have been included in virtually all programming languages.
- Advances in arrays since their introduction in FORTRAN have been the inclusion of ordinal types as indexes.
- Newest arrays are associative arrays.

6.5.9. Implementation of Array Types

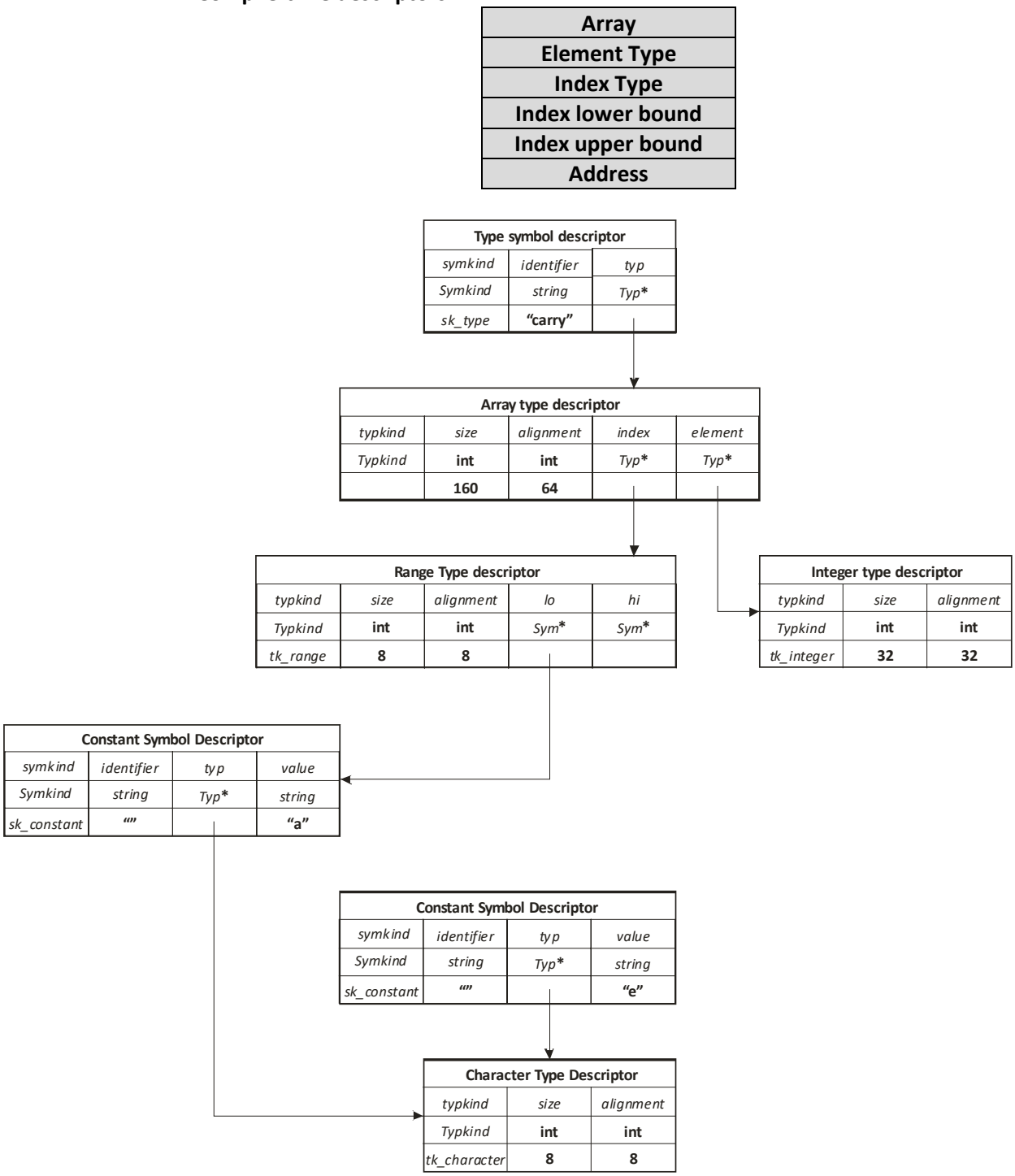
3 4 7
6 2 5
1 3 8

- **row major order**
  3, 4, 7, 6, 2, 5, 1, 3, 8
- **column major order**
  3, 6, 1, 4, 2, 3, 7, 5, 8

