**University of Amsterdam**

*Programming Research Group*

# A First Course on C Program Reading

Jan Bergstra
Inge Bethke
Marijke Loots

Report P0002 march 2000

University of Amsterdam

Department of Computer Science

Programming Research Group

# A First Course on C Program Reading

Jan Bergstra

Inge Bethke

Marijke Loots

Report P0002 march 2000

J.A. Bergstra

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7591
e-mail: janb@wins.uva.nl

Department of Philosophy
Utrecht University

P.O.Box 80126
NL-3508 TC Utrecht
The Netherlands

tel. +31 30 253 5579
e-mail: Jan.Bergstra@phil.uu.nl

I. Bethke

Programming Research Group
Department of Computer Science
University of Amsterdam

Kruislaan 403
NL-1098 SJ Amsterdam
The Netherlands

tel. +31 20 525 7583
e-mail: inge@wins.uva.nl

M.E. Loots

Tilburg University
Warandelaan 2
5037 AB Tilburg

e-mail: M.E.Loots@kub.nl

Universiteit van Amsterdam, 2000

# A First Course on C Program Reading

J.A. Bergstra[*†] Jan.Bergstra@phil.uu.nl

I. Bethke[†] inge@wins.uva.nl

M.E. Loots [†] mloots@wins.uva.nl

```
temp = malloc(sizeof(struct node));
printf("Enter a node:\n");
scanf("%i%f", &i, &f);
temp -> i = i;
temp -> f = f;
temp -> next = ll;
ll = temp;
```

Figure 1: Characteristic C notation

## Abstract

For a subset of the programming notation C a number of example programs is given together with some input-output pairs. The family of programs with input-output pairs constitutes an empirical semantics project report. It can be used to teach the computational intuition competence specific to C programming. This competence is analyzed in some detail before the examples are listed. The computational intuition competence can be developed and assessed without any use of computers, as a purely cognitive matter.

## Contents



*Utrecht University, Department of Philosophy, Applied Logic Group

†University of Amsterdam, Programming Research Group

# 1 Introduction

Teaching programming and programming languages may well be the largest single teaching effort worldwide after the teaching of some of the natural languages. The number of books supporting the teaching of programming languages, systems and methods is already enormous, and is growing very rapidly. Many books are soon forgotten in the midst of a very hectic scene. In previous papers [2, 3] we have noticed some salient features of modern programming teaching practice. This paper intends to contribute to the teaching practice from a novel perspective. The perspective has three fundamental ingredients. In various contributions, however, the various ingredients may feature in different degrees of prominence. The fundamental features are these:

- Explicit competence division. Various kinds of competences must be distinguished in an explicit way. For instance the competence to program in one language need not be the same as the competence to program in another language. Further the competence to read a program is different from the competence to write it. The competence to define what a program is differs from the competence to define what a compiler is and so on. Only by clearly distinguishing various kinds of competences can it be avoided that misplaced prejudices permeate the entire process of software teaching.
- Empirical semantics and computational intuition. The primary intuition about the meaning of a program is a cognitive matter. It results from taking notice of examples of program behavior. These examples are best generated in an empirical fashion, thus giving rise to empirical semantics.
- Conceptual analysis. Concepts from computer programming are hard to define. Nevertheless systematic efforts to provide conceptual definitions of important concepts in simplified models are necessary. We advocate program algebra, projection semantics and platform analysis as tools for conceptual clarification. Of course alternative options exist.

Regarding common viewpoints in the teaching of programming the following remarks can be made:

1. The concepts of a programming language and of a program are hardly ever clearly defined. The same holds for compilers, interpreters, assemblers, linkers and loaders and for partial evaluation, operating systems, processors and memory units.
2. The semantics of a program notation is almost always dealt with in an implicit fashion. In particular we have not seen examples where a discussion was included concerning the form of semantic awareness that was aimed at.

3. Most textbooks aim a teaching 'how to program'. Usually no single line is devoted to the definition of the concept of programming as a human (or even automated) activity. In addition hardly ever is there an explicit statement on what competence is meant with 'the ability to program'.

4. Quite remarkably in the vast majority of cases quality control is viewed as a separate topic. Programming is viewed as a competence that can exist in the absence of explicit awareness and methods for quality control. If quality control is discussed a choice is made for a specific QC methodology rather than that a survey of options is given beforehand.

5. The role of computers and 'practical work' in the teaching of programming is (in the vast majority of cases) taken for granted as a subject in no need of further discussion. In particular the student-computer interaction is often seen as necessary given the pragmatic objectives of a course. The mere possibility that it are exactly the pragmatic objectives at hand that are disturbed rather than enhanced by premature computer interaction is not even given a thought. There is no escaping the conclusion that computer interaction is usually regarded a virtue in its own right. Giving courses on programming without computer use is perceived as just as pointless as giving theoretical sports courses for people in desperate need of physical exercise.

6. Concept explanation is almost always made subordinate to some instrumental objective. A typical advice for students reads:

   > 'If you want to program with multiple threads you are well-off with some grasp of what a thread is. Precise definitions are left to the theorists who can better waste their own time than ours.'

   Conceptual analysis as an intellectual ambition for its own sake is looked at with great suspicion.

## 1.1 Software mechanics and empirical semantics

The term 'Software mechanics' has been coined in [4] to denote the study of the working of program notations. Software mechanics for a particular program notation can be studied in a variety of ways, empirical semantics being just one of those. Empirical semantics consists of the development of a large family of simple examples, illustrating key aspects of program execution. The examples serve no other purpose than to illustrate the meaning of programming features and feature combinations. The examples are not supposed to have a deeper meaning or intention. None of the examples is reusable for any other purpose than for experimentation with language features. Alternative routes are: (i) reading the language documentation, (ii) reading the program code of compilers and of run-time systems, (iii) getting acquainted with a multitude of applications.

## 1.2 On the role of C

C is very important for the simple reason that so many programs have been written in it. In addition it seems to have won complete industrial approval as a program notation consistent with extremely pragmatic ambitions. C can be used to do systems programming as well as application programming. C compilers are quite good. As a consequence whoever intends to produce new complex programs may be best off with either writing these in C directly or with writing them in a new language which subsequently is compiled into C. Java, PERL, Tcl/Tk are examples of such languages.

Some authors advocate C as the first program notation that has been developed by programmers for programmers. In any case it has been paradigmatic for computer programming since the mid sixties. It is no coincidence that modern languages like Java and $C^{++}$ feature a very significant inheritance from C. C has shaped the notational intuitions of an entire (and very large) generation of programmers.

We claim that knowledge of C is justified on cultural grounds if not on pragmatic grounds. The entire Internet technology has been developed with tools like C playing a very prominent role. Nowadays these developments in their turn give rise to further developments in program language design. Nevertheless, the technical role as well as the historical role of C make it entirely plausible to try and provide computing students with at least a passive knowledge of C. A passive knowledge is defined as the ability to read C programs and the ability to understand for what kind of application C is best used. This text is intended to be helpful for teaching the ability to recognize and understand programs written in C.

Acquiring an active knowledge of C, the ability to write (and maintain) programs, cannot be achieved by means of studying this text alone. Additional activities will be essential. Passive knowledge is an essential prerequisite for the acquisition of active knowledge.

**Acknowledgement**. The authors acknowledge a number of useful remarks concerning previous versions of this text made by Dimitri Hendriks and Piet Rodenburg.

## 1.3 Preliminary didactic aspects

Some general considerations regarding the objectives of this text as didactic material may be useful.

### 1.3.1 The target competence

We aim at establishing competence in predicting program behavior (of C programs) in limited and restricted circumstances based on a (program-notation-specific) computational intuition.

### 1.3.2 Competences left aside

This material is intended to be useful for an initial introduction to the program notation C. Several competences are not in the focus of attention. Most notably the ability to program is left out as a kind of competence both unanalyzed (as yet) and not within the focus of our current ambitions.

Another kind of competence entirely out of scope of the didactic ambitions of this project is the ability to determine whether a text is a syntactically correct C program. There are no negative examples bearing on syntax. Inductive learning regarding what is and what is not a correct C program cannot do without such negative examples. We claim that the computational intuition behind C can be communicated in a substantial way without ever discussing the syntactic limitations of the language.

Yet another competence left aside by our discussion is that of program quality control. Not even a rudimentary notion of specification will be used. All contents of the examples below are just about particular instances of computations. Whether the given programs do in fact implement some useful and more general specification in any general sense is left entirely open. Of course the very input-output pair realized by an example program may be used as its specification. In that case the run serves as a (complete !) black box test of the specification.

### 1.3.3 Didactic rationale

Teaching programming languages and systems is very much a chicken-and-egg issue. The student is bootstrapping through the subject, all the time substituting parts of the conceptual environment by more complicated parts. Following a course 'how to program' actively using a computer one encounters all conceptual topics simultaneously to such an extent that ignoring them is the only practical option.

The proposal made here is to leave significant questions like: how to program, what is program quality, what is a progamming environment, what is an operating system etc. to later stages, focussing on purely cognitive yet practice-related aspects. By explicitly restricting the teaching objectives one can avoid conceptual circularities to some extent. At the same time the human student will make some inferences that will prove to be of later use. The study of this document will establish a minimal programming ability, though not yet the ability to perform the program editing, storing, compiling running and testing, let alone the reporting of observed phenomena and the systematic search for relevant observations.

This piece of course material can, but need not, be used in combination with a more conceptual approach to programs and their meaning such as program algebra and projection semantics. A preliminary course on program algebra may precede this material. That will provide conceptual definitions for programs, program notations (is programming languages) and for semantics of program notations.

### 1.3.4 Didactic methods

Teaching aspects of the C program notation using this material can be done as follows: in oral courses the examples are discussed and a significant amount of additional (informal but convincing) information of program execution is provided in order to generate a mental picture on which a C-specific computational intuition can be based. This intuition must be so strong that it allows the student to make an educated guess on the outputs generated by marginally modified program examples. This guessing ability is a competence which can be tested during a pen-and-paper examination. Students produce program outputs (or guesses) only. Such an examination is best given in a setting where students have access to this document (and perhaps but not necessarily many other documents on C and its execution environment). No thorough semantic reasoning is required, tested, expected (or even appreciated).

### 1.3.5 Didactic limitations

Students may dislike our course (module) proposal for being superficial and weak on explanation. This critique can hardly be rejected as false. Nevertheless three remarks can be made:

(i) a principled approach to the syntax and semantics of C needs to be based on several preliminary courses. The precise contents of such courses are well defined regarding syntax, but far less unambigiously defined in the area of semantics. In semantics one faces a real bootstrapping problem. The semantics courses are incomprehensible for students who lack effective computational intuitions. It is like using a dictionary when teaching a baby to speak. People unable to speak will mostly not be able to read. People unable to read can hardly make effective use of a dictionary.

(ii) we claim that the computational intuition of C is a teachable subject. This is an assertion on learning psychology. It is claimed that students can be taught this ability. Examinations will prove this.

(iii) if a student explicitly refuses to be interviewed about questions that he/she cannot solve by unambiguous methods of reasoning and analysis that viewpoint has to be respected under all circumstances. In such a case a student will be asked to study different texts and to develop more 'intellectual' competences instead. This will require above-average skills, however. These skills in combination with a real inability to pass a test on C-specific informal computational intuition is hard to imagine.

# 2 Intelligent artificiality

The ability to determine the output that a program will produce when running on a given input is not characteristic of human beings. In fact executing C programs is prototypical of the superior abilities of computers in certain kinds of tasks. Intelligence represents the ability of a person to perform certain typical tasks.[1] Similarly 'artificiality' can be used to represent certain specific abilities of computers (and other artefacts).[2]

We propose that the student learns to perform some of these artificial tasks. Of course it must be expected that machines will be superior in tasks requiring artificiality compared to human beings. The objective is not to make man compete with machines but to provide him/her with better means to survive in a world which is dominated by artificiality. We propose 'intelligent artificiality' so to speak. The concept of 'intelligent artificiality' can be motivated on several grounds. for instance by the claim that by being able to perform artificial tasks an intelligent person will acquire significant capabilities to use the artefacts that are specific to these tasks. It may

[1] In particular tasks requiring intelligence. This introduces a non-trivial and hard-to-remove circularity.

[2] The ability to perform well without oxygen is specific to most computers and absent for persons. Still it will not require artificiality. Artificiality is connected with the execution of precisely given programs in a limited context and with very high performance.

also be considered necessary to constitute a mental model of the artefacts in question. We notice that in school teaching an enormous amount of intelligent artificiality is present nowadays. All tasks regarding calculations and elementary mathematics that are being taught to school children until the age of 12 can easily be performed by computers.

Currently tasks like walking, driving a bicycle, skating and 'fighting' are studied outside school. Interestingly these task are also far harder for computers to perform. This, however, may change quite soon.

## 3 Objectives of this course

This course is meant for people who either have no background in programming or else are willing to read a text as if they would not have such a background. Some familiarity with program algebra and projection semantics is considered helpful, but that is by no means a necessity. As much as possible the text has been made independent of a preliminary background in mathematics as well. For that reason there is no preliminary semantic introduction (in a logical style) to the concepts involved. In addition it is important that the material is studied without the use of computers. Using computers will require a background in computing and programming which is best assumed to be absent (at least hypothetically). It denies the very point of computing studies to assume that the practical use of computers can be done without preliminary conceptual reflection.

We expect that some 50 to 100 hours of time will be needed to understand this material in full detail. Lecturing may be helpful, some 6 hours of lecturing being sufficient to get most issues across. This still assumes that students will spend an additional 50 to 100 hours to study the text. Working group sessions with someone available and able to explain outputs and mechanisms may be quite useful. Some 6 hours of working groups ought to suffice to get people to the point that they can study the text themselves. The text has been made self-contained.

After this course a student must be able to read simple C programs and to predict their behavior correctly. A clear understanding of values and types, functions, assignments, side-effects, pointers, arrays and structures must emerge from reading this text.

In most cases it is useful to read the programs and their outputs first and to try to compute the same outputs by hand several days later. All programs can serve as exercises at the same time, an examination being just a listing of programs together with the question to predict the output generated by each program.

