BAB V
PEMROGRAMAN BERORIENTASI OBYEK

Pemrograman beriorientasi obyek (*Object Oriented Programming* = OOP) berbeda dengan pemrograman konvensional pada umumnya, terutama dalam memperlakukan prosedur dan data. Pada pemrograman biasa, prosedur dan data merupakan dua hal yang dipisahkan satu sama lain. Sebagai contoh, untuk mengelola data waktu yang terdiri dari jam, menit, dan detik dapat dibuat suatu struktur data dalam C sebagai berikut:

```
struct Time {
    int hour;       // 0-23
    int minute;     // 0-59
    int second;     // 0-59
};
```

Definisi ini terdiri dari tiga data, dimana untuk mengelolanya dibutuhkan prosedur yang disusun pada bagian yang terpisah. Salah satu konsep OOP yang paling penting adalah membungkus prosedur dan data menjadi satu software obyek. Konsep ini disebut sebagai ***encapsulation***.

OOP memodelkan obyek yang ada di dunia nyata (*real-word objects*) ke dalam software obyek dalam pemrograman. Oleh karena itu, di dalam OOP juga dikenal istilah seperti yang ada pada obyek dunia nyata, yaitu pewarisan (***inheritance***), dimana suatu obyek dapat mewariskan sifat-sifat yang dimilikinya kepada obyek turunannya.

Secara umum, beberapa keuntungan yang dapat diperoleh pada OOP antara lain adalah simplicity, modularity, modifiability, extensibility, flexibility, maintainability, dan reasonability. Bahasa pemrograman yang mendukung OOP antara lain adalah: Smalltalk (murni OOP), C++, CLOS (Common Lisp Object System), Java, dan sebagainya. Dalam buku ini digunakan bahasa pemrograman C++ untuk memahami implementasi OOP.

BAHASA PEMROGRAMAN C++

Bahasa permrograman C++ dikembangkan di AT&T Bell Laboratiries pada awal 1980-an oleh Bjarne Storoustrup, yang merupakan evolusi dari bahasa pemrograman C. Tambahan-tambahannya terhadap fasilitas – fasilitas C antara lain adalah:

1. fasilitas untuk membuat dan menggunakan abstraksi –abstraksi data
2. fasilitas untuk design dan pemrograman berorientasi object
3. macam-macam perbaikan tehapadap berbagai macam fasilitas-fasilitas yang telah ada dalam C

Dalam C++, suatu aksi dilakukan dengan suatu ekspresi dimana ekspresi yang diakhiri dengan titk koma membentuk suatu ***statement*** atau pernyataan. Contoh-contoh statement antara lain adalah:

```
int nilai ;                 (declaration statement)
nilai = 7 + 1;              (assignment statement)
cout << nilai ;             (output statement)
```

Beberapa statement dapat dikelompokkan secara *logic* menjadi ***function***. Setiap program C++ mulai eksekusi dengan statement pertama dari **main()**. Sebuah fungsi terdiri dari empat bagian, yaitu: *return type*, *function name*, *argument list*, dan *function body*. Tiga bagian yang pertama membentuk apa yang disebut sebagai ***function prototype***. *Argumnet list* dibatasi sepasang kurung biasa () dan dapat terdiri dari nol atau lebih argumen yang masing-masing dipisahkan oleh tanda koma (,). Sedangkan *function body* dibatasi oleh sepasang kurung kurawal { } dan terdiri dari sebarisan statement. Berikut adalah contoh program C++ sederhana:

```
#include<iostreamm.h>

void read2 (int & , int &) ;                    // forward declaration
int max (int , int );                           // prototype
void writeMax (int );

main ( )
{    int val1 , val2;                                   // symbolic variables
     read2 ( val1 , val2 )  ;
     int maxVal = max (val1 , val2 ) ;
          writeMax (maxVal) ;
          return 0;
}

void read2 ( int & v1, int & v2)
{
     cout << " Ketikkan dua bilangan bulat : ";
     cin >> v1 >> v2 ;
}

int max (int v1 , int v2 )
{
     if (v1 > v2)
          return v1;
     else
          return v2 ;
}

void writeMax  ( int val )
{
     cout << val << "adalah yang terbesar .\n" ;
}
```

Dengan sistem operasi UNIX, setelah program itu dimasukkan dalam berkas dengan nama berakhiran .C (misalnya pr1.c), program dapat dikompilasi dengan perintah:

```
CC –o pr1  pr1.c
```

untuk menghasilkan executable file :pr1. Program itu dapat dijalankan dengan perintah **pr1**. Selain kompilator dan perintah-perintah C++, sebuah implementasi C++ juga memberikan sekumpulan *standard library* (pustaka baku). Pustaka ini merupakan sekumpulan fungsi yang sudah dikompilasikan.