A	**1**	**2**	**…**	***j*-1**	***j***	**…**	***n***
1							
2							
…							
***i*-1**							
i					**X**		
…							
m							

- **location(A[I,j])**
  $location(A[i,j]) = address(A[1,1])$
  $+$ ((((# of rows above the ith row) $\times$ (size of a row)
  $+$ (number of elements left of the jth column)) $*$ element size)

- **Compile-time descriptors**



Array Type Descriptor

type *carray*=**array ['a'..'e'] of** *integer*;

6.6. Associative Arrays

6.6.1. Structure and Operations

- An **associative array** is an unordered collection of data elements that are indexed by an equal number of values called **keys**.
- Perl (called **hashes**)

```
%salaries = ("Gary"=>75000,"Perry"=>57000,"Mary"=>55750,"Cedric"=>47850);

$s=$salaries{"Perry"};
$s=57000
```

- C++

```
static map<string,int> RW;
…
#define CASE 260
RW["and"]=257;
RW["array"]=258;
RW["begin"]=259;
RW["case"]=CASE;
…
int TokenMgr(int t)
{   int tc=t;
    if (t==IDENTIFIER) {
        char* s=ToLower(yytext);
        tc=RW[s];
        if (tc==0) tc=t;
    }
    return tc;
}
```

6.6.2. Implementing Associative Arrays

- Perl, PHP, C++ - hash

6.7. Record Types

- A **record** is an aggregate of data elements in which the individual elements are identified by names and accessed through offsets from the beginning of the structure.
- In C, C++, and C# records are supported with the **struct** data constructor.
- Design issues that are specific to records include:
  - The syntactic form of references to fields
  - Are elliptical references allowed?

6.7.1. Definition of Records

- The fundamental difference between a record and an array is that a record has elements, or **fields**, are not referenced by indices. Instead, the fields are named with identifiers, and references to the fields are made using these identifiers.
- (Arrays are aggregates containing elements of the same type whereas records are aggregates containing elements of different types.)

- Aboriginal COBOL

```
01      EMPLOYEE-RECORD.
        02  EMPLOYEE-NAME.
            05  FIRST PICTURE IS X(20).
            05  MIDDLE PICTURE IS X(10).
            05  LAST PICTURE IS X(20).
        02  HOURLY-RATE PICTURE IS 99V99.
```

- Pascal

```
type name_type=array [1..20] of char;
type employee_name_type = record
   first,middle,last:nametype
end{employee_name_type};

type employee_record_type = record
     employee_name:employee_name_type;
     hourly_rate:real
end{employee_record_type};

var employee_record:employee_record_type;

employee_record.first:="Thomas      ";
employee_record.middle:="Alva       ";
employee_record.last:="Edison      ";
```

6.7.2. References to Record Fields

- COBOL – elliptical

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
MIDDLE OF EMPLOYEE-NAME
MIDDLE
```

- Pascal

```
employee_record.employee_name.first:="Thomas      ";
with employee_record.employee_name do
begin
     first:="Thomas      ";
     middle:="Alva       ";
     last:="Edison      "
end
```

Term	Description
fully qualified reference	A reference in which all intermediate record names, from the largest enclosing record to the specific field, are named in the reference.
elliptical reference	A reference to the specific field is given but any or all of the enclosing record names can be omitted, as long as the resulting reference is unambiguous.

6.7.3. Operations on Records

- COBOL
  MOVE CORRESPONDING
- C, C++, Java
  field assignment
  record assignment

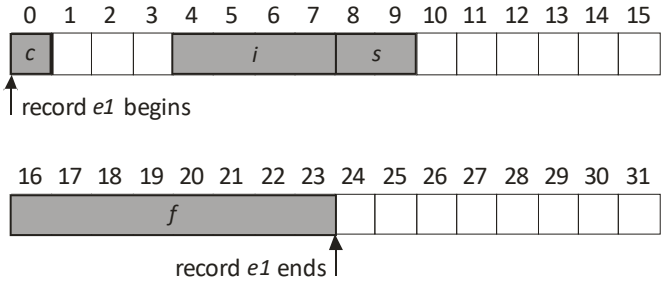
6.7.4. Evaluation

- Elliptical references allowed COBOL detract from readability
- Both records and arrays describe aggregate data. The difference is that elements of arrays typically share the same type where as a record is composed of arbitrary types.

6.7.5. Implementation of Record Types

- Fields in records are stored in adjacent memory locations.
- However, data of different types have different sizes and alignment specifications. For example IEEE 754 Double Binary data occupy 64 bits, or 8 bytes, and are aligned on 8-byte boundaries. Contrast the IEEE 754 Double Binary data to character data that often occupy one byte and are aligned on byte boundaries.