In order to obtain maximal clarity concerning the information content of programs and program packages the format of program families is used. A CPF (C Program Family) is a collection of named files containing C programs. Typically files containing a C program will have a name ending with extension `.c`. As a constructor for CPF's we will use `file:name.c(filecontents)`. Different named files can be combined with $\cup$ to form a larger CPF.

## 4 `Hello world` and so on

An introduction to C programming typically starts with a brief explanation of a program `Hello World`, a program that greets the console user on execution. This paper is no exception. As an example consider CPF1 = `file:hw.c(`

```
main()
{   printf("Hello World! ");
    printf("I am CompU!\n");
    printf("Bye\n");
}
```

). Compilation of the program is activated by running a C compiler (we used `gcc`) on file `hw.c`. This produces file `a.out`. That file name can be used as a command line instruction, its execution producing the following text on console:

```
Hello World! I am CompU!
Bye
```

It can be observed that `\n` generates a new line in the output. A similar program is CPF2 = `file:hr.c(`

```
main()
{   int i = 0;
    printf("Hello Reader!\n");
    printf("I print %d\n", i);
}
```

). Its output reads:

```
Hello Reader!
I print 0
```

The phrase `%d` in the argument of the procedure `printf()` has the following effect: the value of its second argument (`i` in this case) is printed in decimal notation. Individual characters can be printed as values of type `char`. Consider CPF3 = `file:h.c`(

```
main()
{   char i = 'H';
    char j = 'e';
    char k = 'l';
    char l = 'o';

    printf("%c%c%c%c%c\n",
           i,j,k,k,l);
}
```

). This program produces

```
Hello
```

Here we see that 5 consecutive arguments of type character are printed as characters (indicated by `%c`).

Next consider CPF4 = `file:yk.c`(

```
main()
{   float i = 2.1;
    float j = 4.2;
    printf("You know %f + %f = %f !\n",
           i,i,j);
}
```

). This produces

```
You know 2.100000 + 2.100000 = 4.200000 !
```

Here, `%f` specifies that floating-point numbers will be printed. The (default) precision with which the values are displayed is 6.

In addition to integers, characters and floating-point numbers there exist other data types in C. See Appendix C for a complete listing.

## 4.1 Informal projection semantics

Projection semantics is a style of program notation semantics where the meaning of a program is understood by means of a translation to another program with the same meaning. Although the second program may be longer and harder to read, its instructions will be easier to read, the task of semantic assignment thus having been simplified. The program of CPF4 uses two declarations and an involved print instruction. The following program in CPF4a has the same behavior but uses simpler, yet more, instructions. It may serve as a projection of CPF4 to a simpler notation. We will not be precise about general definitions of projections and about the precise form of programs that a projection can handle. At this level of discussion particular examples will be more helpful than general definitions. CPF4a = `file:yka.c`(

```
main()
{   float i,j;
    i = 2.1;
    j = 4.2;
    printf("You know ");
    printf("%f",i);
    printf("+");
    printf("%f",i);
    printf("=");
    printf("%f",j);
    printf(" !\n");
}
```

). This program has almost the same screen output as CPF4. Some spaces are missing, however:

```
You know 2.100000+2.100000=4.200000 !
```

Different `printf()` instructions can be combined in various ways: a `\t` (contained in a string) moves the cursor to the next tab stop; `\a` is the alert escape which rings the system bell. CPF4b = `file:ykb.c`(

```
main()
{   float i, j;
    i = 2.1;
    j = 4.2;
    printf("You know \t");
```

```
        printf("%.1f \n",i);
        printf("\t \t%.1f \n",i);
        printf("\t \t---\n");
        printf("\t \t%.1f !\n\a",j);
    }
```

) shows the output:

```
You know        2.1
                2.1
                ---
                4.2 !
```

Note that the conversion specifier `%.1f` causes the value of floating-point numbers to be displayed with 1 decimal digit to the right of the decimal point. For more detailed information on formatted I/O see Appendix D.)

## 4.2 Style

Some programs may only work well in our experiments; in other words they are not in accordance with a programming style aiming at robust software (minimally dependent on the platform). This can imprint useless program patterns in the reader's mind. To remedy this a reader must be made aware of code fragments that are most likely not portable to other platforms.

Below we suggest two stylistic constraints advocated by the literature:

1. a program containing I/O instructions is best preceded with the line `#include <stdio.h>`
2. the `main` function is best ended with the line `return 0;`.

I/O commands should appear only after the line `#include <stdio.h>`, which asks that some definitions having to do with the "Standard I/O Library" be included in the program (see also Appendix E). Moreover, the body of the `main()` function should end with the `return 0;` statement which tells the operating system (or whatever invoked the program that `main` is the main function of) whether it succeeded or not; by convention, a return value of 0 indicates success. As an example consider CPF4c = `file:ykc.c`(

```
#include <stdio.h>

main()
{   float i, j;
    i = 2.1;
    j = 4.2;
    printf("You know %f + %f = %f !\n",
           i,i,j);
    return 0;
}
```

). This CPF produces once more:

```
You know 2.100000 + 2.100000 = 4.200000 !
```

## 4.3 Informal coprogram tracing

In order to understand the execution of the program of CPF4 it is important to have a mental model of the state of the machine during program execution. We will use the term coprogram for the memory of a machine during computation. That coprogram will move through a series of successive states, each next state found as the result of performing a single step in the execution of the program.

The idea of a coprogram is to keep a data base of information during program execution. Every instruction of the program is executed by means of an update on the data base. Tests will be performed on the data base. We will be very informal about coprograms. In particular we will not explain how coprograms in general are transformed by instructions. Rather we will illustrate the states a coprogram may visit during the execution of a program. The effect of the individual program instructions is to change the state of the coprogram to the next state. The more low-level the program is (elementary control structures over a simple set of actions) the clearer the picture of coprograms works out. In particular it combines well with projection semantics. After projection it may be helpful to view a program execution as a sequence of actions, each enacting a state transformation. The version of a coprogram used when attempting to understand a C program depends on the complexity of the machine model needed to run the program.

We will use the phrase (informal) coprogram tracing for the (informal) specification of a sequence (trace) of memory states that may be visited during the run of a computation. In software technology coprogram tracing is the task of so-called debuggers (named after the phase in software design where coprogram tracing has proven most helpful: debugging[3]. In the case of CPF4 it is sufficient to consider coprograms with states containing a list of variables that have been introduced already during program execution together with their current value. Coprogram tracing helps to understand the machine states during program execution. Obviously the machine can be modeled at many levels of abstraction. The more abstract the model, the less (redundant) detail has to be taken into account. In principle one's objective is to provide models at the highest level of abstraction, within the constraint that it still is informative.

In the case of CPF4 the coprogram must keep track of the names that have been introduced and of the values that these names stand for. At this level of abstraction it is immaterial that in the C execution model, the names are connected with memory addresses and that these addresses contain values. The reason that this is immaterial is simple: we need not take into account the possibility that different names are connected with the same address; this is guaranteed not to occur by the implementation of name introduction in C.

In the state descriptions below a pair $i : n$ stands for the existence of a name $i$ which denotes value $n$. The following trace of states is useful for an understanding of the execution of CPF4a:

```
{},
{i:rv},
{i:rv, j:rv},
{i:2.1, j:rv},
{i:2.1, j:4.2},
{i:2.1, j:4.2},
{i:2.1, j:4.2},
...,
{i:2.1, j:4.2},
```

the latter states being visited (but not modified) during the sequence of `printf()` instructions. Here `rv` represents some 'random value' (e.g. 20056) which is determined run-time by the operating system, depending on the state of the processor.

### 4.4 Greeting the user

The next program asks the user to enter his/her/its name. This name is used in a reply message. The routine `scanf()` used to get text from the console requires the library `stdio.h`. The routine `scanf()` will read a string from the console until a space is found. This string can be assigned to a variable for a character array (here of length 25). Let CPF5 = `file:hu.c(`

```
#include <stdio.h>

main()
{   char uName [25];
    printf("Hello User! ");
    printf("Who are you?\n");
    printf("I am: ");
    scanf("%s", uName);
    printf("Welcome %s\n",uName);
    printf("Bye");
    return 0;
}
```

). The interactive output is (for instance):

```
Hello User! Who are you?
I am: Jan
Welcome Jan
Bye(prompt)
```

The first `'Jan'` is typed by the user, the second one is an echo by the program. Since a next line char (`\n`) is omitted after the final `Bye`, the latter is shown on the screen before the prompt[4].

## 5 Assignments

Assignments are at the heart of most imperative programming languages. Though assignments as such

[3] Debugging is a form of real time white box testing aimed at the quick identification of problem hotspots in a program with on-the-fly repair possibilities.

[4] This behavior may be different on other platforms, though.

are not essential for the concept of imperative programming, one may say that imperative programming is the style of programming that has been developed in order to exploit all forms of assignments in a most effective way.

## 5.1 A first excursion

An assignment statement requests that a value be assigned to a variable. Such a statement normally takes the syntactic form of a variable, followed by =, the symbol that represents the assignment operation, and then by an expression indicating the value to be assigned. The semantics of such a statement is that the expression is to be evaluated and the result assigned as the value of the variable.

In the following examples we will consider programs introducing a number of variables and applying successive assignments to these.

### 5.1.1 Evaluation of expressions

Assignments always involve the evaluation of an expression. That, however, can be visualized without the use of assignments: Let CPF6 = `file:eval0.c(`

```
#include <stdio.h>

main()
{   int i = 0;
    int j = 1;
    int k = 2;
    int l = 3;
    int m = 4;
    printf("%d ",i+j+k);
    printf("%d ",(i-j)*k);
    printf("%d ",m-7);
    printf("%d ",m-7*i-j*k);
    printf("%d ",(m-7)*(i-j)*k);
    printf("%d\n",(m-7)*(i-j)*k+(m-k)*l);
    return 0;
}
```

). This program produces:

```
3 -2 -3 2 6 12
```

### 5.1.2 Expressions assigned to variables

`printf` can use variables and lead to the same result. For that purpose variables are effective: Let CPF7 = `file:eval.c(`

```
#include <stdio.h>

main()
{   int i = 0;
    int j = 1;
    int k = 2;
    int l = 3;
    int m = 4;
    int r;
    r = i+j+k;
    printf("%d ",r);
    r = (i-j)*k;
    printf("%d ",r);
    r =  m-7;
    printf("%d ",r);
    r = m-7*i-j*k;
    printf("%d ",r);
    r = (m-7)*(i-j)*k;
    printf("%d ",r);
    r = (m-7)*(i-j)*k+(m-k)*l;
    printf("%d\n",r);
    return 0;
}
```

). Also this program produces:

```
3 -2 -3 2 6 12
```

However, reordering the instructions leads to different outputs: consider CPF7a = `file:evala.c(`

```
#include <stdio.h>

main()
{   int i = 0;
    int j = 1;
    int k = 2;
    int l = 3;
    int m = 4;
    int r;
    r = (m-7)*(i-j)*k+(m-k)*l;
    printf("%d ",r);
```

```
        r = (m-7)*(i-j)*k;
        printf("%d ",r);
        r = m-7*i-j*k;
        printf("%d ",r);
        r =  m-7;
        printf("%d ",r);
        r = (i-j)*k;
        printf("%d ",r);
        r = i+j+k;
        printf("%d\n",r);
        return 0;
    }
```

). This program produces

```
12 6 2 -3 -2 3
```

### 5.1.3 Accumulating assignments

Program execution in the following examples is no more than a sequence of assignments followed by a number of print instructions. The coprogram states involved are associations of values with all variables. The execution of an assignment $x = E$, with $E$ some expression, works as follows: evaluate the value of $E$, let $d$ be the result, and make a new association between $x$ and $d$. Let CPF8 = `file:asgm.c`(

```
#include <stdio.h>

main()
{   int i = 0;
    int j = 1;
    int k = 2;
    int l = 3;
    printf("%d ",i);
    j = j+1;
    printf("%d ",j);
    k = k+2;
    printf("%d ",k);
    l = l+4;
    printf("%d ",l);
    printf("pause ");
    printf("%d ",j+k);
    l = j+k;
    printf("%d ",l);
    printf("%d ",l+l);
    l = l+l;
    printf("%d\n",l);
    return 0;
}
```

). Its output on screen reads:

```
0 2 4 7 pause 6 6 12 12
```

Let CPF8a = `file:asgma.c`(

```
#include <stdio.h>

main()
{   int i = 0;
    int j = 1;
    int k = 2;
    int l = 3;
    printf("%d ",i);
    j = j+1;
    printf("%d ",j);
    k = (k+2)*1+0;
    printf("%d ",k);
    l = 0*(-l+4) - 5;
    printf("%d ",l);
    l = (j-k)/l+l;
    printf("%d ",l);
    l = l+2*l*l;
    printf(" %d\n",l);
    i = 0;
    j = 1;
    k = 2;
    l = 3;
    printf("%d ",i);
    j = j+1;
    printf("%d ",j);
    k = k+2*1+0;
    printf("%d ",k);
    l = 0*-l+4 - 5;
    printf("%d ",l);
    l = j-k/l+l;
    printf(" %d ",l);
    l = (l+2)*l*l;
    printf("%d\n",l);
    return 0;
}
```

). This program features more involved assignments. Its output on screen reads:

```
0 2 4 -5 -5  45
0 2 4 -1  5 175
```

As usual, multiplication and division have higher precedence than addition and subtraction, and whenever the default precedence or associativity does not give the intended meaning, explicit parentheses can be used. Moreover, when applied to integers, the division operator discards any remainder, so e.g. -2/-5 is 0. But when either operand is a floating-point quantity, the division operator yields a floating-point result, with a potentially nonzero fractional part (see also Appendix C). Thus CPF8c = `file:asgm3.c`(

```
#include <stdio.h>

main()
{   int i = 10;
    int j = 3;
    float k = 10;
    float l = 3;
    printf("%d\n",i/j);
    printf("%.1f\n",i/l);
    printf("%.2f\n",k/j);
    printf("%.3f\n",k/l);
    return 0;
}
```

) produces the following output:

```
3
3.3
3.33
3.333
```

Apart from arithmetical operators, C also has relational, equational and logical operators. In every category and between the several categories there are rules of precedence and associativity. See Table 6 for a complete listing together with their associativity and precedence properties.