Dalam program tadi, **iostream.h** adalah suatu *header file*. File ini mengandung informasi tentang I/O (misalnya **cout**) yang diperlukan dalam program itu. Input dari terminal pemakai (*standard input*) dikaitkan dengan iostream **cin**. Output ke terminal pemakai (*standard output*) dikaitkan dengan iostream **cout**. Iostream **cerr** juga dikaitkan dengan terminal pemakai dan dipakai untuk menampilkan pesan-pesan kesalahan.

Operator output (<<) dipakai untuk menyalurkan nilai ke *standard output* dan operator input (>>) dipakai untuk menyalurkan nilai dari *standard input*. Dan komentar yang mencakup banyak baris dibatasi oleh pasangan /* … */, sedangkan komentar sebaris cukup diawali dengan tanda dua garis miring (//).

Preprocessor Directives

Header files mengandung informasi yang diperlukan untuk penggunaan pustaka. Untuk mengakses variabel atau fungsi yang didefenisikan dalam *standard libraries*, kita harus menyertakan *header file* yang bersangkutan dalam program. *Header file* dapat disertakan dalam program dengan menggunakan *include directives. Directive* ditentukan dengan meletakkan tanda # pada kolom pertama. Directives diproses sebelum kompilator bahasa diaktifkan. Program yang memproses directve disebut *preprocessor*.

Include directive berfungsi membaca isi dari file yang disebutkan. Contoh:

```
#include <iostream.h>
#include ”time.h”
```

Jika nama file diapit oleh tanda **<…>,** file itu dianggap file yang baku dan akan dicari dalam direktori-direktori baku seperti yang didefiniskan oleh kompilator. Sedangkan jika nama file diapit oleh tanda kutip “…“, file itu dianggap berasal dari pengguna dan akan dicari dalam *current directory*.

Tipe Data

Sebuah bit terdiri dari sel memori yang hanya dapat menyimpan nilai 0 dan 1. Sebarisan bit diberi arti berdasarkan tipe dari nilai yang direpresentasikan dan biasanya disebut sebagai data. *Predefined data types* yang dikenal di C++ adalah:

1. Tipe integral: **char**, **short**, **int**, **long**
2. Tipe non-integral: **float**, **double**, **long double**

Tipe integral dapat berbentuk **signed** atau **unsigned**. Di dalam bentuk signed, bit yang paling kiri berlaku sebagai *sign bit*, dimana 0 menunjukkan positif dan 1 menunjukkan

negatif. Berdasarkan alokasi memori, tipe **char** biasanya menempati 1 byte memori, **int** menempati 1 word (2 bytes), **float** biasanya menempati 1 word memori (4 bytes), dan **double** menempati 2 word memori (8 bytes).

Sebuah nilai seperti 3 merupakan suatu *literal constant*. Disebut literal karena hanya berupa nilai dan bersifat konstan (tidak dapat diubah). Setiap literal mempunyai tipe, misalnya 0 bertipe integer, 1.24 bertipe double. *Literal constant* bersifat *nonaddressable*, karena alaamtnya tidak dapat diakses.

Literal integer constant dapat ditulis dalam notasi desimal, oktal, atau heksa decimal. Sebagai contoh, 20 (desimal), 024 (oktal), dan 0x14 (heksadesimal). Secara otomatis, *literal integer constant* bertipe **signed int**. Notasi L dapat dipakai untuk menentukan tipe **long** dan U untuk menetukan tipe **unsigned** seperti contoh: 3.14 (notasi desimal biasa), 1.0E-3 (notasi ilmiah), 1.0L (long), dan sebagainya.

Literal Character Constant yang dapat dicetak dapat ditulis dalam tanda kutip '..', misalnya 'a', '2', ',', dan '' (kosong). Karakter *Nonprintable, single quatiation*, dan *double quatiation mark* direpresentasikan dengan *escape sequences* sebagai berikut:

New line	\n
Horizontal tab	\t
Vertical tab	\v
Backspace	\b
Carriage return	\r
Formfeed	\f
Alert (bell)	\a
Backslash	\\
Tanda tanya	\?
Single quote	\'
Double quote	\"

Secara umum *escape sequence* dapat ditulis dengan digit oktal, yaitu **kkk** dimana kkk adalah barisan yang terdiri dari satu sampai tiga digit oktal. Nilai dari digit-digit oktal itu adalah nilai numerik dari karakter yang bersangkutan dalam *machine's character set*, misalnya dalam ASCII character set: \7 (bell), \0 (null), \13 (newline), dan \062 ('2').

String literal constant terdiri dari nol atau lebih karakter yang ditulis di dalam tanda kutip dobel "…". Backslash sebagai karakter terakhir pada suatu baris menyatakan bahwa string tersebut berlanjut ke baris berikutnya. Perhatikan contoh berikut:

```
"" (null string)
"a"
" \n  Program \t CPP \n"
"string ini \
panjang \
sekali "
```

Tipe string literal adalah array dari karakter yang diakhiri dengan karakter **null** ('**\0**'), misalnya 'a' merepresentasikan karakter tunggal a, sedangkan "a" merepresentasikan karakter a diikuti dengan \0.