```
struct example1 {
    char c;
    int i;
    unsigned short s;
    double f;
};
example1 e1;
```



Record e1 layout

6.8. Tuple Types

A tuple is a data type that is similar to a record, except that the elements are not names.

Python tuple:

myTuple = (3, 5.8, ‘apple’)

myTuple[1] = 3

ML tuple:
val myTuple = (3, 5.8, 'apple');
ML tuples must have at least two elements.
#1(myTuple)=3

type intReal = int * real;

Values of this type consist of an integer and a real.

F#
let tup = (3, 5, 7);;
let a, b, c = tup;;
a=3, b=5,c=7

Tuples are used in Python, ML, and F# to allow functions to return multiple values.

6.9. List Types

Lists were first supported in the first functional programming language, LISP. They have always been part of the functional languages, but it recent years they have their way into some imperative languages.

List is Scheme and Common LISP are delimited by parentheses and the elements are not separated by any punctuation. For example

(A B C D)

Nested lists have the same form, so we could have

(A (B C) D)

Data and code have the same syntactic for in LISP and its descendants. If the list (A B C) is interpreted as code, it is a call to the function A with parameters B and C.

The fundamental list operations in Scheme are two functions that take lists apart and two that build lists. The CAR function returns the first element of its list parameter. For example:

(CAR '(A B C))

The quote before the parameter list is to prevent the interpreter from considering the list a call to the A function with the parameters B and C, in which case it would interpret it. This call to CAR returns A.

The CDR function returns its parameter list minus its first element. For example, consider

(CDR '(A B C))

This function call return the list (B C).

In Scheme and Common LISP, new lists are constructed with the CONS and LIST functions. The function CONS takes two parameters and returns a new list with its first parameter as the first element and its second parameter as the remainder of that list. For example, consider the following:

(CONS 'A '(B C))

This call returns the new list (A B C).

The LIST function takes any number of parameters and returns a new list with the parameters as its elements.

(LIST 'A 'B '(C D))

This call returns the new list (A B (C D))

ML lists [5, 7, 9]
[] – the empty list
CONS in ML is 3 :: [5, 7, 9] resulting in [3, 5, 7, 9]

Scheme	ML
CAR	hd (head)
CDR	tl (tail)

hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]

F# Operations hd and tl are the same, but they are called as methods of the List class as in List.hd [1; 3; 5; 7], which returns 1.

Python has a powerful method for creating arrays called **list comprehensions**.

[x * x for x in range (12) if x % 3 == 0]

The **range** function creates the array [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

The conditional filters out all numbers in the array that are not evenly divisible by 3.

The expression squares the remaining numbers resulting in the following array.

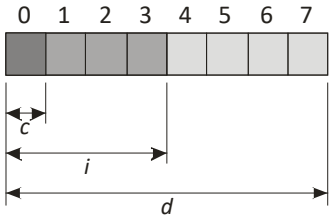
[0, 9, 36, 81]

6.10. Union Types

- A **union** is a type whose variables may store different type values at different times during program execution.
- A **union** is a type whose storage has multiple definitions.