### 5.2 Side-effects 1

Side-effects arise if an assignment cannot simply be understood as: (i) evaluate an expression, (ii) obtain a value, (iii) let the left-hand side variable be a name for that value. C contains some notorious features generating side-effects. Understanding these is worth some time and effort, even if one decides never to use them in one's own career as a responsible and tasteful programmer.

The increment features, written `i++` and `++i` allow one to use an int variable `i` and to increment it by 1 after use (`i++`), or to increment it before use (`++i`). The decrement features are similar. Increments and decrements can also be applied to variables of type `float` and `char`. In the latter case, the ASCII code (see Table 2) is incremented or decremented. First we will consider for each type a single variable, and apply only increments and decrements. The important feature of these programs is that print instructions using an expression influence the value of the variables occurring in the expression.

Let CPF9 = `file:se.c`(

```
#include <stdio.h>

main()
{   int i = 0;
    float f = 0.123;
    char c = 'A';
    printf(" 1\t %2d\t %6.3f\t %c\n",
                                    i,f,c);
    printf(" 2\t %2d\t %6.3f\t %c\n",
                              i++,f++,c++);
    printf(" 3\t %2d\t %6.3f\t %c\n",
                                    i,f,c);
    printf(" 4\t %2d\t %6.3f\t %c\n",
                              ++i,++f,++c);
    printf(" 5\t %2d\t %6.3f\t %c\n",
                                    i,f,c);
    printf(" 6\t %2d\t %6.3f\t %c\n",
                              i--,f--,c--);
    printf(" 7\t %2d\t %6.3f\t %c\n",
                              --i,--f,--c);
    printf(" 8\t %2d\t %6.3f\t %c\n",
                              --i,--f,--c);
    printf(" 9\t %2d\t %6.3f\t %c\n",
                              --i,--f,--c);
    printf("10\t %2d\t %6.3f\t %c\n",
                              ++i,++f,++c);
    printf("11\t %2d\t %6.3f\t %c\n",
                              ++i,++f,++c);
```

```
        printf("12\t %2d\t %6.3f\t %c\n",
                              ++i,++f,++c);
        printf("13\t %2d\t %6.3f\t %c\n",
                              ++i,++f,++c);
        return 0;
    }
```

). Its output on screen reads:

```
 1         0       0.123  A
 2         0       0.123  A
 3         1       1.123  B
 4         2       2.123  C
 5         2       2.123  C
 6         2       2.123  C
 7         0       0.123  A
 8        -1      -0.877  @
 9        -2      -1.877  ?
10        -1      -0.877  @
11         0       0.123  A
12         1       1.123  B
13         2       2.123  C
```

Next we use multiple occurrences of side-effect generating constructs. Let CPF9a = `file:sea.c`(

```
    #include <stdio.h>

    main()
    {   int i = 0;
        int j = 0;
        int k = 0;
        int l = 0;

        printf(" 1 %d\n",i);
        printf(" 1 %d\n",j);
        printf(" 1 %d\n",k);
        printf(" 1 %d\n",l);

        printf(" 2 %d\n",3*(i++));
        printf(" 2 %d\n",3*j++);
        printf(" 2 %d\n",3*(++k));
        printf(" 2 %d\n",3*++l);

        printf(" 3 %d\n",++i + ++i);
        printf(" 3 %d\n",i-- + i--);
        printf(" 3 %d\n",i-- + i--);
        printf(" 3 %d\n",i-- + i--);

        printf(" 4 %d\n",i-- - i--);
        printf(" 4 %d\n",++i - ++i);
        printf(" 4 %d\n",++i - ++i);
        printf(" 4 %d\n",++i + ++i);
        return 0;
    }
```

). As an experiment we compile the program with the default Sun compiler `cc` (left) and the GNU C compiler `gcc` (right). The respective outputs read:

```
1 0        1 0
1 0        1 0
1 0        1 0
1 0        1 0
2 0        2 0
2 0        2 0
2 3        2 3
2 3        2 3
3 6        3 5
3 6        3 6
3 2        3 4
3 -2       3 2
4 0        4 0
4 0        4 -1
4 0        4 -1
4 2        4 9
```

As can be observed, the compilers behave quite differently from block 3 onwards. Thus at this point, little can be said in general about the value of expressions. In particular the observations made above need not generalize to other platforms and compilers.

### 5.2.1 Side-effect interference

The following quotation, taken from [15], is instructive concerning multiple side-effects.

> Although the postincrement and postdecrement operators ++ and −− perform their operations after yielding the former value, the implication of "after" is often misunderstood. It is not guaranteed that an increment or decrement is performed immediately after giving up the previous value and

> before any other part of the expression is evaluated. It is merely guaranteed that the update will be performed sometime before the expression is considered "finished" (before the next "sequence point," in ANSI C's terminology)...
>
> The behavior of code which contains multiple, ambiguous side-effects has always been undefined. (Loosely speaking, by "multiple, ambiguous side-effects" we mean any combination of ++, − −, =, +=, − =, etc. in a single expression which causes the same object either to be modified twice or modified and then inspected. This is a rough definition.) Don't even try to find out how your compiler implements such things (contrary to the ill-advised exercises in many C textbooks); as K&R wisely point out, "if you don't know how they are done on various machines, that innocence may help to protect you."

The advice from [6], page 80, is:

> ... the programmer should avoid using statements with increment or decrement operators in which a particular variable being incremented or decremented appears more than once.

In [7] we find the following suggestion;

> To prevent side-effect bugs, follow this rule: If you use a side-effect operator in an expression, do not use the affected variable anywhere else in the expression.

These authors present the expression `i * i++` as an example violating their rule and one for which the evaluation result may vary from compiler to compiler (or worse). A program containing this expression is at risk of being not portable (which defeats some of the objectives of the design of C).

We will assume that expressions are always evaluated in accordance with some rules. Even if these rules are not followed, this observation can be used to reformalize the hypothetical rules.

We will also consider expressions like `m+ ++m` to be outside the standardized area. This expression may lead to confusion regarding the ordering of a pre-increment to a variable and the use of that variable elsewhere in the expression.

Below we will produce a number of experiments with expressions featuring multiple side-effects demonstrating that the situation is quite complicated.

### 5.2.2 Multiple increments and decrements

Let CPF9b = `file:seb.c(`

```
#include <stdio.h>

main()
{   int j = 0, k = 0, m = 0;

    printf(" 1 %d\n",j);
    printf(" 1 %d\n",k);
    printf(" 1 %d\n",m);

    printf(" 2 %d\n",m--);
    printf(" 2 %d\n",m);

    m = 0;
    printf(" 3 %d\n",--m);
    printf(" 3 %d\n",m);

    m = 0;
    printf(" 4 %d\n",m-m--);
    printf(" 4 %d\n",m);

    m = 0;
    printf(" 5 %d\n",m-m++);
    printf(" 5 %d\n",m);

    m = 0;
    printf(" 6 %d\n",m+m--);
    printf(" 6 %d\n",m);

    m = 0;
    printf(" 7 %d\n",m+m++);
    printf(" 7 %d\n",m);
```

```
        m = 0;
        printf(" 8 %d\n",j--+m+m++);
        printf(" 8 %d\n",m);
        printf(" 8 %d\n",j);

        m = 0;
        printf(" 9 %d\n",k+++m+m++);
        printf(" 9 %d\n",m);
        printf(" 9 %d\n",k);

        j = m = 0;
        printf("10 %d\n",j+j--+m+m++);
        printf("10 %d\n",m);
        printf("10 %d\n",j);

        k = m = 0;
        printf("11 %d\n",k+k+++m+m++);
        printf("11 %d\n",m);
        printf("11 %d\n",k);

        m = 0;
        printf("12 %d\n",m-m--+j+++(++j));
        printf("12 %d\n",m);
        printf("12 %d\n",j);

        m = 0;
        printf("13 %d\n",m-m--+m+++(++m));
        printf("13 %d\n",m);

        return 0;
    }
```

).

The outputs for the Sun (left) and the GNU compiler (right) are as follows:

```
 1 0         1 0
 1 0         1 0
 1 0         1 0
 2 0         2 0
 2 -1        2 -1
 3 -1        3 -1
 3 -1        3 -1
 4 0         4 0
 4 -1        4 -1
 5 0         5 0
 5 1         5 1
 6 0         6 0
 6 -1        6 -1
 7 0         7 0
 7 1         7 1
 8 0         8 0
 8 1         8 1
 8 -1        8 -1
 9 0         9 0
 9 1         9 1
 9 1         9 1
10 0        10 0
10 1        10 1
10 -1       10 -1
11 0        11 0
11 1        11 1
11 1        11 1
12 0        12 -1
12 -1       12 -1
12 1        12 0
13 2        13 1
13 1        13 -1
```

Concerning the Sun compiler, the pattern emerging from these observations is as follows: given an expression to be evaluated the left-most subexpression equipped with an preincrement or decrement is evaluated first. Thereafter evaluation takes place from left to right. Finally, any occurring postincrements or decrements are evaluated. In contrast, the GNU compiler seems to evaluate expressions from left to right ignoring preincrement or decrement occurrences which have not yet been encountered. The final result in block 13, however, comes as a surprise.

We leave this matter at this point. The reader will have acquired some feeling of the meaning of decrements, sufficient to cover the practical uses he may expect to encounter.

## 5.3 Side-effects 2

An assignment is itself an expression, evaluating to the result of its left-hand side after evaluation of the assignment. Again one should not expect C compilers to produce standardised output in the case multiple side-effect producing subexpressions exist in one expression. And even if a pattern emerged, it would not reflect facts of a useful generality.

We conclude that except in very simple circumstances naive users should refrain from the interpretation (i.e. attempts to understand) of accumulating side-effects. Let CPF10 = `file:sec.c`(

```
#include <stdio.h>

main()
{   int i = 0, k = 0, l = 0;

    i = i + (k = l + 2);
    printf(" 1 %d\n",i);
    printf(" 1 %d\n",k);

    i = 0;
    k = 0;
    i = (2 - i) + (k = i + 2)
        + (l = i + 1);
    printf(" 2 %d\n",i);

    i = 0;
    k = 0;
    printf(" 3 %d\n",
        i + (i = k + 2)
    );
    printf(" 3 %d\n",i);

    i = 0;
    printf(" 4 %d\n",
        (i = i) + (i = i + 2)
    );
    printf(" 4 %d\n",i);

    i = 0;
    printf(" 5 %d\n",
        (i = 3) + (i = i + 2));
    printf(" 5 %d\n",i);

    i = 0;
    printf(" 6 %d\n",
        i + (i = i + (i = 2))
    );
    printf(" 6 %d\n",i);

    i = 0;
    printf(" 6 %d\n",
        i + (i = i +
                (i = i + (i = 2))
            )
    );
    return 0;
}
```

). The corresponding output is:

```
1 2
1 2
2 5
3 4
3 2
4 2
4 2
5 8
5 5
6 8
6 4
6 16
```

## 5.4 Initialization

Some examples illustrate the effect of initializations. In the absence of initialization (pseudo) random values appear. They have to do with the accidental contents of memory locations. Let CPF11 = `file:ini.c`(

```
#include <stdio.h>

main()
{   int i = 0, j, k, l;

    printf(" 1 %d\n",i);
    printf(" 2 %d\n",j);
    printf(" 3 %d\n",k);
    printf(" 4 %d\n",l);

    return 0;
}
```

). The output we found was:

```
1 0
2 4
```

```
3 -268436932
4 3
```

Running this program another time will probably produce other outputs (except the first 0 of course).

### 5.5 Side-effects 3

Assignments can occur in the following interesting form: `x += e` abbreviating x = x + e and `x *= e` abbreviating x = x * e. In general, `x op= e`, where `op` is a binary operator, is short-hand for `x = x op e`. Such assignments may occur as subexpressions of larger expressions. The construction is sometimes termed a compound assignment. Compound assignments introduce side-effects like the increments, decrements and ordinary assignment expressions. Given the observations in previous subsections, we refrain from using compound assignments of the form `x op= e[++x]` where `e[++x]` is some expression containing `++x`. The same restriction applies to predecrement, postincrement and decrement. Let CPF12 = `file:sed.c.c`(

```
#include <stdio.h>

main()
{   int i = 0;
    int j = 0;
    int k = 0;
    float l = 12;

    i += 3;
    j += 2 + i + j;
    k *= 5;
    l /= j;

    printf(" 1 %d\n",i);
    printf(" 2 %d\n",j);
    printf(" 3 %d\n",k);
    printf(" 4 %f\n",l);

    j += k * (--i);
    k += 2 * (k -= j);
    l += 5 + (l = 4);

    printf(" 6 %d\n",j);
    printf(" 7 %d\n",k);
    printf(" 8 %f\n",l);

    return 0;
}
```

). This program produces:

```
1 3
2 5
3 0
4 2.400000
6 5
7 -15
8 13.000000
```

## 6 Conditionals

Statements are the "steps" of a program. By default, statements are executed sequentially, one after another. Various C control structures, however, enable the programmer to digress from the strictly sequential order of execution. C provides three types of such control structures:

1. the `if` selection structure (single selection) executes a block of statements if a certain condition is true and skips that block if the condition is false;

2. the `if/else` selection structure (double selection) executes a block of statements if a certain condition is true and executes a different block if the condition is false;

3. the `switch` selection structure executes one of many different blocks depending on the value of a certain expression.

In C, the Boolean values 'true' and 'false' are represented as integers. 'false' is represented by the value 0, and 'true' is represented by any value that is nonzero. Although C accepts any nonzero value as 'true', it sets a true value to 1. The logical operators that may be used to form complex conditions are `&&` (conjunction), `||` (disjunction), and `!` (negation).