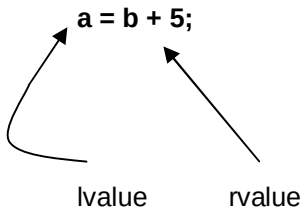
Symbolic Variable

Variable diidentifikasi dengan nama yang diberikan oleh pengguna, dan setiap variable mempunyai tipe tertentu, misalnya pernyataan berikut mendeklarasikan sebuah variabel ch dengan tipe char:

```
char ch;
```

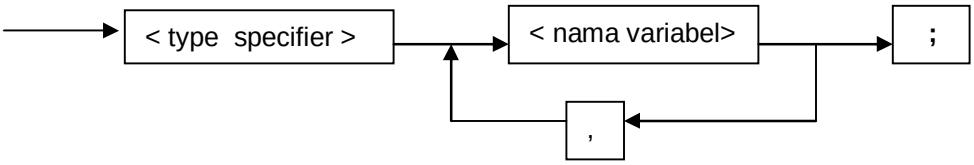
Char adalah suatu penentu tipe seperti halnya dengan **short, int, long, float,** dan **double**. Tipe data menentukan jumlah memori yang dialokasikan untuk variabel yang bersangkutan dan operasi-operasi yang dapat dilakukan pada variabel tersebut. Baik variabel ataupun *literal constant* memerlukan lokasi penyimpanan dan mempunyai tipe. Perbedaanya adalah bahwa variabel bersifat *addressable* sehingga terdapat dua nilai yang diasosiasikan dengan sebuah variabel, yaitu:

1. Nilai data yang disimpan pada lokasi memori, dan biasa disebut dengan **rvalue**
2. Nilai lokasi, yaitu alamat dari lokasi dimana nilai data itu tersimpan, dan biasa disebut sebagai **lvalue**



Variabel harus dideklarasikan atau didefenisikan dalam program sebelum dapat dipakai. Definisi dari suatu variabel menyebabkan memori dialokasikan, sedangkan deklarasinya tidak menybabkan memori dialokasikan.

Sintaks defenisi variabel dapat digambarkan sebagai berikut:



Contoh:

```
int hari;
int bulan;
int tahun;
```

dapat ditulis sebagai: `int hari, bulan, tahun;`

```
unsigned long jarak;
```

Variabel juga dapat diinisialisasikan pada waktu pendefinisian sehingga variabel dapat diinisialisasikan dengan suatu ekspresi. Contoh:

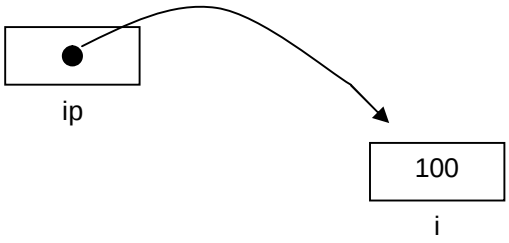
```
int  nilai = -108;
double harga = 15.9, potongan = 0.2;
unsigned nilaiAbs = abs (nilai);
```

Tipe Pointer

Variabel pointer berisi alamat dari obyek-obyek dalam memori. Setiap pointer memiliki tipe, dimana tipe ini menentukan tipe dari obyek data yang ditunjuk oleh pointer itu. Pointer int misalnya, akan menunjuk ke obyek yang bertipe int. Lokasi yang dialokasikan untuk sebuah pointer bergantung pada ukuran yang diperlukan untuk menyimpan suatu alamat dari memori, dan tidak bergantung pada tipe obyeknya.

Definisi sebuah pointer ditentukan dengan mendahului identifier dengan *derequrence operator* yang dituliskan dengan menggunakan tanda bintang (*). Perhatikan contoh berikut:

```
int  i = 100 ;
int *ip = &i;              // ip menunjuk ke i
int k = *ip ;              // k berisi 100
*ip = -5 ;                 // i = -5
*ip = abs ( *ip );         // i = abs(i)
ip = ip + 2 ;              // menambah address yang dikandung ip 2 ukuran object
*ip = *ip + 2;             // i = i +2
```



String dan Pointer

Tipe dari suatu *literal string constant* adalah pointer ke karakter pertama dari string itu, sehingga setiap konstanta string bertipe **char *** seperti pada contoh berikut:

```
char  *st = "reformasi\n";
```