```
union U {
    char c;
    int i;
    double d;
};
U u;
u.c='a';
u.i=5;
u.d=1.602e-19;
```

In the example above variable c occupies the most significant byte of integer *i* and floating point variable *d*.



union *U*

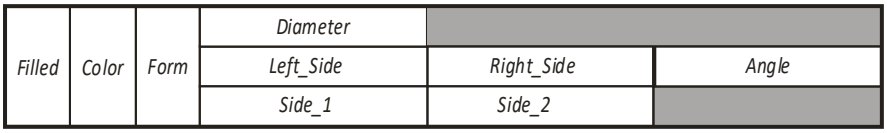
6.10.1. Design Issues

- A **union** is a type whose variables may store different type values at different times during program execution.

6.10.2. Discriminated Versus Free Unions

- A **free union** defines a type where any field in the union may be assigned without validating the field type. This is inherently dangerous because overwriting a part of the storage allocated to a field of a different type will likely destroy that field.
- A **discriminated union** contains an additional field, called the **discriminant**, that specifies which of the several types is currently valid.

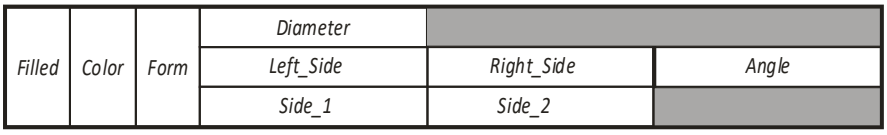
```
type Shape = (Circle,Triangle,Rectangle);
type Colour = (Red,Green,Blue);
type Figure = record
    Filled: Boolean;
    Color: Colour;
    case Form: Shape of
        Circle:(Diameter: real);
        Triangle:(Left_Side: integer;Right_Side: integer;Angle:real;);
        Rectangle:(Side_1: integer;Side_2: integer;);
end{Figure};
```



Discriminated Union *Figure* (Pascal)

6.10.3. Ada Union Types

```
type Shape is (Circle,Triangle,Rectangle);
type Colour is (Red,Green,Blue);
type Figure (Form: Shape) is
    record
        Filled: Boolean;
        Color: Colour;
        case Form is
            when Circle =>
                Diameter: Float;
            when Triangle =>
                Left_Side: Integer;
                Right_Side: Integer;
                Angle: Float;
            when Rectangle =>
                Size_1: Integer;
                Side_2: Integer;
        end case;
    end record;
```



Discriminated Union *Figure* (Ada)

- A **constrained variant variable** permits static type checking.

  Figure_2: Figure(Form => Triangle);

  The constrained variable Figure_2 can only be a triangle and cannot be changed to another variant.

- An **unconstrained variant variable** does not permit static type checking. However, consistency is maintained by allowing only entire records to be assigned. Individual fields may not be assigned in this variant record.

  Figure_1: Figure;

  Figure_1:=
  (Filled => True

```
, Color => Blue
, Form => Rectangle
, Side_1 => 12
, Side_2 =>3
);
```

- **Run-time (dynamic) type checking** of variant records detects the error arising from the if-statement below when the *Form*-tag was assigned the value *Circle*.

  if (*Figure_1***.***Diameter* **> 3.0) …**

6.10.4. Evaluation

- Unions are potential unsafe in some languages including Fortran, C, and C++ that are not strongly typed.
- Unions can be safely used in Ada because the design allows for static type checking and, if desired, run-time type checking.
- Java and C# do not permit unions

6.10.5. Implementation of Union Types