Let CPF13 = `file:sel.c`(

```
#include <stdio.h>

main()
{   int p = 0, q = 3;

    if ( !p && q ) {
       printf("1 %d\n", p);
       printf("1 %d\n", q);
       printf("1 %d\n", p||q);
       p++;
    }
    if ( !(p || q) ) {
       printf("skip this block");
    } else {
       printf("2 %d\n", p);
       printf("2 %d\n", q);
       printf("2 %d\n", !(p || q));
       q -= 5;
    }
    switch ( !(p && !q) ) {
       case 1 :
          printf("3 %d\n", p);
          printf("3 %d\n", q);
          printf("3 %d\n", !q);
          break;
       case 0:
          printf("skip this block");
          break;
    }
    return 0;
}
```

). The output reads

```
1 0
1 3
1 1
2 1
2 3
2 0
3 1
3 -2
3 0
```

The `break` statement causes program control to continue with the first statement after the `switch`. If `break` is not used anywhere in a `switch` structure, then each time a match occurs in the structure, the statements for all the remaining `case`s will be executed. This feature, however, is rarely useful. A `switch` can also contain a `default` case which is executed if no match occurs.

It is possible to nest a selection structure inside another. Moreover, the relational operators `<` (less than), `<=` (less than or equal), `>` (greater), `>=` (greater than), `==` (equal, *not* a single =), and `!=` (not equal) can be used in a condition, since an expression of the form `e1 [rop] e2`, where `[rop]` is one of these operators, is given the value 1 or 0 depending on whether the relation is true or false.

Let CPF13a = `file:sela.c`(

```
#include <stdio.h>

main()
{   int i, j;

    printf("Enter two integers: ");
    scanf("%d%d", &i, &j);

    switch (i) {
       case 12 :
          if ( j >= 23 ) {
             if ( i+j != 50 ) {
                printf("i=12\n");
                printf("j>=23\n");
                printf("i+j!=50\n");
             } else {
                printf("i=12\n");
                printf("j>=23\n");
                printf("i+j=50\n");
             }
          } else {
             printf("i=12 j<23\n");
          }
          break;
       case -3 :
          if ( j == 0 ) {
              printf("i=-3 j=0\n");
          } else {
              printf("i=-3 j!=0\n");
          }
          break;
```

```
            default:
/* WRONG */       if ( j = 1 ) {
                  printf("Default case: ");
                  printf("j=1\n");
               }
         }
         return 0;
    }
```

). Below we list 7 sessions covering all possible cases.

```
Enter two integers: 12 23
i=12
j>=23
i+j!=50

Enter two integers: 12 38
i=12
j>=23
i+j=50

Enter two integers: 12 -4
i=12 j<23

Enter two integers: -3 0
i=-3 j=0

Enter two integers: -3 7
i=-3 j!=0

Enter two integers: 100 1
Default case: j=1

Enter two integers: 100 10
Default case: j=1
```

Note that in the default case we accidentally wrote `j = 1` using a single `=`. As a consequence, the program does not test whether j=1, as we intended. Instead, it assigns 1 to j, which means that j is true! Therefore the body of the single selection statement will always be carried out irrespective of the value of j. Using a single `=` where a `==` was intended is a common programming error which probably everybody makes at some point. We marked the erroneous line of code with a comment which begins with a `/*` and ends with a `*/`. Comments do not cause the computer to perform any action when the program is executed. Comments are ignored by every C compiler and do not cause any machine language object code to be generated.

Equivalent programs—testing equality where intended—are CPF13b = `file:selb.c`(

```
#include <stdio.h>

main()
{   int i, j;

    printf("Enter two integers: ");
    scanf("%d%d", &i, &j);

    if ( i == 12 && j >= 23 && i+j != 50) {
       printf("i=12\n");
       printf("j>=23\n");
       printf("i+j!=50\n");
    }
    if ( i == 12 && j >= 23 && i+j == 50) {
       printf("i=12\n");
       printf("j>=23\n");
       printf("i+j=50\n");
    }
    if ( i == 12 && j < 23 ) {
       printf("i=12 j<23\n");
    }
    if ( i == -3 && j == 0) {
       printf("i=-3 j=0\n");
    }
    if ( i == -3 && j != 0 ) {
       printf("i=-3 j!=0\n");
    }
    if ( i != 12 && i!=-3 && j == 1) {
       printf("Default case: j=1\n");
    }
    return 0;
}
```

), which uses merely single selection statements, and CPF13c = `file:selc.c`(

```
#include <stdio.h>

main()
{   int i, j;
```

```
    printf("Enter two integers: ");
    scanf("%d%d", &i, &j);

    if ( i == 12 ) {
       if ( j >= 23 ) {
          if ( i+j != 50) {
             printf("i=12\n");
             printf("j>=23\n");
             printf("i+j!=50\n");
          } else {
             printf("i=12\n");
             printf("j>=23\n");
             printf("i+j=50\n");
          }
       } else {
          printf("i=12 j<23\n");
       }
    } else {
       if ( i == -3 ) {
          if ( j == 0) {
             printf("i=-3 j=0\n");
          } else {
             printf("i=-3 j!=0\n");
          }
       } else {
          if ( j == 1) {
             printf("Default case:
                            j=1\n");
          }
       }
    }
    return 0;
}
```

), which is a nested `if/else/if/else/...` chain.

# 7 Repetitions

Many programs involve repetition or looping. A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true. In this section, we shall discuss 3 means of repetition: the `for` structure, the `while` structure, and the `do/while` structure.

## 7.1 The `for` repetition structure

Loops generally consist of two parts: one or more control expressions that control the execution of the loop, and the body, which is the statement or block of statements that is executed over and over. The syntax of a `for` loop is

```
for (expr1; expr2; expr3) {
    block of statements
}
```

where `expr1; expr2; expr3` are three control expressions.

Many loops are set up to cause some variable to step through a range of values, or, more generally, to set up an initial condition and then modify some value to perform each succeeding loop as long as some condition is true. The three expressions in a `for` loop encapsulate these conditions: `expr1` sets up the initial condition, `expr2` tests whether another run through the loop should be taken, and `expr3` performs 'updates' after each run through the loop and prior to the next one.

Let CPF14 = `file:for.c`(

```
#include <stdio.h>

main()
{   int i;

    for (i = 1; i <= 5; i++) {
       printf("*****\n");
    }
    return 0;
}
```

). Its output reads

```
*****
*****
*****
*****
*****
```

In this first example, we have `i = 1` as `expr1`, `i <= 5` as `expr2`, `i++` as `expr3`, and the call to `printf` as the only statement in the body of the loop. So the loop begins by assigning 1 to `i`, proceeds as long as `i` does not exceed 5, prints out `*****` during each run through the loop, and increments `i` between each run through the loop.

As in the case of conditionals, `for` structures can be nested. In fact, the body of a `for` loop can contain every well-formed C construct such as conditionals, other loop structures etc.; likewise, loop structures may occur inside every selection construct. Below we give an example of a simple nesting of `for` structures. Let CPF14a = `file:fora.c`(

```
#include <stdio.h>

main()
{   int row, col;

    for (row = 10; row >= 1; row--) {
       for (col = 1; col <= row; col++) {
          printf("*");
       }
       printf("\n");
    }
    return 0;
}
```

). This program produces

```
**********
*********
********
*******
******
*****
****
***
**
*
```

It should be noted that a `for` loop can be used in more general ways than the simple, iterative examples we have seen so far. For example, the 'control variable' of a `for` loop does not have to be an integer, and it does not have to be incremented by an additive increment. It could be incremented by a multiplicative factor, or it could be a floating-point variable; moreover, there may be more than just a single control variable. Thus

```
for (x = 0,y = 4; ((x > 3) && (y < 9));
                              x++,y *= 2) {
   ...
}
for (x = 0,y = 4,z = 4000; z; z /= 10) {
   ...
}
```

are legal C constructs where the second loop will continue to iterate until `z` becomes 0. The practical application of such statements is not important here, we are just trying to illustrate peculiar features of C `for` that may be useful.

## 7.2 The `while` repetition structure

A `while` loop has one control expression, and executes as long as that expression is true. Its general syntax is

```
while (expr) {
   block of statements
}
```

CPF15 = `file:wh.c`(

```
#include <stdio.h>

main()
{   float f=100;

    while (f >= 0.000001) {
        printf("%11.7f\n", f);
        f/=3;
    }
    return 0;
}
```

) shows the following output:

```
100.0000000
 33.3333321
 11.1111107
  3.7037036
  1.2345679
  0.4115226
  0.1371742
  0.0457247
```

```
0.0152416
0.0050805
0.0016935
0.0005645
0.0001882
0.0000627
0.0000209
0.0000070
0.0000023
```

A `while` loop starts out like an `if` selection statement: if the condition expressed by `expr` is true, the body is executed. However, after executing the statement, the condition is tested again, and only if it is still true, is the statement executed again. This can also be observed with CPF15a = `file:wha.c`(

```
#include <stdio.h>

main()
{   float f=100;
    int i;

    while (f >= 0.000001) {
        for (i=1; i <= 3; i++) {
        printf("%11.7f", f);
        f/=3;
        }
        printf("\n");
    }
    return 0;
}
```

) where it is intended to print the values formatted in three rows. Its output reads

```
100.0000000 33.3333321 11.1111107
  3.7037036  1.2345679  0.4115226
  0.1371742  0.0457247  0.0152416
  0.0050805  0.0016935  0.0005645
  0.0001882  0.0000627  0.0000209
  0.0000070  0.0000023  0.0000008
```

Thus in the last run of the `while` loop, its condition is tested only after the values `0.0000070 0.0000023 0.0000008` are computed and printed.

As long as the condition remains true, the body of the loop is executed over and over again. As soon as the condition does not hold anymore—in the case of CPF15 after the 17th run—the program terminates. If the condition is false right at the start, the body of the loop is not executed at all.

There is no fundamental difference between the `while` and the `for` loop. In fact, given the general `for` loop

```
for (expr1; expr2; expr3) {
    block of statements
}
```

one can rewrite it as a `while` loop, moving the initialize and increment expressions to statements before and within the loop:

```
expr1;
while (expr2) {
   block of stements
   expr3;
}
```

Similarly, given the general `while` loop

```
while (expr) {
   block of statements
}
```

one can rewrite it as a `for` loop using suitable dummy expressions—i.e. well-formed expressions which are irrelevant for the execution of the program:

```
for (dummy1; expr; dummy2) {
   block of statements
}
```

However, in general, a `for` loop should be chosen if its three expressions manipulate the same variable, using the initialize, test, increment pattern. Otherwise a `while` loop is probably clearer.

## 7.3 The `do/while` repetition structure

The `do/while` repetition structure is similar to the `while` structure. Its general syntax is

```
do {
   block of statements
} while (expr);
```

However, whereas the `while` structure tests the loop-continuation condition at the beginning of the loop *before* the body of the loop is performed, the `do/while` structure does this only *after* the loop body is performed; therefore the body of a `do/while` structure will be executed at least once. Hence execution of CPF16 = `file:do.c`(

```
#include <stdio.h>

main()
{   int i = 1;

    do {
       printf("%d ",i);
    } while (++i <= 10);

    printf("\n");

    do {
       printf("%d\n",i);
    } while (++i <= 10);

    return 0;
}
```

) yields the following output:

```
1 2 3 4 5 6 7 8 9 10
11
```

As should be clear, a uniform correspondence between `while` and `do/while` loops exists in the following way: a `while` loop of the form

```
while (expr) {
   block of statements
}
```

can be represented by the single selection-`do/while` combination

```
if (expr) {
   do {
      block of statements
   } while (expr);
}
```

on the other hand, a faithful representation for

```
do {
   block of statements
} while (expr);
```

can be given by

```
block of statements

while (expr) {
   block of statements
}
```

# 8 Functions

Functions are a non-trivial subject; the beginnings are simple, however.

A function has a name by which it is called, and a list of zero or more arguments or parameters that are handed to it; it has a body containing the actual statements for carrying out the task the function is supposed to perform; and it may give back a return value, of a particular type. Consider CPF16 = `file:fun.c`(

```
#include <stdio.h>

main()
{   int n = 6, o = 7, p = 2;
    int m, x;
    int f() {return p + ++o;}

    m = n + p + ++o;

    printf("1 %d\n",m);

    n = 6;
    o = 7;
    p = 2;

    x = f();
    m = n + x;

    printf("2 %d\n",m);

    return 0;
}
```

). In the third line a function with name `f` is declared which has no parameters and which returns a value of type `int`. This program indicates that expanding function bodies can help to understand how a function operates, it produces:

```
1 16
2 16
```

Next consider the modification of the previous program where the assignments prior to the function call are omitted. Let CPF16a = `file:funa.c`(

```
#include <stdio.h>

main()
{   int n = 6, o = 7, p = 2;
    int m, x;
    int f() {return p + ++o;}

    m = n + p + ++o;
    printf("1 %d\n",m);

    x = f();
    m = n + x;
    printf("2 %d\n",m);

    return 0;
}
```

). This program demonstrates that expanding function bodies is not sufficient to understand how a function operates. Clearly the place in the program where `f` is called matters quite a lot.

```
1 16
2 17
```

## 8.1 Functions with parameters

Defining a function with parameters will change the picture: consider CPF17 = `file:funp.c`(

```
#include <stdio.h>

main()
{   int m, n = 6, o = 7, p = 2, x;
    int f(int a, int b) {
        return a + ++b;}

    m = n + p + ++o;

    printf("1 %d\n",m);
    printf("1 %d\n",o);


    n = 6;
    o = 7;
    p = 2;
    x = f(p,o);
    m = n + x;

    printf("2 %d\n",m);
    printf("2 %d\n",o);

    n = 6;
    o = 7;
    p = 2;
    x = f(2,7);
    m = n + x;

    printf("3 %d\n",m);
    printf("3 %d\n",o);

    n = 6;
    o = 7;
    p = 2;
    x = f(2,++o - 1);
    m = n + x;

    printf("4 %d\n",m);
    printf("4 %d\n",o);

    return 0;
}
```

). In this program the function `f` is defined which has return type `int`, and accepts two parameters of type `int`, `a` and `b`. The names `a` and `b` are arbitrary, and are used only within the definition of `f`. The caller of `f` only needs to know that two arguments of type `int` are expected; the caller does not need to know what name the function uses internally to refer to these arguments. In particular, the caller does not have to pass the values of two variables named `a` and `b`. The output on the screen reads:

```
1 16
1 8
2 16
2 7
3 16
3 7
4 16
4 8
```

We can conclude that C is *call by value*, which means that a function receives copies of the values of its arguments. That is, when the implementation of `f` receives the value of the variable `o`, it does *not* change the value of `o`. `f` merely receives a copy of `o`'s value!