Untuk memahami konsep pointer di dalam string, perhatikan dua program berikut dimana program pertama menunjukkan adanya kesalahan (mengapa?) dan diperbaiki di dalam program kedua (apa keluaran program?).

```
// Program versi 1
#include <iostream.h>
char *st = "reformasi\n";
main ()
{
     int pj =  0
     while ( st++ != '\0' ) ++pj ;
     cout << pj << ":" << st;
     return 0;
}
```

```
// Program versi 2
#include <iostream>
char *st = “reformasi\n”;
main ()
{
     int pj =  0;
     char *p = st;
     while ( *p++)
          ++pj;
     cout << pj << “:“ << st;
     return 0 ;
}
```

Tipe-tipe Konstanta

Kata kunci **const** dipakai untuk menentukan suatu tipe konstan, misalnya:

```
const int bufSize = 512;          // bufSize adalah suatu konstanta simbolik betipe int
```

Nilai dari konstanta simbolik tidak dapat diubah dan harus diinisialisasikan pada waktu pendefenisian. Disamping itu, tidak boleh meng-*assign* alamat dari suatu konstanta simbolik ke suatu pointer biasa. Perhatikan contoh-contoh berikut:

```
bufSize = 1024 ;                  // error
const double pi;                  // error
const double upah = 3.8 ;
double *p= &upah ;                // error
```

Namun demikian, pemrogram boleh mendeklarasikan pointer yang ditunjuk oleh konstanta seperti pada contoh berikut:

```
cons double *pc ;
```

dimana **pc** adalah pointer ke obyek **const** bertipe double, walaupun pc sendiri bukan merupakan suatu konstanta. Hal ini berarti pc dapat diubah untuk menunjuk ke obyek lain yang bertipe double, dan nilai dari obyek yang ditunjuk oleh pc tidak dapat ditambah melalui pc.

```
const double *pc;
pc = &upah;
double d;
pc =&d;
d = 3.14;
*pc = 3.14                        // error.
```

Pointer Konstan
Perhatikan contoh berikut:

```
int nomor , nomor1 ;
int *const pn = &nomor ;
```

Variabel pn adalah pointer konstan yang menunjuk ke obyek bertipe integer. Nilai dari obyek yang ditunjuk oleh pn dapat dirubah tetapi alamat yang dikandung pn tidak dapat dirubah, misalnya:

```
*pn  = 0   ;
pn = &nomor1 ; //error
```

Bentuk lain dari tipe konstan ini adalah pointer konstan ke obyek konstan dimana baik nilai obyek yang ditunjuk maupun alamat yang dikandung pointer tidal dapat dirubah seperti contoh berikut:

```
const int pass = 1 ;
const int *const cpc =&pass ;
```

Tipe Referensi
Suatu tipe referensi (*reference type)* didefinisikan dengan memberikan oparator address of yang dinotasikan dengan karakter "**&**" setelah penentu tipe. Seperti halnya dengan konstanta, obyek referensi harus diinisialisasi.

Suatu obyek referensi sering disebut sebagai suatu alias karena ia berlaku sebagai nama alternatif untuk obyek yang menginisialisasinya. Semua operasi yang dilakukan pada obyek referensi dilakukan juga pada obyek alias. Secara implisit, setiap obyek referensi bersifat konstan sehingga dapat diinisialisasi hanya satu kali. Penggunaan utama dari tipe referensi ini adalah sebagai tipe argumen atau *return-type* dari suatu fungsi. Perhatikan contoh-contoh berikut:

```
int val =10;
int &refVal = Val;          // lvalue yang diambil
int &refVal2;               // error karena belum diinisialisasi

refVal + = 2;               // nilai val menjadi 12
int  ii  = refVal;          // ii diberi nilai dari Val
int *pi = &refVal ;         // pi menunjuk ke val
```

Tipe Enumerasi
Suatu tipe enumerasi mendeklarasikan suatu himpunan konstanta-konstanta bulat simbolik. Unsur dari suatu enumerasi disebut enumerator. Suatu enumerasi dideklarasikan dengan kata kunci **enum** dan sebarisan enumerator yang diapit oleh sepasang kurung kurawal {…}. Secara otomatis, enumerator pertama diberi nilai 0, tiap enumerator berikutnya diberikan nilai 1 lebih dari nilai yang diberikan padaenumerator yang mendahuluinya, misalnya:

```
enum { false, true };       // false (0), true (1)
```

Suatu enumerator dapat diberi satu nilai tertentu secara eksplisit dan nilai ini tidak harus unik, misalnya:

```
enum { false, fail = 0, pass, true = 1true };
```

Suatu enumerator dapat diberi suatu nama pengenal (*tag name*). Enumerasi yang bernama merupakan suatu tipe yang unik dan nama itu dapat dipakai sebagai penentu tipe. Perhatikan contoh penggunaan tipe enumerasi pada cuplikan program berikut:

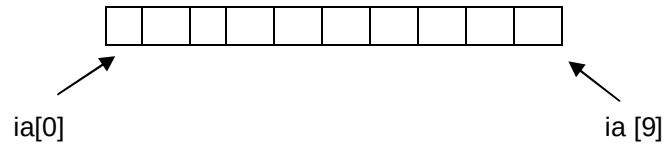
```
enum testStatus {not_run = -1, fail, pass};
enum boolean { false, true };
main()
{
    const testSize = 50;
    testStatus testSuite [testSize];
    boolean found = false;
    testSuite[0] = not_run;
    …
    …
    …
}
```

Tipe Array

Sebuah array terdiri dari sekumpulan obyek bertipe sama, dimana obyek itu diakses melalui posisinya dalam array tersebut. Misalnya:

```
int ia[10];
```

mendefenisikan suatu array yang terdiri dari 10 obyek integer. Elemen-elemen suatu array diindeks mulai dari 0 seperti yang ditunjukkan pada diagram berikut:



Berikut disajikan contoh beberapa penggunaan tipe array:

```
extern int getSize();
const  int  bufSize = 512, maxFiles = 20;
int staffSize = 25;
chah inputBuffer [bufSize];
char *fileTable [maxFiles - 3];
double upah [staffSize];                    // error
int test [getSize()];                       // error
```

Array dapat diinisialisasi secara eksplisit, dan dapat pula diinisialisasi secara eksplisit tanpa mencantumkan nilai dimensinya karena kompilator yang akan menghitungnya secara otomatis. Perhatikan contoh-contoh berikut:

```
const ukuran = 3 ;
int ia[ukuran] = { -1 , 0 , 1 };
```

atau dapat ditulis sebagai:

```
int ia [] = {-1, 0, 1} ;
```

Contoh-contoh lain adalah:

```
const size = 5;
int  ia[size] = {1, 2, 3 } ;          // ia → {1, 2, 3, 0, 0}
char c1[] = { 'C' , '+' ,'+' }        // dimensi 3
char c2[] = "C++" ;                   // dimensi 4
har c3[5] = "Usman";                  // error, karena dimensinya kurang
```

Sifat-sifat lain dari suatu array adalah bahwa array tidak dapat diinisialisasi dengan array lain, tidak dapat di-*assign* ke array lain, dan *array-of-references* tidak diperbolehkan. Perhatikan contoh cuplikan program berikut berikut:

```
const  int size = 3;
int ix, iy, iz;
int &iar[] = { ix, iy, iz };          //error
int  ia[] = {0, 1, 2 };
int  ia2[] = ia;                      //error

main()
{
     int  ia3[size ];
     ia3 = ia;                        // error
     return 0;
}
```

Dimensi array dapat lebih dari satu dan sering disebut sebagai array dimensi banyak, misalnya array dua dimensi seperti berikut:

```
int ia[4][3];                         // 2 dimensi
```

yang dapat juga diinisialisasi seperti contoh berikut:

```
int ia[2][3] = { {0, 1, 2} {3, 4, 5} };
```

atau

```
int ia[2][3] = {0, 1, 2, 3, 4, 5};
```

Perhatikan perbedaan antara dua deklarasi variabel array berikut:

```
Int ia[4][3] = { {0}, {3}, {6}, {9} };
```

dengan

```
int ib[4][3] = {0, 3, 6, 9};
```

yang dapat dituliskan sebagai catatan matrik berikut:

$$ia = \begin{pmatrix} 0 & 0 & 0 \\ 3 & 0 & 0 \\ 6 & 0 & 0 \\ 9 & 0 & 0 \end{pmatrix} \quad \text{dan} \quad ib = \begin{pmatrix} 0 & 3 & 6 \\ 9 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Indeks suatu array berdimensi banyak memerlukan pasangan tanda kurung […] untuk tiap dimensi seperti contoh berikut:

```
main()                                  // init array 2 dimensi
{
     const baris = 4;
     const kolom = 3;
     int ia[baris][kolom];
     for ( int  i = 0 ; i < baris ; ++ i )
          for ( int j = 0 ; j < kolom ; ++j )
               ia[i][j]  = i + j;
     return 0;
}
```

Array dan Pointer

Definisi dari sebuah array terdiri dari empat komponen, yaitu penentu tipe (*type identifier*), *identifier*, operator indeks ([]), dan nilai dimensi dalam operator [], misalnya char buf[8] = "abcdefg". Identifier suatu array merupakan pointer ke unsur pertama dari array itu, sehingga buf[0] sama artinya dengan *buf dan keduanya bernilai ‘a’. Dua cara berikut adalah sama yaitu untuk mengakses unsur kedua:

```
*(buf + 1);              // hasilnya = ‘b’
buf[1];                  // hasilnya = ’b‘  (cara ini lebih baik)
```

Nilai dari ekspresi buf juga sama dengan &buf[0]:

```
char  *pBuf  = buf ;
count  << *buf ++;       // error
count << *pBuf ++;       // OK
```

Hubungan array dan pointer ini dapat lebih jelas dilihat pada contoh program berikut:

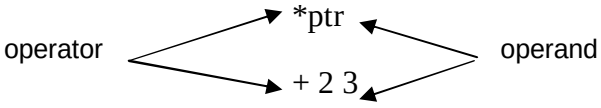
```
# include < ioetream .h >
const int ukuran = 4;
int main (void)
{
    int a[] [ukuran] = { {0, 1, 2, 3 } , {4, 5, 6 } , {7, 8, 9, 10 } };
    int *p = &a [0] [0];
    int *q = a [0];
    int *r = a [1] ;
    int *s = a [2] ;
    cout << *( p + ukuran +1 )  << endl;          // a[1] [1]
    cout << p[ukuran + 1] << endl;
    cout << p[2 * ukuran + 1 ] << endl;           // a[2] [1]
    cout << *(q + 2 * ukuran + 2 ) << endl;
    cout << *r << endl;                           // a[1] [0]
    cout << *(r – 2) << endl;                     // a[0] [2]
    return 0;
}
```

Keluaran dari program tersebut adalah:

```
5
5
8
9
4
2
10
```

EKSPRESI DAN PERNYATAAN (*STATEMENT*)

Ekspresi terdiri dari operan dan operator, dimana operator menyatakan operasi yang dilakukan. Evaluasi dari suatu ekspresi menghasilkan suatu nilai. Contoh suatu ekspresi sebagai berikut:



Operator yang bekerja hanya pada sebuah operan disebut operator **unary**, sedangkan operator yang bekerja pada dua operand disebut operator **binary**, misalnya:

```
unary:   * ptr
binary:  v1 + v2
```

Ekspresi yang terdiri dari dua atau lebih operator disebut ekspresi majemuk. Bentuk ekspresi yang paling sederhana adalah literal constant atau variabel, misalnya 3.14, “ipb“, var12, dan sebagainya. Hasil evaluasi dari 3.14 adalah 3.14 dengan tipe double, hasil evaluasi dari “ipb“ adalah alamat memori dari unsur pertama string itu dengan tipe char*, dan hasil evaluasi dari var12 adalah rvalue dengan tipe seperti yang ditentukan oleh definisinya.

Operator-operator Aritmatika

OPERATOR	OPERASI	PEMAKAIAN
*	Perkalian	expr * expr
/	Pembagian	expr / expr
%	Modulus	expr % expr
+	Pertambahan	expr + expr
-	Pengurangan	expr - expr

Operator-operator logika, relasional, dan kesamaan

Oprator-operator ini memberikan nila salah benar (1) atau salah (0), dengan daftar sebagai berikut:

OPERATOR	OPERASI	PEMAKAIAN
!	Logical NOT	! expr
<	Less than	expr < expr
<=	Less than or equal	expr <= expr
>	Greater than	expr > expr
>=	Greater than or equal	expr >= expr
==	Kesamaan	expr == expr
!=	Ketidaksamaan	expr != expr
&&	Logical AND	expr && expr
| |	Logical OR	expr || expr

Operator Penugasan

Operan kiri dari operator penugasan ("=") harus suatu lvalue, dan nilai dari operan kanannya disimpan di memori yang diasosiasikan dengan operan kirinya. Hasil dari suatu operator penugasan adalah nilai dari ekspresi yang di-*assign* ke operan kirinya, dan tipenya sesuai dengan tipe dari operan kirinya. Contoh:

```
int i, j;
i = j = 0;          // i, j masing-masing 0
                    // urutan evaluasi: kanan ke kiri
```

Bentuk khusus dari operator penugasan adalah ***compound assigment operator*** dengan bentuk umum:

```
a op= b
```

dimana op adalah operator yang dapat berupa: +, -, *, /, %, <<, >>, &, ^, dan ||. Bentuk pernyataan "a op = b" sebenarnya adalah singkatan dari "a = a op b". Misalnya a = a + 5 dapat ditulis sebagai a += 5.

Operator *Increment* dan *Decrement*

Operator *increament* ("++") dan *decrement* ("--") memberikan kemudahan notasi untuk menambah atau mengurang pada suatu variabel, dimana keduanya bersifat sebagai operator penugasan dan operannya harus berupa lvalue. Masing-masing operator mempunyai bentuk prefix dan postfix seperti pada contoh berikut:

```
main ( )
{
    int c = 10;
    ++c;                // increment  prefix
    c++ ;               // increment postfix
    --C;                // decrement prefix
    C++;                // decrement postfix
}
```

Bentuk prefix berarti bahwa nilai ekspresi adalah nilai variabel setelah diubah, sedangkan bentuk postfix berarti bahwa nilai ekspresi adalah nilai variabel sebelum diubah.