```
type Node (Tag:Boolean) is
    record
    case Tag is
        when True => Count: Integer;
        when False => Sum: Float;
    end case;
    end record;
```

6.11. Pointer and Reference Types

- A **pointer** type is one in which the variables have a range of values that consist of **memory addresses** and a special value, **nil**.
- Pointers are used for
  - Indirect addressing
  - Access dynamically allocated storage in a space commonly called a **heap**.
- Variables that are dynamically allocated from the heap are called **heap-dynamic variables**. They often do not have identifiers associated with them and thus can be referenced only by pointer or reference type variables. Variables without names are called **anonymous variables**.
- A pointer variable is distinguished from other types of variables and called a **reference type**. Other types of variables, including arrays and records, called structured types; and scalar variables are called **value types** because they store actual values rather than references to values.

6.11.1. Design Issues

- What are the scope and lifetime of a pointer variable?
- What is the lifetime of a heap-dynamic variable?
- Are pointers restricted as to the type of value to which they can point?

- Are pointers used for dynamic storage management, indirect addressing, or both?
- Should the language support pointer types, reference types, or both?

6.11.2. Pointer Operations

- Assignment
  int*p* **= new int;**

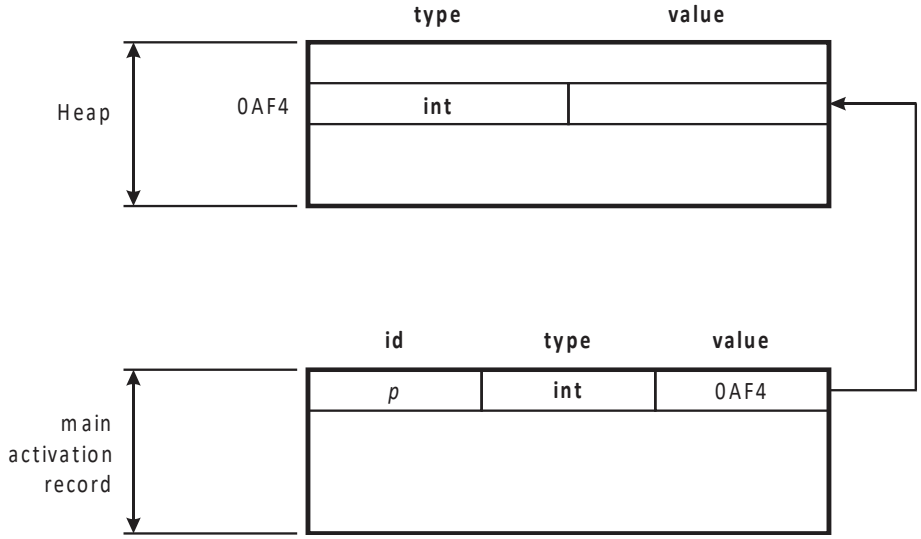


Figure 6.9.2 Pointer Assignment

- Dereferencing

6.11.3. Pointer Problems

PL/I introduced pointers. PL/I pointers were highly flexible and could point to both heap-dynamic variables and other program variables. To avoid problems with pointers, some newer languages, like Java, have replaced pointers completely with reference types, which, along with implicit deallocation, minimize the primary problems with pointers. A reference type is really only a pointer with restricted operations.

6.11.3.1. Dangling Pointers

A **dangling pointer**, or **dangling reference**, is a pointer that contains the address of a heap-dynamic variable that has been deallocated.

Dangers:

1. The location being pointed to may have been reallocated to some new heap-dynamic variable.
2. If the new variable is not the same type as the old one, type checks of uses of the dangling pointer are invalid.
3. If the dangling pointer is used to change the heap-dynamic variable, the value of the new heap-dynamic variable will be destroyed.
4. It is possible that the location now is being temporarily used by the storage management system, possibly as a pointer in a chain of available blocks of storage, thereby allowing a change to the location to cause the storage manager to fail.

Creating a dangling pointer.

1. A new heap-dynamic variable is created and pointer p1 is set to point at it.
2. Pointer p2 is assigned p1's value.
3. The heap-dynamic variable pointed to by p1 is explicitly deallocated but p2 is not changed by the operation. p2 is now a dangling pointer.

```
int* arrayPtr1;
int* arrayPtr2=new int[100];
arrayPtr1=arrayPtr2;
delete[] arrayPtr2;

//Now, arrayPtr1 is dangling.
```

6.11.3.2. Lost Heap-Dynamic Variables

A **lost heap-dynamic variable** is an allocated heap-dynamic variable that is no longer accessible to the user program. Such variables are often called **garbage**.

1. Pointer p1 is set to point to a newly created heap-dynamic variable.
2. p1 is later set to point to another newly created heap-dynamic variable.

```
int* p1=new int[100];
p1=new int;
//The array created by new int[100] is lost and is now garbage.
```

6.11.4. Pointers in Ada

Ada's pointers are called **access** types. The dangling-pointer problem is partially alleviated by Ada's design, at least in theory. A heap-dynamic variable may be (at the implementer's option) implicitly deallocated at the end of the scope of its pointer type; thus, dramatically lessening the need for explicit deallocation.

However, few if any Ada compilers implement this form of garbage collection, so the advantage is nearly always in theory only.

Unfortunately, the Ada language also has an explicit deallocator, Unchecked_Deallocation. Unchecked_Deallocation can cause dangling pointers.

Lost heap-dynamic variables are not eliminated by Ada's design.

6.11.5. Pointers in C and C++

In C and C++, pointers can be used in the same ways as addresses are used in assembly languages.

C and C++ permit pointer arithmetic making their use more interesting in those languages compared to languages that do not permit pointer arithmetic.