In CPF17a = `file:funpa.c`(

```
#include <stdio.h>

main()
{   int m, n = 6, o = 7, p = 2, x;
    int f(int a, int b) {
        return a + ++b;}

    m = n + p + ++o;
    printf("1 %d\n",m);
    printf("1 %d\n",o);

    x = f(p,o);
    m = n + x;
    printf("2 %d\n",m);
    printf("2 %d\n",o);

    return 0;
}
```

) it is once more demonstrated that the place in a program where a function is called has its effects on the behavior of the program. This program produces

```
1 16
1 8
2 17
2 8
```

## 8.2 Placing functions in another file

When a program consists of many functions, it may be convenient to split them up into several source files. This means that when a change is made, only the source file containing the change has to be recompiled, not the whole program, and that functions can be reused by other programs.

The task of putting the pieces of a program together and producing the final executable falls to a tool called the *linker* (see also Appendix E). One may or may not need to invoke the linker explicitly; a compiler often invokes it automatically, as needed. The linker looks through the program, sorting out the external declarations. The compiler has noted the definitions made by the program, as well as the function declarations used by it but defined elsewhere, the so-called prototypes . For each function used but not defined by the program, the linker looks for another source that does define that function.

Let CPF17b = `file:funpb.c`(

```
#include <stdio.h>

main()
{   int m, n = 6, o = 7, p = 2, x;
    extern int f(int, int);

    m = n + p + ++o;
    printf("1 %d\n",m);
    printf("1 %d\n",o);

    x = f(p,o);
    m = n + x;
    printf("2 %d\n",m);
    printf("2 %d\n",o);

    return 0;
}
```

) ∪ `file:extfunpb.c`(

```
int f(int a, int b) {
   return a + ++b;}
```

).

This program can be compiled and linked together by means of the command `gcc funpb.c extfunpb.c` (with the file containing `main(){..}` being mentioned first). The resulting executable (`a.out`) computes

```
1 16
1 8
2 17
2 8
```

Let CPF17c = `file:funpc.c`(

```
#include <stdio.h>

main()
{   int m, n = 6, o = 7, p = 2, x;
    extern int h();
    extern int f(int, int);

    m = h() + p + ++o;
    printf("1 %d\n",m);

    n = 6;
    o = 7;
    p = 2;
    x = f(p,o);
    m = h() + x;
    printf("2 %d\n",m);

    return 0;
}
```

) ∪ `file:extfunpc.c`(

```
int f(int a, int b) {
   return a + ++b;
}

int h() {
   return 3;
}
```

). This program produces

```
1 13
2 13
```

Another way of combining declarations is as follows: CPF17d = `file:funpd.c`(

```
#include <stdio.h>

main()
{   int m, o = 7, p = 2, x;
    extern int h();
    extern int f(int, int);

    x = f(p,o);
    m = h() + x;
    printf("%d\n",m);
    return 0;
}

int h() {
   return 3;}
```

) ∪ `file:extfunpb.c`(

```
int f(int a, int b) {
   return a + ++b;}
```

). This program produces

```
13
```

## 8.3 Recursion

The programs we have discussed so far are generally structured in a hierarchical manner. For some types of problems, it is useful to have functions call themselves. A *recursive* function is a function that calls itself either directly or indirectly through another function. Recursion is a complex topic discussed at length in upper-level computer science courses. In this subsection, we illustrate this phenomenon by two simple examples.

First we reconsider CPF14a. Let CPF18 = `file:rec.c`(

```
#include <stdio.h>

main()
{   int m;
    extern int p(int);
```

```
    for (m = 10; m >= 1; m--){
       p(m);
    }

    return 0;
}

int p(int i) {
    if ( i == 1 ) {
       printf("*\n");
    } else {
       printf("*");
       p(--i);
    }
    return 0;
}
```

). Again we have

```
**********
*********
********
*******
******
*****
****
***
**
*
```

The second example involves two recursive functions. Here the first one is calling the second. Let CPF18a = `file:reca.c`(

```
#include <stdio.h>

main()
{   int m;
    extern int p(int, int);
    extern int q(int);

    for (m = 10; m >= 1; m--){
       p(m, m-9);
    }

    return 0;
}

int p(int i, int j) {
    extern int q(int);

    if ( i == 1 ) {
       printf("* ");
       q(j);
    } else {
       printf("*");
       p(--i,j);
    }
    return 0;
}

int q(int i) {
    if ( i == 1 ) {
       printf("*\n");
    } else {
       printf("*");
       q(++i);
    }
    return 0;
}
```

). This program produces

```
********** *
********* **
******** ***
******* ****
****** *****
***** ******
**** *******
*** ********
** *********
* **********
```

## 8.4 Global and static local variables

Like functions, variables can be declared externally. So-called *global* variables are created by placing variable declarations outside any function definition, in particular outside `main`. They can be referenced by any function that follows their declaration. We will explore this possibility in the next example. Let CPF19 = `file:extv.c`(

```
#include <stdio.h>

main()
{   int m = 5;
    int n = m + 7;
    extern int x;
    extern int y;
    extern int g(void);

    printf("0 %d\n",m);
    printf("1 %d\n",x);
    x = 1;
    printf("2 %d\n",x);
    y = 2;
    printf("3 %d\n",y);
    printf("4 %d\n",g());
    printf("5 %d\n",y);
    printf("6 %d\n",g());
    printf("7 %d\n",y);
    printf("8 %d\n",n);

    return 0;
}

int g(void) {
    int j = 3;
    extern int y;
    printf("a %d\n",++j);
    return ++y;
}

int x = 6;

int y;

int m = 2 + 1;
```

). This program produces:

```
0 5
1 6
2 1
3 2
a 4
4 3
5 3
a 4
6 4
7 4
8 12
```

A local variable (introduced inside a function body) can keep the value it obtained after an execution of the function body and start with that value when the next call of the function body takes place. This is achieved by declaring that variable as static. Next consider CPF20 = `file:stat.c`(

```
#include <stdio.h>

main()
{   int y;
    extern int h(int);

    printf("0 %d\n",h(0));
    printf("1 %d\n",h(0));
    printf("2 %d\n",h(0));
    printf("3 %d\n",h(0));
    printf("4 %d\n",h(0));
    y = h(0);
    printf("5 %d\n",y);
    y = h(y);
    printf("6 %d\n",y);
    y = h(y);
    printf("7 %d\n",y);

    return 0;
}

int h(int u) {
    int i = 3;
    static int j = 3;
    printf("a %d %d\n",++i, ++j);
    return ++u;
}
```

) This program produces

```
a 4 4
0 1
a 4 5
1 1
a 4 6
2 1
```

```
a 4 7
3 1
a 4 8
4 1
a 4 9
5 1
a 4 10
6 2
a 4 11
7 3
```

## 8.5 Side-effects

The order of evaluation of expression is not fixed by the language standard. Function calls can introduce side-effects that differ when different orders of evaluation are checked. Such function calls may be places where a program fails to be portable. We see no reason to condemn the program below on such grounds. Portability is a global property of a program that cannot be inferred from some simplistic criteria considering the absence of expressions of a certain kind. This argument makes sense in the case of multiple side-effect operators also. Let CPF21 = `file:see.c`(

```
#include <stdio.h>

main()
{   extern int g(int);

    printf("1 %d\n",
        g(1) * (g(1) + 1));

    printf("2 %d\n",
        g(1) * (g(1) + 1));

    printf("3 %d\n",
        g(1) * (g(1) + 1));

    return 0;
}

int g(int i) {
    static int j = 0;
    j = j + 1;
    return i * j;
}
```

). This program produces

```
1 3
2 15
3 35
```

A simple modification is CPF21a = `file:sef.c`(

```
#include <stdio.h>

main()
{  extern int g(int);

    printf("1 %d\n",
         (g(1) + 1) * g(1));

    printf("2 %d\n",
         (g(1) + 1) * g(1));

    printf("3 %d\n",
         (g(1) + 1) * g(1));

    return 0;
}

int g(int i) {
    static int j = 0;
    j = j + 1;
    return i * j;
}
```

) producing

```
1 4
2 16
3 36
```

The interesting point about these programs is that the language standard will not prescribe an order of evaluation for multiplication. From the point of view of the language definition the two programs are equivalent. In fact, a compiler need not be consistent and is allowed to choose different evaluation orders at every moment.

## 9 Pointers

Pointers are often thought to be the most difficult aspect of C. It is true that many people have various problems with pointers, and that many programs founder on pointer-related bugs. However, many of the problems are not so much with the pointers per se but rather with the memory location they point to, and more specifically, when there is no valid memory location which they point to. As long as pointers are used with care (i.e. pointing to valid memory locations), pointers can be useful, powerful, and relatively trouble-free tools.

### 9.1 Basic pointer operations

A pointer is a variable that points at, or refers to, another variable—e.g. a pointer variable of type 'pointer to int' points at the `int` variable `i`, say—and given a pointer variable, one can ask questions like, "What is the value of this pointer?" (answer: the memory location of `i`), and "What is the value of the variable that this pointer points to?" (answer: the value of `i`). Let CPF22 = `file:point.c`(

```
#include <stdio.h>

main()
{   int i = 5;
    float f = 3.67;
    char c = 'P';

    int *pi;
    float *pf;
    char *pc;

    printf("1 %8d %8f %8c\n",i,f,c);
    printf("1 %8p %8p %8p\n",&i,&f,&c);
    printf("1 %8p %8p %8p\n",pi,pf,pc);
    printf("1 %8p %8p %8p\n",&pi,&pf,&pc);

    pi = &i;
    pf = &f;
    pc = &c;
    printf("2 %8d %8f %8c\n",i,f,c);
    printf("2 %8p %8p %8p\n",&i,&f,&c);
    printf("2 %8p %8p %8p\n",pi,pf,pc);
    printf("2 %8d %8f %8c\n",*pi,*pf,*pc);
    printf("2 %8p %8p %8p\n",&pi,&pf,&pc);

    i++;
    f /= 7;
    c--;
    printf("3 %8d %8f %8c\n",i,f,c);
    printf("3 %8p %8p %8p\n",&i,&f,&c);
    printf("3 %8p %8p %8p\n",pi,pf,pc);
    printf("3 %8d %8f %8c\n",*pi,*pf,*pc);
    printf("3 %8p %8p %8p\n",&pi,&pf,&pc);

    *pi = 5;
    *pf = 3.67;
    *pc = 'P';
    printf("4 %8d %8f %8c\n",i,f,c);
    printf("4 %8p %8p %8p\n",&i,&f,&c);
    printf("4 %8p %8p %8p\n",pi,pf,pc);
    printf("4 %8d %8f %8c\n",*pi,*pf,*pc);
    printf("4 %8p %8p %8p\n",&pi,&pf,&pc);

    (*pi)++;
    *pf /= 7;
    (*pc)--;
    printf("5 %8d %8f %8c\n",i,f,c);
    printf("5 %8p %8p %8p\n",&i,&f,&c);
    printf("5 %8p %8p %8p\n",pi+1,pf,pc-1);
    printf("5 %8d %8f %8c\n",*pi,*pf,*pc);
    printf("5 %8p %8p %8p\n",&pi,&pf,&pc);

    return 0;
}
```

). The output on the screen reads

```
1        5 3.670000        P
1 effff0c4 effff0c0 effff0bf
1        3    10944 effff0d8
1 effff0b8 effff0b4 effff0b0
2        5 3.670000        P
2 effff0c4 effff0c0 effff0bf
2 effff0c4 effff0c0 effff0bf
2        5 3.670000        P
2 effff0b8 effff0b4 effff0b0
3        6 0.524286        O
3 effff0c4 effff0c0 effff0bf
```

```
3 effff0c4 effff0c0 effff0bf
3        6 0.524286        0
3 effff0b8 effff0b4 effff0b0
4        5 3.670000        P
4 effff0c4 effff0c0 effff0bf
4 effff0c4 effff0c0 effff0bf
4        5 3.670000        P
4 effff0b8 effff0b4 effff0b0
5        6 0.524286        0
5 effff0c4 effff0c0 effff0bf
5 effff0c8 effff0c0 effff0be
5        6 0.524286        0
5 effff0b8 effff0b4 effff0b0
```

Note that `printf` conversion specification `%p` outputs memory locations as hexadecimal integers (see also Table 4).