Operator SizeOf

Operator ini memberikan urutan (dalam byte) dari suatu ekspresi atau penentu tipe, dan bentuknya ada dua yaitu:

```
sizeof ( < type – specifier > );
sizeof  < ekspresi >;
```

Misalnya:

```
int ia [] = {1, 2, 3};
const sz = sizeof ia / sizeof (int);
```

Operator Koma

Suatu ekspansi koma terdiri dari sebarisan ekspresi-ekspresi yang dipisahkan dengan tanda koma, misalnya:

```
int i = 2 , ++i ;
```

Ekspansi dari ekspresi ini dievaluasi berturut-turut dari kiri ke kanan, dan hasilnya adalah nilai dari ekspresi yang paling kanan.

Operator Bitwise

Operator bitwise menginterpretasikan operan-operannya sebagai suatu kumpulan terurut dari bit, dimana tiap bit dapat mengandung nilai 0 dan 1. Operan-operan dari suatu operator bitwise harus bertipe integral. Berikut adalah daftar operator bitwise:

OPERATOR	FUNGSI	PEMAKAIAN
~	Bitwise – NOT	~ < ekspresi >
<<	Leftshift	<eksp1> << <eksp2>
>>	Rightshift	<eksp1> >> <eksp2>
&	Bitwise AND	<eksp1> & <eksp2>
^	Bitwise XOR	<eksp1> ^ <eksp2>
|	Bitwise OR	<elsp1> | <eksp2>

Contoh:

Unsigned char bits = 0027;

1	0	0	1	0	1	1	1

bit = ~ bits

0	1	1	0	1	0	0	0

unsigned char bits = 1 ;

0	0	0	0	0	0	0	1

bits = bits << 1;

0	0	0	0	0	0	1	0

bits = bits << 2;

0	0	0	0	1	0	0	0

bits = bits >>3;

0	0	0	0	0	0	0	1

unsigned char hasil;
unsigned char b1 = 0145;

0	1	1	0	0	1	0	1

unsigned char b2 = 0257;

1	0	1	0	1	1	1	1

hasil = b1 & b2;

0	0	1	0	0	1	0	1

hasil = b1 ^ b2;

1	1	0	0	1	0	1	0

hasil = b1 || b2;

1	1	1	0	1	1	1	1

FUNCTION NAME OVERLOADING

Yang dimaksud dengan *function name overloading* adalah penggunaan nama yang sama untuk beberapa fungsi yang menyatakan suatu operasi pada tipe-tipe argumen yang berbeda-beda. Secara abstrak operasi-operasi itu sebaiknya sama di antara fungsi-fungsi tersebut. Sebagai contoh, operator + dalam operasi aritmatika yang sudah tersedia sebagai bagian dari sistem. Pemakai dapat mendefinisikan fungsi yang di-overload menurut keperluannya, misalnya:

```
3 + 4 ;
3.2 + 5.6 ;
```

Kata (nama) yang di-overload harus ditentukan maksudnya berdasarkan konteks. Tanpa konteks, kata/nama yang di-overloaded itu akan *ambigous* (berarti banyak).

Dua atau lebih fungsi dapat diberi nama yang sama asalkan masing-masing fungsi mempunyai *signature* yang unik. Signature dibutuhkan oleh jumlah atau tipe dari argumen, misalnya:

```
int  max ( int , int ) ;
int  max ( const int *, int ) ;
int  max ( const list & ) ;
```

Masing-masing memerlukan implementasi yang berbeda, tetapi sebaiknya operasi umum yang dilakukan sama, yaitu memberikan nilai maksimum. Dari sudut pandang pemakai, hanya ada satu operasi, yaitu penentuan nilai maksimum, yaitu misalnya:

```
int  ix = max ( i , j );
int  iy = max ( iArray , 1024 );
```

Kompilator bertugas membedakan berbagai *instance* itu, bahkan pemakai. Tanpa overloading, ketiga fungsi maksimum di atas masing-masing harus mempunyai nama yang unik, misalnya i_max, ia_max, atau lainnya. Kerumitan leksikal inilah yang dapat diatasi dengan overloading. Apabila suatu nama fungsi dideklarasikan lebih dari satu kali dalam sebuah program, kompilator akan menginterpretasikan deklarasi-deklarasi yang kedua dan seterusnya sebagai berikut:

1. Jika return-type dan signature dari kedua fungsi itu tepat sama, yang kedua dianggap sebagai suatu redeklarasi dari yang pertama, misalnya:

```
extern void  print ( int *ia , int sz ) ;
void print (int *ary , int size ) ;              // redeklarasi
```

2. Jika signature dari kedua fungsi itu tepat sama tetapi mempunyai return-type berbeda, deklarasi yang kedua dianggap sebagai suatu redeklarasi yang keliru dan merupakan kesalahan pada waktu kompilasi, misalnya:

```
unsigned int max ( int * , int sz ) ;
extern int max ( int *ia , int ) ;        // error
```