```
int* ptr;
int count, init;
…
ptr =&init;        //assign the address of init to ptr
count = *ptr;      //dereference the value stored in variable ptr to refer to the value stored
                   //in init.  Assign the value in init to count.
```

Pointer arithmetic in C and C++

```
int list[10];
int* ptr;
ptr=list;
```

- *(ptr+1) is equivalent to list[1]
- *(ptr+index) is equivalent to list[index]
- ptr[index] is equivalent to list[index]

C and C++ include pointers of type **void***, which can pint at values of any type.

6.11.6. Reference Types

A reference type variable is similar to a pointer, with one important and fundamental difference: A pointer refers to an address in memory, while a reference refers to an object or value in memory. As a result, although it is natural to perform arithmetic on address, it is not sensible to do arithmetic on references.

C++ includes a special kind of reference type that is used primarily for the formal parameters in function definitions. A C++ reference type variable is a constant pointer that is always implicitly dereferenced.

```
int result = 0;
int& ref_result=result;
ref_result=100;
```

In the foregoing code segment result and ref_result are aliases.

In their quest for increased safety over C++, the designers of Java removed C++-style pointers altogether. Java reference variables can be assigned to refer to different class instances: they are not constants. All Java class instances are deferenced by reference variables. That is, in fact, the only use of reference variables in Java.