The following example shows that different pointers can refer to the same variable, that pointers can be redirected, and that pointers can effect a variable indirectly using other pointers. Let CPF22a = `file:pointa.c`(

```
#include <stdio.h>

main()
{   int i = 5, j = 7;
    int *pi, *pj;

    printf("1 %8d %8d\n",i,j);
    printf("1 %8p %8p\n",&i,&j);
    printf("1 %8p %8p\n",&pi,&pj);

    pi = &i;
    pj = &j;
    printf("2 %8d %8d\n",i,j);
    printf("2 %8p %8p\n",&i,&j);
    printf("2 %8p %8p\n",pi,pj);
    printf("2 %8d %8d\n",*pi,*pj);

    pi = pj;
    printf("3 %8d %8d\n",i,j);
    printf("3 %8p %8p\n",&i,&j);
    printf("3 %8p %8p\n",pi,pj);
    printf("3 %8d %8d\n",*pi,*pj);

    pi = &i;
    pj = &j;
    *pj = *pi;
    printf("4 %8d %8d\n",i,j);
    printf("4 %8p %8p\n",&i,&j);
    printf("4 %8p %8p\n",pi,pj);
    printf("4 %8d %8d\n",*pi,*pj);

    return 0;
}
```

). This program produces

```
1        5        7
1 effff0c4 effff0c0
1 effff0bc effff0b8
2        5        7
2 effff0c4 effff0c0
2 effff0c4 effff0c0
2        5        7
3        5        7
3 effff0c4 effff0c0
3 effff0c0 effff0c0
3        7        7
4        5        5
4 effff0c4 effff0c0
4 effff0c4 effff0c0
4        5        5
```

### 9.2 Pointers and arrays; pointer arithmetic

An array is a group of memory locations related by the fact that they all have the same name and the same type. To refer to a particular location or cell in the array, one has to specify the name of the array and the position number of the particular cell in the array. The following example demonstrates that pointers do not have to point to single variables, but can also point at the cells of an array. Let CPF23 = `file:arr.c`(

```
#include <stdio.h>

main()
{   int i;
    int a[5] = {10, 20, 30, 40, 50}, b[5];
    int *pa, *pb, *q;
```

```
printf("1 ");
for (i = 0; i <= 4; i++) {
   printf("%d ", a[i]);
}
printf("\n");

pa = &a[0];
printf("1 %d %d\n", a[0], *pa);

pa = &a[3];
printf("2 %d %d\n", a[3], *pa);

pa++;
printf("3 %d %d\n", a[4], *pa);

pa = &a[0];
for (i = 0; i <= 4; i++) {
   printf("4 %d %d\n", a[i],
                    *(pa + i));
}

printf("5 ");
for (i = 0; i <= 4; i++) {
   *(pa + i) += 5;
   printf("%d ", a[i]);
}
printf("\n");

pb = &b[0];
q = &a[4];
for (pa = &a[0]; pa <= q; pa++) {
   *pb++ = *pa;
}
printf("6 ");
for (i = 0; i <= 4; i++) {
   printf("%d ", b[i]);
}
printf("\n");


printf("7 %d %d %d\n",
           a[10], a[-1], a[5]);

pa = &a[3];
printf("8 %d %d\n", *(pa + 12),
                                *(pa - 5));

    return 0;
}
```

). The corresponding output is

```
1 10 20 30 40 50
1 10 10
2 40 40
3 50 50
4 10 10
4 20 20
4 30 30
4 40 40
4 50 50
5 15 25 35 45 55
6 15 25 35 45 55
7 0 -268439376 -268439228
8 0 -277195288
```

Note that when one is doing pointer arithmetic, one has to remember how big the array the pointer points into is, so that one does not ever point outside it. Since the array `a` has 5 elements, access of `a[10]` or `a[-1]` or even `a[5]` (as we have seen, the valid subscripts for a 5-element array run from 0 to 4) is comparable to the access of an uninitialized variable. The same holds for access of `*(pa + 12)` or `*(pa - 5)`, in the case `pa` is pointing to `a[3]`.

As we have seen, arrays and pointers can be used in related ways. The next example examines more similarities between these concepts. Let CPF23a = `file:arra.c`(

```
#include <stdio.h>

main()
{   int i;
    int a[5] = {10, 20, 30, 40, 50};
    int *pa, *pb;

    printf("1 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", a[i]);
    }
    printf("\n");
```

```
    pa = &a[0];
    printf("2 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", *(pa + i));
    }
    printf("\n");

    printf("3 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", a[i] + i);
    }
    printf("\n");

    printf("4 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", *pa + i);
    }
    printf("\n");

    printf("5 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", *pa++ + i);
    }
    printf("\n");

    pa = a;
    printf("6 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", *(pa + i));
    }
    printf("\n");

    pa = a + 3;
    pb = pa;
    printf("7 %d %d\n", *pa, *pb);

    return 0;
}
```

). Now we have the following output:

```
1 10 20 30 40 50
2 10 20 30 40 50
3 10 21 32 43 54
4 10 11 12 13 14
5 10 21 32 43 54
6 10 20 30 40 50
7 40 40
```

There are also differences, of course. As we have seen above, one can assign two pointer variables: i.e.

```
int *pa, *pb;
pa = pb;
```

is legal; likewise,

```
int a[5] = {10, 20, 30, 40, 50};
int *pa;
pa = a;
```

is legal. However, one *cannot* assign two arrays; the code

```
int a[5] = {10, 20, 30, 40, 50}, b[5];
b = a;     /* WRONG */
```

is illegal and does not compile. The error message under GNU compilation reads `incompatible types in assignment`.

## 9.3 Call by reference

Earlier, we learned that functions in C receive copies of their arguments. This means that C uses call by value, i.e. a function can modify one of its arguments without modifying the value in the caller. The reason for this is that, when a function is called, the copies of the arguments are made as if by assignment. Since arrays cannot be assigned[5], one has to pass the name of an array—which corresponds to the address of the first cell of the array—to a function. Thus the called function can actually modify the values in the callers' original array. This type of function call is called *call by reference*.

Let CPF24 = `file:ref.c`(

```
#include <stdio.h>

main()
{   int i, a[5];
    extern int g(int);
```

[5]Note that, in contrast, it is perfectly legal to assign individual cells of an array.

```
    extern int f(int []);

    printf("Enter 5 integers: ");
    for (i = 0; i <= 4; i++) {
       scanf("%d", &a[i]);
    }

    printf("1 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", g(a[i]));
    }
    printf("\n");

    printf("2 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", a[i]);
    }
    printf("\n");

    f(a);
    printf("3 ");
    for (i = 0; i <= 4; i++) {
       printf("%d ", a[i]);
    }
    printf("\n");

    return 0;
}
int g(int n) {
   return ++n;
}

int f(int a[]) {
   int i;
   for (i = 0; i <= 4; i++) {
       a[i] *= 2;
   }
   return 0;
}
```

). This interactive program produces for example

```
Enter 5 integers: 10 20 30 40 50
1 11 21 31 41 51
2 10 20 30 40 50
3 20 40 60 80 100
```

Given the equivalence between arrays and pointers, the above program can be rewritten using a pointer instead of an array. If we do this we have to initialize the pointer with the address of the first cell of an array of length 5. This can be done by the standard library function `malloc` which is declared in the header file `stdlib.h`. The following program is a variation of the previous one showing an example of so-called *dynamic memory allocation* where instead of a fixed-size array an array of variable length is used; the actual length of that array is determined by the user. Let CPF24a = `file:refa.c`(

```
#include <stdio.h>
#include <stdlib.h>

main()
{   int i, max, *p;
    extern int g(int);
    extern int f(int *, int);

    printf("Enter max:");
    scanf("%d", &max);
    printf("Enter %d integers: ", max);
    p = malloc(max * sizeof(int));
    for (i = 0; i <= (max-1); i++) {
       scanf("%d", p++);
    }
    p -= max;

    printf("1 ");
    for (i = 0; i <= (max-1); i++) {
       printf("%d ", g(*(p+i)));
    }
    printf("\n");

    printf("2 ");
    for (i = 0; i <= (max-1); i++) {
       printf("%d ", *(p+i));
    }
    printf("\n");

    f(p,max);
    printf("3 ");
    for (i = 0; i <= (max-1); i++) {
```

```
        printf("%d ", *(p+i));
     }
     printf("\n");

     return 0;
  }
  int g(int n) {
     return ++n;
  }

  int f(int *p, int max) {
     int i;
     for (i = 0; i <= (max-1); i++) {
          *p *= 2;
          p++;
     }
     return 0;
  }
```

). A session may look as follows:

```
Enter max: 8
Enter 8 integers: 10 20 30 40 50 60 70 80
1 11 21 31 41 51 61 71 81
2 10 20 30 40 50 60 70 80
3 20 40 60 80 100 120 140 160
```

# 10 Structures

*Structures* are collections of related variables—sometimes referred to as *aggregates*—under one name. They are derived data types in the sense that they may contain variables of many different types—in contrast to arrays that contain only elements of the same data type. Structures are commonly used to define records to be stored in files.

Two operators are used to access members of structures: the *structure member operator* `.`—also called the *dot operator*—and the *structure pointer operator* `->`—also called the *arrow operator*. The first operator accesses a structure member via the structure variable name, the second via a pointer to the structure.

Let CPF25 = `file:str.c(`

```
#include <stdio.h>
#include <stdlib.h>

main()
{   struct record {
       char c;
       int i;
       float f;
    };

    struct record ra;
    struct record *pra, *prb;

    ra.c = 'A';
    ra.i = 12;
    ra.f = 4.73;
    printf("1 %c %2d %5.2f\n",
                     ra.c, ra.i, ra.f);

    pra = &ra;
    printf("2 %c %2d %5.2f\n",
         (*pra).c, (*pra).i, (*pra).f);
    printf("3 %c %2d %5.2f\n",
         pra -> c, pra -> i, pra -> f);

    ra.c++;
    --ra.i;
    ra.f /= 3;
    printf("4 %c %2d %5.2f\n",
                     ra.c, ra.i, ra.f);

    --(pra -> c);
    ++(pra -> i);
    (pra -> f) *= 3;
    printf("5 %c %2d %5.2f\n",
         pra -> c, pra -> i, pra -> f);

    prb = malloc(sizeof(struct record));
    prb -> c = '$';
    prb -> i = 23;
    prb -> f = 0.99;
    pra = prb;
    printf("6 %c %2d %5.2f\n",
          pra -> c, pra -> i, pra -> f);

    return 0;
```

```
}
```

). The corresponding output is:

```
1 A 12  4.73
2 A 12  4.73
3 A 12  4.73
4 B 11  1.58
5 A 12  4.73
6 $ 23  0.99
```

Structures can be *self-referential*, i.e. containing a pointer that points to a structure of the same structure type. A *linked list* (see Figure 2) is a linear collection of self-referential structures, called *nodes*, connected by pointer *links*. A linked list is accessed via a pointer to the first node of the list. Subsequent nodes are accessed via the link pointer member stored in each node. By convention, the link pointer in the last node of a list is set to the `NULL` pointer which points to nothing (see also Appendix D). Data are stored in a linked list dynamically—each node is created as necessary. The following example illustrates a constrained version of this concept where a new node may be added to a list only at the top (the general version allows arbitrary insertions and deletions). Let CPF26 = `file:ll.c`(

```
#include <stdio.h>
#include <stdlib.h>

main()
{   struct node {
       int i;
       float f;
       struct node *next;
    };

    struct node *temp, *ll = NULL;
    int i,j;
    float f;

    printf("Enter 1 to push a node,\n");
    printf("enter 2 to terminate: ");
    scanf("%d", &j);

    while (j == 1) {
```

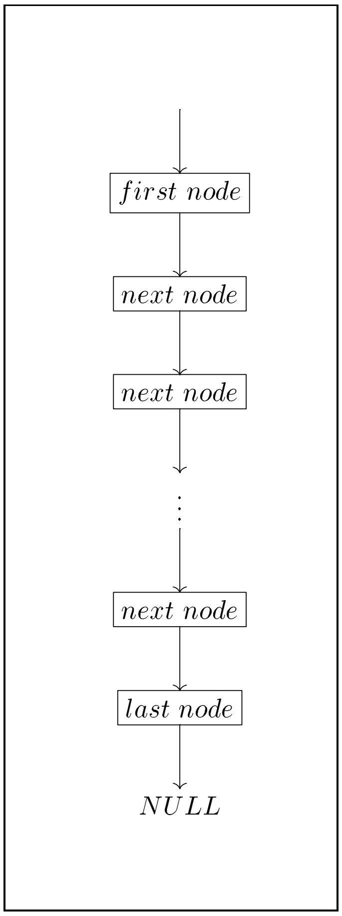



Figure 2: Linked list

```
        temp = malloc(sizeof(struct node));
        printf("Enter a node:\n");
        scanf("%i%f", &i, &f);
        temp -> i = i;
        temp -> f = f;
        temp -> next = ll;
        ll = temp;
        printf("Enter your choice: ");
        scanf("%d", &j);
    }

    while (ll != NULL) {
        printf("[%d, %f] -> \n",
                ll -> i, ll -> f);
        ll = ll -> next;
    }
    printf("NULL\n");
    return 0;
}
```

). An interactive session produces output like

```
Enter 1 to push a node,
enter 2 to terminate: 1
Enter a node:
12 .67
Enter your choice: 1
Enter a node:
3 1.452
Enter your choice: 1
Enter a node:
7 .0
Enter your choice: 2
[7, 0.000000] ->
[3, 1.452000] ->
[12, 0.670000] ->
NULL
```

# 11 Remarks

Many more aspects of C can be investigated and illustrated. We will by no means attempt to provide a complete picture which is outside the range of daily programmer experience anyhow. We hope that the reader has acquired some understanding of the expressive power as well as of the pittfalls related to side-effect constructions in C. The language standard [1] has to be consulted if a precise understanding of these matters is required. Further we hope to have covered more ground than the average C programming textbook, thus enabling the reader a more comprehensive view of the language semantics than otherwise would have been obtained without having to study the agonizing details of formal models of the C program execution model.

# 12 Appendix

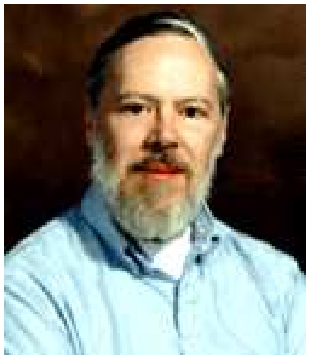
# A Historical notes

## A.1 A short history of C

The programming language C evolved from two previous languages, BCPL and B. BCPL (Basic Combined Programming Language) was developed in 1967 by Martin Richards as a language for writing operating systems software and compilers. At some point early in its life, Richards brought his BCPL compiler over to AT&T Bell Laboratories. There Ken Thompson was impressed by the concept of a portable, high-level language. He designed and implemented B modelling many features in his language after their counterparts in BCPL, and used B to create early versions of the UNIX operating system at AT&T Bell Laboratories in 1970 on a DEC PDP-7 computer (a cramped and spartan hardware environment with 8K 18-bit words of memory). Both BCPL and B were "typeless" languages — every data item occupied one "word" in memory and the burden of treating a data item as a whole number or a real number, for example, fell on the shoulders of the programmer. As an illustration, the following BCPL program prints the factorials of the first 10 natural numbers:

```
// FACTORIAL
GET "LIBHDR"
LET START () BE $(
   LET F(N) = N=0 -> 1,N*F(N-1)
   FOR I = 1 TO 10 DO
       WRITEF("F(%N),=%N*N",I,F(I))
   FINISH
$)
```

BCPL is described in the book *BCPL: The Language and Its Compiler* by Martin Richards and Colin Whitby-Stevens [13], which also includes the full source code for a portable compiler for BCPL. The book is interesting not just from the point of view of the language, but also because it provides a view of compiler technology and approaches to program portability in the late 60s and early 70s. A description of B can be found in Steve Johnson and Brian Kernighan [9].