3. Jika signature dari kedua fungsi itu berbeda, kedua fungsi itu dianggap overloaded, misalnya:

```
extern void print ( int * , int ) ;
void print ( double *da, int sz ) ;
```

Pada suatu *function overloaded* dipanggil, *instance* mana yang dipilih ditentukan melalui proses *argument matching*, yaitu membandingkan argumen aktual dengan argumen journal dari tiap instance. Misalnya diberikan empat instance berbeda dari fungsi print():

```
extern void print ( unsign int ) ;
extern void print  (char * ) ;
extern void print  (char ) ;
extern void print  ( int ) ;
```

Ada tiga kemungkinan matching dari suatu call, yaitu:

1. Match, suatu instance cocok, misalnya:

```
unsign int a ;
print ('a') ;          // cocok dengan print (char) ;
print ("a") ;          // cocok dengan print (char* );
print ( a ) ;          //  cocok dengan print ( unsign int ) ;
```

2. No Match, argumen aktual tidak dapat dibuat cocok dengan argumen journal dari semua instance yang tersedia, misalnya:

```
int *ip ;
small int  si ;
print (ip) ;           // error : no match
print (si) ;           // error  : no match
```

3. Ambigous Match, dimana argumen aktual dapat "match" lebih dari satu instance, misalnya:

```
unsign long  ul ;
print (ul) ;           // error : dapat match dengan print (int) ;
                       // print (unsign int ) , print (char ).
```

POINTER KE FUNGSI

Fungsi tidak dapat disalurkan sebagai argumen , tetapi pointer ke fungsi dapat. Tipe suatu fungsi ditentukan oleh return-type dan signature dari daftar argumennya. Nama fungsi bukan bagian dari tipe fungsi. Misalnya:

```
void  (*pf) (int * , int , int ) ;
```

menyatakan bahwa pf adalah pointer ke fungsi yang memerlukan tiga argumen bertipe int * , int , int dan mengembalikan nilai bertipe void. Contoh fungsi yang dapat ditunjuk oleh pf adalah:

```
void quickSort (int * , int , int) :
```

```
void heapSort (int * , int , int) ;
void ogah (int * , int , int) ;
```

Evaluasi dari nama fungsi memberikan pointer ke fungsi itu. Sebagai contoh, tipe dari hasil evaluasi quickSort adalah: void (*) (int * , int , int). Penggunaan operator **address-of** pada nama fungsi juga menghasilkan pointer ke tipe fungsi itu, sehingga quickSort ekivalen dengan &quickSort.

Pointer ke suatu fungsi dapat di inisialisasi sebagai berikut:

```
void (*pf) (int *, int , int ) = quickSort ;
void (*pfv2) (int * , int , int) = pfv ;
```

Perhatikan beberapa contoh berikut:

```
pfv = quickSort ;
pfv2 = pfv ;
extern void g (char ) ;
extern void g (unsign ) ;
void (*pg) (char ) = g ;            // ok : void g (char ).
void (*pg2) (int ) = g ;            // error
void (*pg3) (char ) = g ;           // error
int (*testFunction [10] ) ( ) ;

#include<isoterm.h>
extern min (int *, int ) ;
int (*pf) (int *, int ) = min ;
const int iaSize = 5 ;
int ia [iaSize] = {1,1,2,3,5} ;

main ( ) {
     count << “call langsung : “ << min (ia , iaSize) << endl ;
     count << “call lewat pointer : “ <<  pf (ia , iaSize) << endl ;
}

min (int *ia , int sz )  {
     int minVal = ia [0] ;
     for (int i = 1 ; i < sz ; ++i )
          if (minVal > ia [0] )   minVal = ia [0] ;
     return minVal ;
}
```

Pointer ke fungsi dapat menjadi argumen dari fungsi, misalnya:

```
extern void quickSort (int *, int , int ) ;
void sort (int , int , int , void ( *) (int *, int ,int , ) = quickSort ) ;
```

Aplikasi pointer ke fungsi dapat dilihat pada contoh program sort berikut:

```
#include<assert.h>
void sort ( int*ia , int bawah , int atas , void (*pf) (int *, int , int ) )
{
     assert (ia != 0 ) ;
     assert (pf != 0 ) ;
     pf ( ia , bawah , atas ) ;
}

typedef void (*pfi3) (int *, int ,int ) ;
void sort ( int * ia , int b , int a , pfi3 , pf ) ;
```

Pointer ke fungsi dapat menjadi return type di suatu fungsi, misalnya:

```
int ( *ff (int )) (int * ,int ) ;
```

yang mendeklarasikan bahwa ff() adalah suatu fungsi yang memerlukan suatu argumen bertipe int . Return type dari ff ( ) adalah int (*) (int *, int). Penulisan tersebut dapat disederhanakan dengan typedef, yaitu:

```
typedef int (*pfi) ( int*, int ) ;
pfi ff (int ) ;
```