```
String str2;
```

str1 = “This is a Java literal string”;

In this code, str1 is defined to be a reference to a String class instance or object. It is initially set to null. The subsequent assignment sets str1 to reference the String object, “This is a Java literal string”.

Because Java class instances are implicitly deallocated, there cannot be dangling references in Java.

C# includes both the references of Java and the pointers of C++.

6.11.7. Evaluation

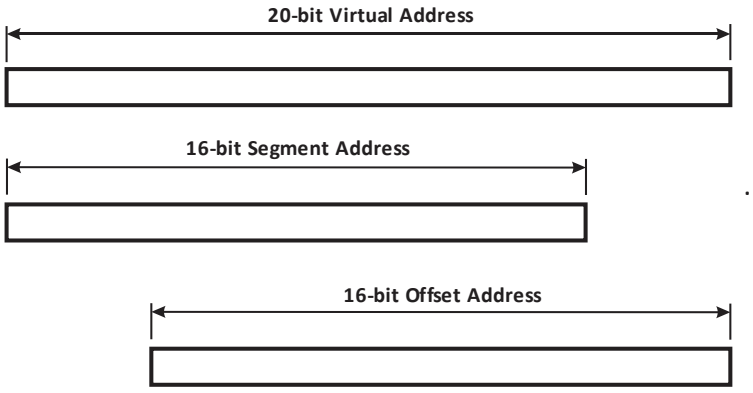
The problems of dangling pointers and garbage have already been discussed at length.

Pointers have been compared with the goto. The goto statement widens the range of statements that can be executed next. Pointer variables widen the range of memory cells that can be referenced by a variable. Perhaps the most damning statement about pointers was made by Hoare (1973): “Their introduction into high-level languages has been a step backward from which we may never recover.”

6.11.8. Implementation of Pointer and Reference Types

6.11.8.1. Representations of Pointers and References

Machine	Representation
Typical	Single values stored in memory cells
Intel 8086	Addresses have two parts, segment and offset.
IBM System/38	16-byte capability address contains security and permission



Intel 8086 Segment and Offset Registers.

6.11.8.2. Solutions to the Dangling-Pointer Problem

Name	Description
Tombstone	Every heap-dynamic variable includes a special cell, called a tombstone that is itself a pointer to the heap-dynamic variable. The actual pointer variable points only at tombstones and never to heap-dynamic variables. When a heap-dynamic variable is deallocated, the tombstone remains but is set to `nil`, indicating that the heap-dynamic variable no longer exists. This approach prevents a pointer from ever pointing to a deallocated variable. Any reference to any pointer that points to a `nil` tombstone can be detected as an error.
Lock-and-key	Pointer values are represented as ordered pairs (key, address), where the key is an integer value. Heap-dynamic variables are represented as the storage for the variable plus a header cell that stores an integer lock value. When a heap-dynamic variable is allocated, a lock value is created and placed both in the lock cell of the heap-dynamic variable and in the key cell of the pointer that is specified in the call to new. Every access to the dereferenced pointer compares the key value of the pointer to the lock value in the heap-dynamic variable. If they match, the access is legal; otherwise, the access is treated as a run-time error. Any copies of the pointer value to other pointers must copy the key value. Therefore, any number of pointers can reference a give heap-dynamic variable. When a heap-dynamic variable is deallocated with dispose, its lock value is cleared to an illegal lock value. Then, if a pointer other the one specified in the dispose is dereferenced, its address value will still be intact, but its key value will no longer match the lock, so the access will not be allowed.

6.11.8.3. Heap Management

Single-Size Cells – Lisp: The simplest situation is when all allocation and deallocation is of single-size cells. It is further simplified when every cell already contains a pointer. This is the scenario of many implementations of LISP, where the problems of dynamic storage allocation were first encountered on a large scale. All LISP programs and most LISP data consist of cells in linked lists.

In a single-size allocation heaps, all available cells are linked together using the pointers in the cells, forming a list of available space. Allocation is a simple matter of taking the required number of cells from this list when they are needed. Deallocation is a much more complex process. A heap-dynamic variable can be pointed to by more than one pointer, making it difficult to determine when the variable is no longer useful to the program. Simply because one pointer is disconnected from a cell obviously does not make it garbage; there could be several other pointers still pointing to the cell.

Garbage collection

- **Reference counters**, in which reclamation is incremental and is done when inaccessible cells are created.
- **Mark-sweep**, in which reclamation occurs only when the list of available space becomes empty.
    - First, all cells in the heap have their indicators set to indicate they are garbage.
    - The second part, called the marking phase, is the most difficulty. Every pointer in the program is traced into the heap, and all reachable cells are marked as not being garbage.
    - The third phase, called the sweep phase, is executed. All cells in the heap that have not been specifically marked as still being used are returned to the list of available space.
- **Incremental mark-sweep** is similar to mark-sweep but garbage collection occurs more frequently, long before memory is exhausted, making the process more effective in terms of the amount of storage that is reclaimed. Also, the time required for each run of the process is obviously sorter, thus reducing the delay in application execution.

Variable-Size Cells – Most Programming Languages

- Because the cells are different sizes, scanning them is a problem. One solution is to require each cell to have the cell size as the first field. Scanning can be accomplished but requires slightly more storage and more time in comparison with fixed sized cells.
- The marking process is challenging. How can a chain be followed from a pointer if there is no predefined location for the pointer in the referenced cell? Cells that contain no pointers are a problem. Adding an internal pointer to each cell, which is maintained in the background by the run-time system, will work. However, background maintenance adds space and execution time overhead.
- Maintaining the list of available space is another source of overhead. The list can begin with a single cell consisting of all available space. Requests for segments simply reduce the size of this block. Reclaimed cells are added to the list. The problem is that before long, the list becomes a long list of various size segments, or blocks. This slows allocation because requests casus the list to be searched for sufficiently large blocks. Eventually, the list may consist of a large number of very small blocks, which are not large enough for most requests. At this point, adjacent blocks may need to be collapsed into larger blocks.

6.12. Type Checking

- **Type checking** is the activity of ensuring that operands of an operator are of *compatible* types.
- A **compatible** type is one that either is legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type.
- Automatic conversion is called **coercion**. For example, if an `int` variable and a float variable are added in Java, the value of `int` variable is coerced to `float` and a floating-point addition is performed.

- A **type error** is the application of an operator to an operand of an inappropriate type. For example, in C++ is a variable having an `array` type is multiplied by a variable having a `struct` type (a record), an compile-time error is produced.
- **Static type checking**: If all bindings of variables to types are static in a language, then type checking can nearly always be done statically – during compilation.
- **Dynamic type checking**: Dynamic type binding requires type checking at run time, called dynamic type checking. Some languages, such as JavaScript and PHP, because of their dynamic type binding, allow only dynamic type checking. It is better to detect errors, *earlier*, at compile time, because the sooner an error is found and corrected the less costly it is to fix.
- Type checking is complicated when a language allows a memory cell to store values of different types at different times during execution: Such memory cells can be created with Ada variant records, C and C++ unions, and discriminated unions in ML, Haskell, and F#. In these cases, type checking, *if done*, must be dynamic and requires the run-time system to maintain the type of the current value of such memory cells. So, even though all variables are statically bound to types in languages such a C++, not all type errors can be detected by static type checking.

6.13. Strong Typing

The concept of structured programming was developed in the 1970s and with it the idea of **strong typing**. A programming language is strongly typed if type errors are always detected. This requires that the types of all operands can be determined, either at compile time or at run time. The motivation to detect all type errors is to reduce the costliest part of program development – the cost of testing.

Ada is nearly strongly types. It is only *nearly* strongly typed because it allows programmers to breach the type-checking rules by specifically requesting that type checking be suspended for a particular type conversion.

C and C++ are not strongly typed languages because both include union types, which are not checked.

ML is strongly typed, even though the types of some function parameters may not be known at compile time. F# is strongly typed.

Java and C#, although they are based on C++, are strongly typed in the same sense as Ada. Types can be explicitly cast, which could result in a type error.

6.14. Type Equivalence

Two types are equivalent if an operand of one type in an expression is substituted for one of the other type, without coercion. Type equivalence is a strict form of type compatibility – compatibility without coercion.

Two approaches to defining type equivalence:

Name type equivalence means that two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name. Name type equivalence is easy to implement but is more restrictive.

Ada:

```
type Indextype is 1..100;
```

count: Integer;
index: Indextype;

The types of the variables count and index would not be equivalent; count could not be assigned to index or vice versa.

Structure type equivalence means that two variables have equivalent types if their types have identical structures.

```
const RANK=100;
type
    ivector = array[1..RANK] of integer;
var
    IV: ivector;
    OV: array[1..RANK] of integer;
```

Using structure type equivalence variables IV and OV are equivalent but under name type equivalence they are not.

Under structure type equivalence, however, the entire structures of the two types must be compared. This comparison is not always simple. For example, are two record types equivalent if they have the same structure but different field names? Are two single-dimensioned array types in a Fortran or Ada program equivalent if they have the same element type but have subscript ranges of 0..10 and 1..11? Are two enumeration types equivalent if they the same number components but spell the literals differently?

Another difficulty with structure type equivalence is that it disallows differentiating between types with the same structure. For example, consider the following Ada-like declarations:

```
type    Celsius     =  Float;
        Fahrenheit  =  Float;
```

The types of variables of the two types are considered equivalent under structure type equivalence, allowing them to be missed in expressions, which is surely undesirable in this case, considering the difference indicated by the type's names.

A **derived type** is a new type that is based on some previously defined type with which it is not equivalent, although it may have identical structure. Derived types inherit all the properties of their parent types.

```
type Celcius is new Float;
type Fahrenheit is new Float;
```

The types of variables of these two derived types are not equivalent, although their structues are identical. Furthermore, variables of both types are not type equivalent with any other floating-point type.

An Ada subtype is a possibly range-constrained version of an existing type. A subtype is equivalent with its parent type.

subtype *Small_type* **is** *Integer* **range 0..99;**

The type *Small_type* is equivalent to the type *Integer*.

type *Derived_ Small _Int* **is new** *Integer* **range 1..100;**
subtype *Subrange_Small_Int* **is new** *Integer* **range 1..100;**

Variables of both types have the same range of legal values and both inherit the operations of Integer. However, variables of type Derived_Small_Int are not compatible with any Integer type.

On the other hand, variables of type Subrange_Small_Int are compatible with variables and constants of Integer type and any subtype of Integer.

Ada unconstrained array types:

type *Vector* **is array (***Integer* **range <>) of** *Integer***;**

*Vector_1***:** *Vector***(1..10);**
Vector_2:*Vector***(11..20);**

The types of these two objects are equivalent, even though they have different subscript ranges, because for objects of unconstrained array types, structure type equivalence rather than name type equivalence is used. Because both types have 10 elements and the elements of both are of type Integer, they are type equivalent.

Consider

*A***: array(1..10) of** *Integer***;**
*B***: array(1..10) of** *Integer***;**

Both *A* and *B* have anonymous types – types that have no unique name – but even though the anonymous types are identical arrays *A* and *B* are not compatible. The elements of the arrays are compatible but not the arrays themselves.

*C***,** *D***: array(1..10) of** *Integer***;**

Arrays *C* and *D* are not equivalent because Ada treats the foregoing declaration as

*C***: array(1..10) of** *Integer***;**
*D***: array(1..10) of** *Integer***;**

To make the arrays equivalent, we must use the following declarations:

type *List_10* **is array (1..10) of** *Integer*;
C, *D*: *List_10*;

6.15. Theory and Data Types

under construction