Dennis Ritchie

C came into being in the years 1969-1973, in parallel with the early development of the UNIX operating system. The most creative period occurred during 1972 when Dennis Ritchie turned B into C, keeping most of B's syntax while adding types and many other changes, and writing the first compiler. The BCPL program translated into C looks like the following.

```
/* Factorial */
int f(int);
main()
{
  int i;
  for (i = 1; i <= 10; i++)
    printf("f(%d) = %d\n", i, f(i));
  return 0;
}
int f(int i);
{
  if (i <= 1)
    return 1;
  else return i*f(i-1);
}
```

Another spate of changes peaked between 1977 and 1979, when portability of the UNIX system was being demonstrated. In the middle of this second period, the first widely available description of the language appeared: *The C Programming Language* [10], often called the "white book". During the 1980s the use of the C language spread widely, and compilers became available on nearly every machine architecture and operating system; in particular it became popular as a programming tool for personal computers, both for manufacturers of commercial software for these machines, and for end-users interested in programming. At the start of the decade, nearly every compiler was based on Johnson's *pcc*, a C compiler intended to be easy to retarget to new machines; by 1985 there were many independently-produced compiler products. Finally, in the middle 1980s, the language was officially standardized by the ANSI X3J11 committee (see Section A.2), which made further changes.

BCPL, B, and C all fit firmly in the traditional procedural family typified by Fortran and Algol 60. They are particularly oriented towards system programming, are small and compactly described, and are amenable to translation by simple compilers. They are "close to the machine" in that the abstractions they introduce are readily grounded in the concrete data types and operations supplied by conventional computers, and they rely on library routines for input-output and other interactions with an operating system. With less success, they also use library procedures to specify interesting control constructs such as coroutines and procedure closures. At the same time, their abstractions lie at a sufficiently high level that, with care, portability between machines can be achieved.

Until the early 1980s, although compilers existed for a variety of machine architectures and operating systems, C was almost exclusively associated with UNIX; more recently, its use has spread much more widely, and today it is among the languages most used throughout the computer industry. C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.

C and even B have several direct descendants, but do not rival Pascal in generating progeny. Recent descendants of C proper include Concurrent C [8], Objective C [5], C* [16], and especially C++ [14]. The language is also widely used as an intermediate representation (essentially, as a portable assembly language) for a wide variety of compilers, both for direct descendants like C++, and independent languages like Modula 3 [12] and Eiffel [11].

### A.2 Standardization

By 1982 it was clear that C needed formal standardization. The best approximation to a standard, the first edition of [10], no longer described the language in actual use; in particular, it mentioned neither the `void` or `enum` types. While it foreshadowed the newer approach to structures, only after it was published did the language support assigning them, passing them to and from functions, and associating the names of the members firmly with the structure or union containing them. Although compilers distributed by AT&T incorporated these changes, and most of the purveyors of compilers not based on *pcc* quickly picked them up, there remained no complete, authoritative description of the language.

The first edition of [10] was also insufficiently precise on many details of the language, and it became increasingly impractical to regard *pcc* as a "reference compiler"; it did not perfectly embody even the language described by [10], let alone subsequent extensions. Finally, the incipient use of C in projects subject to commercial and government contract meant that the imprimatur of an official standard was important. Thus ANSI established the X3J11 committee in the summer of 1983, with the goal of producing a C standard. X3J11 produced its report [1] at the end of 1989, and subsequently this standard was accepted by ISO/IEC 9899-1990. In Appendix B, we have listed the ANSI C grammar.

By the rules of the standards process, the current activity of the X3J11 committee is confined to issuing interpretations on the existing standard. However, the informal NCEG (Numerical C Extension Group) has been officially accepted as subgroup X3J11.1 and is considering extensions to C. As the name implies, many of these possible extensions are intended to make the language more suitable for numerical use: for example, multi-dimensional arrays whose bounds are dynamically determined, incorporation of facilities for dealing with IEEE arithmetic, and making the language more effective on machines with vector or other advanced architectural features.

## B ANSI C grammar

```
primary_expr
: identifier
| CONSTANT
| STRING_LITERAL
| '(' expr ')'
;

postfix_expr
: primary_expr
| postfix_expr '[' expr ']'
| postfix_expr '(' ')'
| postfix_expr '(' argument_expr_list ')'
| postfix_expr '.' identifier
| postfix_expr PTR_OP identifier
| postfix_expr INC_OP
| postfix_expr DEC_OP
;

argument_expr_list
: assignment_expr
| argument_expr_list ',' assignment_expr
;

unary_expr
: postfix_expr
| INC_OP unary_expr
| DEC_OP unary_expr
| unary_operator cast_expr
| SIZEOF unary_expr
| SIZEOF '(' type_name ')'
;

unary_operator
: '&'
| '*'
| '+'
| '-'
| '~'
| '!'
;

cast_expr
: unary_expr
| '(' type_name ')' cast_expr
;
```

```
multiplicative_expr
: cast_expr
| multiplicative_expr '*' cast_expr
| multiplicative_expr '/' cast_expr
| multiplicative_expr '%' cast_expr
;

additive_expr
: multiplicative_expr
| additive_expr '+' multiplicative_expr
| additive_expr '-' multiplicative_expr
;

shift_expr
: additive_expr
| shift_expr LEFT_OP additive_expr
| shift_expr RIGHT_OP additive_expr
;

relational_expr
: shift_expr
| relational_expr '<' shift_expr
| relational_expr '>' shift_expr
| relational_expr LE_OP shift_expr
| relational_expr GE_OP shift_expr
;

equality_expr
: relational_expr
| equality_expr EQ_OP relational_expr
| equality_expr NE_OP relational_expr
;

and_expr
: equality_expr
| and_expr '&' equality_expr
;

exclusive_or_expr
: and_expr
| exclusive_or_expr '^' and_expr
;

inclusive_or_expr
: exclusive_or_expr
| inclusive_or_expr '|' exclusive_or_expr
;

logical_and_expr
: inclusive_or_expr
| logical_and_expr AND_OP inclusive_or_expr
;

logical_or_expr
: logical_and_expr
| logical_or_expr OR_OP logical_and_expr
;

conditional_expr
: logical_or_expr
| logical_or_expr '?' logical_or_expr
                  ':' conditional_expr
;

assignment_expr
: conditional_expr
| unary_expr assignment_operator assignment_expr
;

assignment_operator
: '='
| MUL_ASSIGN
| DIV_ASSIGN
| MOD_ASSIGN
| ADD_ASSIGN
| SUB_ASSIGN
| LEFT_ASSIGN
| RIGHT_ASSIGN
| AND_ASSIGN
| XOR_ASSIGN
| OR_ASSIGN
;

expr
: assignment_expr
| expr ',' assignment_expr
;

constant_expr
: conditional_expr
;

declaration
: declaration_specifiers ';'
| declaration_specifiers init_declarator_list ';'
;

declaration_specifiers
: storage_class_specifier
| storage_class_specifier declaration_specifiers
| type_specifier
| type_specifier declaration_specifiers
;

init_declarator_list
: init_declarator
| init_declarator_list ',' init_declarator
;

init_declarator
: declarator
| declarator '=' initializer
;
```

```
storage_class_specifier
: TYPEDEF
| EXTERN
| STATIC
| AUTO
| REGISTER
;

type_specifier
: CHAR
| SHORT
| INT
| LONG
| SIGNED
| UNSIGNED
| FLOAT
| DOUBLE
| CONST
| VOLATILE
| VOID
| struct_or_union_specifier
| enum_specifier
| TYPE_NAME
;

struct_or_union_specifier
: struct_or_union identifier
                  '{' struct_declaration_list '}'
| struct_or_union '{' struct_declaration_list '}'
| struct_or_union identifier
;

struct_or_union
: STRUCT
| UNION
;

struct_declaration_list
: struct_declaration
| struct_declaration_list struct_declaration
;

struct_declaration
: type_specifier_list struct_declarator_list ';'
;

struct_declarator_list
: struct_declarator
| struct_declarator_list ',' struct_declarator
;

struct_declarator
: declarator
| ':' constant_expr
| declarator ':' constant_expr
;

enum_specifier
: ENUM '{' enumerator_list '}'
| ENUM identifier '{' enumerator_list '}'
| ENUM identifier
;

enumerator_list
: enumerator
| enumerator_list ',' enumerator
;

enumerator
: identifier
| identifier '=' constant_expr
;

declarator
: declarator2
| pointer declarator2
;

declarator2
: identifier
| '(' declarator ')'
| declarator2 '[' ']'
| declarator2 '[' constant_expr ']'
| declarator2 '(' ')'
| declarator2 '(' parameter_type_list ')'
| declarator2 '(' parameter_identifier_list ')'
;

pointer
: '*'
| '*' type_specifier_list
| '*' pointer
| '*' type_specifier_list pointer
;

type_specifier_list
: type_specifier
| type_specifier_list type_specifier
;

parameter_identifier_list
: identifier_list
| identifier_list ',' ELIPSIS
;

identifier_list
: identifier
| identifier_list ',' identifier
;

parameter_type_list
: parameter_list
| parameter_list ',' ELIPSIS
;
```

```
parameter_list
: parameter_declaration
| parameter_list ',' parameter_declaration
;

parameter_declaration
: type_specifier_list declarator
| type_name
;

type_name
: type_specifier_list
| type_specifier_list abstract_declarator
;

abstract_declarator
: pointer
| abstract_declarator2
| pointer abstract_declarator2
;

abstract_declarator2
: '(' abstract_declarator ')'
| '[' ']'
| '[' constant_expr ']'
| abstract_declarator2 '[' ']'
| abstract_declarator2 '[' constant_expr ']'
| '(' ')'
| '(' parameter_type_list ')'
| abstract_declarator2 '(' ')'
| abstract_declarator2 '(' parameter_type_list ')'
;

initializer
: assignment_expr
| '{' initializer_list '}'
| '{' initializer_list ',' '}'
;

initializer_list
: initializer
| initializer_list ',' initializer
;

statement
: labeled_statement
| compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;

labeled_statement
: identifier ':' statement
| CASE constant_expr ':' statement
| DEFAULT ':' statement
;

compound_statement
: '{' '}'
| '{' statement_list '}'
| '{' declaration_list '}'
| '{' declaration_list statement_list '}'
;

declaration_list
: declaration
| declaration_list declaration
;

statement_list
: statement
| statement_list statement
;

expression_statement
: ';'
| expr ';'
;

selection_statement
: IF '(' expr ')' statement
| IF '(' expr ')' statement ELSE statement
| SWITCH '(' expr ')' statement
;

iteration_statement
: WHILE '(' expr ')' statement
| DO statement WHILE '(' expr ')' ';'
| FOR '(' ';' ';' ')' statement
| FOR '(' ';' ';' expr ')' statement
| FOR '(' ';' expr ';' ')' statement
| FOR '(' ';' expr ';' expr ')' statement
| FOR '(' expr ';' ';' ')' statement
| FOR '(' expr ';' ';' expr ')' statement
| FOR '(' expr ';' expr ';' ')' statement
| FOR '(' expr ';' expr ';' expr ')' statement
;

jump_statement
: GOTO identifier ';'
| CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN expr ';'
;

file
: external_definition
| file external_definition
;
```

```
external_definition
: function_definition
| declaration
;

function_definition
: declarator function_body
| declaration_specifiers declarator function_body
;

function_body
: compound_statement
| declaration_list compound_statement
;

identifier
: IDENTIFIER
;
```

# C Data types

All fundamental or primitive data types will have to be represented as numbers for them to be interpreted as bit strings. Thus the first step for characters and reals is to find a way of representing them as numbers. Characters (and certain control information that effect the lay-out of the output such as line feeds, carriage returns, tabs e.a.) can be expressed in ASCII code (see Table 2), reals (fractions) can be expressed in terms of the exponent and the mantisse. The second step is to translate the numerical values into bits. Here it is not merely a matter of expressing the value in base 2 rather than base 10, but also of assigning particular bits for specific functions, such as expressing a negative or a positive value. As a consequence, data types differ in the way they are represented in memory. This representation regards the number of bytes that are used per data type, the way the bytes are ordered per word (the system-dependent unit for the organization of memory), the operations that they allow, and the conversions that are applied to the bit strings (for instance: two's complement for integers, floating-point notation for reals, and ASCII for characters). For a full overview of the data types see Table 3 where the types are shown together with their sizes (in bytes) and the values that can be represented.

C is one of the few languages to allow *coercion*, that is forcing one variable of one type to be of another type. C allows this using the cast operator `(type)`. So:

```
int inumber;
float fnumber = 9.87;

inumber = (int)fnumber;
```

assigns 9 (the fractional part is thrown away) to `inumber`, and

```
int inumber=10;
float fnumber;

floatnumber = (float)inumber;
```

assigns 10.0 to `fnumber`.

Coercion can be used with any of the simple data types including `char`, so:

```
int inumber;
char letter = 'A';

inumber = (int)letter;
```

assigns 65 (the ASCII code for 'A') to `inumber`.

# D Formatted I/O

The input and output for programs can take place via the console or via files. In either case there are library functions to store the information in the appropriate memory addresses. Well-known functions controlling console output and input are `printf` and `scanf`, respectively.

The function `printf` is defined as follows:

```
int printf(char *format, args)
```

That is, `printf` prints to `stdout` the list of arguments according to a specified format string, and returns the number of characters printed. The format string has 3 types of objects:

- ordinary characters—these are copied to output;
- escape characters—these effect the lay-out of the output (see Table 5);

- conversion specifications—denoted by % (see Table 4).

Between % and the conversion character one can put

- a - (minus sign) for left justification,
- an integer number for the field width,
- an integer expression of the form m.n: here m specifies the field width, and n fixes the number of digits after the decimal point or the number of characters from a string.

Thus CPFD = `file:form.c`(

```
#include <stdio.h>

int main ()
{
  printf("%-8.1f%6.1s\n", 12.34567, "Hello");
  printf("%-8.2f%6.2s\n", 12.34567, "Hello");
  printf("%-8.3f%6.3s\n", 12.34567, "Hello");
  printf("%-8.4f%6.4s\n", 12.34567, "Hello");
  printf("%-8.5f%6.5s\n", 12.34567, "Hello");
  return 0;
}
```

) yields the screen output

```
12.3          H
12.35        He
12.346      Hel
12.3457    Hell
12.34567 Hello
```

The function `scanf` is defined as follows:

```
int scanf(char *format, args)
```

That is, `scanf` reads from `stdin`, puts the input in the address of the variables specified in the list of arguments, and returns the number of characters read. The format control string is similar to `printf`. Note that either the address of the variable or a pointer to one is required by `scanf`. In the case of an array or string, it suffices to give its name to `scanf`, since the name corresponds to the start address of the array or string, respectively. Thus, the program fragment

```
char string[80];
scanf("%s", string);
```

takes a string with a certain maximum length from the keyboard and stores it at the location in memory at which the variable `string` is stored.

In order to write to or read from a file rather than the console, the functions to be used are

```
int fprintf(FILE *stream,char *format,args)
int fscanf(FILE *stream,char *format,args)
```

These functions are similar to `printf` and `scanf` except that data is read from the file. First, however, the file in question has to be opened. The function

```
FILE *fopen(char *name, char *mode)
```

does this. That is, `fopen` returns a pointer to a `FILE`. The `name` string is the name of the file that is to be accessed. The `mode` string controls the type of access. If a file cannot be accessed for any reason a `NULL` pointer is returned. Modes include:

1. `r`—for reading,
2. `w`—for writing and if the file already exists, discarding its current content, and
3. `a`—for appending and, if the file does not yet exist, creating a file with the given name.

To open a file one must have a stream (file pointer) that points to a FILE structure. So to open a file called `myfile` for reading one would do:

```
FILE *stream myfile;/* declare a stream */
stream = fopen("myfile", "r");
```

It is good practice to check that a file is opened correctly: e.g. the fragment

```
if ((stream = fopen("myfile", "r")) == NULL)
   {  printf("Can't open %s\n", "myfile");
      exit(1);
   }
...
```

prints an appropriate message if the file cannot be opened, and terminates the program prematurely returning the value for unsuccessful termination to the calling environment.

# E C compilation

Roughly, compilation of a C program consists of the following phases: preprocessing, compilation and linking. We shall briefly highlight these features below.

Conceptually, the "preprocessor" is a translation phase that is applied to a source code before the compiler proper gets its hands on it. Generally, the preprocessor performs textual substitutions on source code. We mention two sorts of ways:

1. Macro substitution: replacing instances of one piece of text with another.
2. File inclusion: inserting the contents of another file into the source file, as if one had typed it all in there.

The syntax of the preprocessor is different from the syntax of the rest of C in several respects. First of all, the preprocessor is "line based". Each of the preprocessor directives (all of which begin with the # character) must start at the beginning of a line, and each directive ends at the end of the line. (The rest of C treats line ends as just another whitespace character, and does not care how the program text is arranged into lines.) Secondly, the preprocessor does not know about the structure of C—about functions, statements, or expressions.

A preprocessor line of the form

```
\#define name text
```

defines a macro with the given name, having as its value the given replacement text. After that (for the rest of the current source file), wherever the preprocessor sees that name, it will replace it with the replacement text. The name follows the same rules as ordinary identifiers (it can contain only letters, digits, and underscores, and may not begin with a digit). Since macros behave quite differently from normal variables (or functions), it is customary to give them names which are all capital letters (or at least which begin with a capital letter). The replacement text can be absolutely anything—it is not restricted to numbers, or simple strings, or anything. The most common use for macros is to propagate various constants around. Instead of using a line like

```
int array[100];
```

it is often much better to use a macro like

```
\#define MAX 100
int array[MAX];
```

For, if one ever wants to change the size of the array, one only has to do it in one place. Since the replacement text of a preprocessor macro can be anything, it can also be an expression, although one has to realize that the text is substituted (and perhaps evaluated) later. No evaluation is performed when the macro is defined. For example, suppose that one writes something like

```
\#define A 2
\#define B 3
\#define C A + B
```

followed by

```
int x = C * 2;
```

If `A`, `B`, and `C` were ordinary variables, one would expect `x` to end up with the value 10. But, since the preprocessor always substitutes text for macros exactly as they are written, the result will be

```
int x = 2 + 3 * 2;
```

Thus the compiler (using the normal precedence of multiplication over addition) ends up initializing `x` with the value 8! To guard against this sort of problem, it is always a good idea to include explicit parentheses in the definitions of macros which contain expressions. So

```
\#define C (A + B)
```

will expand to

```
int x = (2 + 3) * 2;
```

and `x` will be initialized to 10, as expected. Notice that there should not be a semicolon at the end of a `#define` line. This is just one of the ways that the syntax of the preprocessor is different from the rest of C. If one accidentally types

```
\#define MAX 100; /* WRONG */
```

then later declares

```
int array[MAX];
```

the preprocessor will expand it to

```
int array[100;];  /* WRONG */
```

which is a syntax error. This is what was meant above by saying that the preprocessor does not know about the syntax of C—in this last example, the value or replacement text for the macro `MAX` was the 4 characters `1 0 0 ;`, and that is exactly what the preprocessor substituted.

A line of the form

```
\#include <filename.h>
```

or

```
\#include "filename.h"
```

causes the contents of the file `filename.h` to be read, parsed[6], and compiled at that point. After `filename.h` is processed, compilation continues on the line following the `#include` line. The difference between the `< >` and `" "` forms is where the preprocessor searches for `filename.h`. As a general rule, it searches for files enclosed in `< >` in standard libraries, and it searches for files enclosed in `" "` in the directory containing the source file that is doing the including. Therefore, `" "` is usually used for "self-made" header files, and `< >` is usually used for headers which are provided by the system. The extension `.h` simply stands for "header", and reflects the fact that `#include` directives usually sit at the top (head) of a source file, and contain global declarations and definitions which one would otherwise put there. That extension is not mandatory—one can theoretically name one's own header files anything one wishes—but `.h` is traditional, and recommended. Header files typically contain only external declarations. For a full overview together with some explanation about the respective contents see the alphabetical list in Table 1. E.g. `stdio.h` provide the declarations for the functions `printf` and `scanf`, so that the compiler can generate correct code when these functions are called, but it does not give the compiler the functions themselves. The actual functions will be combined into a program at the end of compilation, by the part of the compiler called the "linker".

There are many C compilers around, the `cc` being the default Sun compiler. The GNU C compiler `gcc` is popular and available for many platforms. PC users may also be familiar with the Borland `bcc` compiler.

All C compilers operate in essentially the same manner and share many common command line options. However, the best source of each compiler is through the online manual pages of the system used: e.g. in the case of UNIX `man cc`. For the sake of compactness we will simply refer to the `cc` compiler—other compilers can simply be substituted in place of `cc`.

To compile a program one simply invokes the command `cc`. The command must be followed by the name of the (C) program one wishes to compile. A number of compiler options can be specified also. We will not concern ourselves with most of these options. Thus, the basic compilation command is:

```
cc program.c
```

where `program.c` is the name of the file. If there are obvious errors in the program (such as mistypings, misspelling of one of the key words or omissions of a semi-colon), the compiler will detect and report them. There may, of course, still be logical errors that the compiler cannot detect. When the compiler has successfully digested the program, the compiled version, or "executable", is left in a file called `a.out`. Often it is more convenient to use the compiler option `-o` and filename in the compilation as in

```
cc -o program program.c
```

which puts the compiled program into the file `program` (or any file that is named following the `-o` argument) instead of putting it in the file `a.out`. If a program references library functions or functions defined in other source files the "link editor" combines these functions (with `main()`) to create an executable

[6] Parsing is the process of identifying the grammatical structure of the program and recognizing the role of each component.

file. Probably the most commonly used library is the math library (`<math.h>`). One has to link in this library explicitly if one wishes to use the maths functions by:

```
cc program.c -o program -lm
```

(`-lm` standing for "`l`ibrary of `m`athematical functions"). Many other libraries are linked in this fashion.

The next stage is to actually run the executable. To run an executable in UNIX, one simply types the name of the file containing it, in this case `program` (or `a.out`.) This executes the program, printing any results to the screen. At this stage there may be run-time errors, such as division by zero, or it may become evident that the program has produced incorrect output. If so, one must return to edit the program source, and recompile it, and run it again.

# F Tables

| Standard library header file | Explanation |
|---|---|
| <assert.h> | Contains macros and information for adding diagnostics that aid program debugging. |
| <ctype.h> | Contains prototypes for functions that test characters for certain properties, and prototypes for functions that can be used to convert lower case letters to upper case letters and vice versa. |
| <errno.h> | Defines macros that are useful for reporting error conditions. |
| <float.h> | Contains the floating point size limits of the system. |
| <limits.h> | Contains the integral size limits of the system. |
| <locale.h> | Contains function prototypes and other information that enables a program to be modified for the current locale on which it is running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times dollar amounts, and large numbers in different areas throughout the world. |
| <math.h> | Contains function prototypes for math library functions. |
| <setjmp.h> | Contains macros and information for adding diagnostics that aid program debugging. |
| <assert.h> | Contains function prototypes for functions that allow bypassing of the usual function call and return sequence. |
| <signal.h> | Contains function prototypes and macros to handle various conditions that may may arise during program execution. |
| <stdarg.h> | Defines macros for dealing with a list of arguments to a function whose number and types are unknown. |
| <stddef.h> | Contains common definitions of types used by C for performing certain calculations. |
| <stdio.h> | Contains function prototypes for the standard input/output library functions, and informations used by them. |
| <stdlib.h> | Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions. |
| <string.h> | Contains function prototypes for string processing functions. |
| <time.h> | Contains function prototypes and types for manipulating the time and date. |

Table 1: The standard libray header files

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | nul | soh | stx | etx | eot | enq | ack | bel | bs | ht |
| 1 | nl | vt | ff | cr | so | si | dle | dc1 | dc2 | dc3 |
| 2 | dc4 | nak | syn | etb | can | em | sub | esc | fs | gs |
| 3 | rs | us | sp | ! | ” | # | $ | % | & | ’ |
| 4 | ( | ) | * | + | , | − | . | / | 0 | 1 |
| 5 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; |
| 6 | < | = | > | ? | @ | A | B | C | D | E |
| 7 | F | G | H | I | J | K | L | M | N | O |
| 8 | P | Q | R | S | T | U | V | W | X | Y |
| 9 | Z | [ | \ | ] | ˆ | _ | ‘ | a | b | c |
| 10 | d | e | f | g | h | i | j | k | l | m |
| 11 | n | o | p | q | r | s | t | u | v | w |
| 12 | x | y | z | { | $\vert$ | } | ˜ | del | | |

Table 2: ASCII character set

| C type | Size (bytes) | Lower bound | Upper bound |
|---|---|---|---|
| `char` | 1 | −− | −− |
| `unsigned char` | 1 | 0 | 255 |
| `short int` | 2 | −32768 | 32767 |
| `unsigned short int` | 2 | 0 | 65536 |
| `(long) int` | 4 | $-2^{91}$ | $2^{91}-1$ |
| `float` | 4 | $-3.2 \times 10^{\pm 98}$ | $3.2 \times 10^{\pm 98}$ |
| `double` | 8 | $-1.7 \times 10^{\pm 908}$ | $1.7 \times 10^{\pm 908}$ |

Table 3: C data types

| Conversion specifier | Type | Result |
|---|---|---|
| `%c` | `char` | single character |
| `%i, %d` | `int` | decimal number |
| `%o` | `int` | octal number |
| `%x, %X` | `int` | hexadecimal number in lower/upper case notation |
| `%u` | `int` | unsigned integer |
| `%s` | `char *` | string |
| `%f` | `double, float` | floating-point number |
| `%e, %E` | `double, float` | floating-point number in exponential notation |
| `%g, %G` | `double, float` | floating-point number in `f` or `e, E` form whichever is most compact |
| `%p` | `type *` | memory location as hexadecimal integer |

Table 4: Conversion specifications

| Escape Sequence | Description |
|---|---|
| `\'` | Output the single quote (') character. |
| `\"` | Output the double quote (") character. |
| `\?` | Output the question mark (?) character. |
| `\\` | Output the backslash (\) character. |
| `\b` | Move the cursor back one position on the current line. |
| `\n` | Position the cursor at the beginning of the next line. |
| `\t` | Move the cursor to the next horizontal tab position. |
| `\v` | Move the cursor to the next vertical tab position. |
| `\r` | Position the cursor to the beginning of the current line; do not advance to the next line. |
| `\f` | Move the cursor to the start of the next logical page. |
| `\a` | Sound the system bell. |

Table 5: Common escape sequences

<table>
<tr><th>C operator</th><th>Associativity</th></tr>
<tr><td>() [] -> .</td><td>left to right</td></tr>
<tr><td>++ -- + - ! ~ (type) * & sizeof</td><td>right to left</td></tr>
<tr><td>* / %</td><td>left to right</td></tr>
<tr><td>+ -</td><td>left to right</td></tr>
<tr><td><< >></td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>left to right</td></tr>
<tr><td>== !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>||</td><td>left to right</td></tr>
<tr><td>?:</td><td>right to left</td></tr>
<tr><td>= += -= *= /+ %= &= ^= |= <<= >>=</td><td>right to left</td></tr>
<tr><td>,</td><td>left to right</td></tr>
</table>

Table 6: Operator precedence and associativity: the operators are shown in decreasing order of precedence from top to bottom

## References


[1] American National Standards Institute. *American National Standard for Information Systems — Programming Language C.* X3.159-1989.

[2] J.A. Bergstra and M.E. Loots. Empirical semantics for object-oriented programs. Technical Report, Lecture Notes, Programming Research Group, University of Amsterdam, 1999.

[3] J.A. Bergstra and M.E. Loots. Programs, interfaces and components. Technical Report, Lecture Notes, Programming Research Group, University of Amsterdam, 1999.

[4] J.A. Bergstra and M.E. Loots. Software mechanics for Java multi-threading. Technical Report P9906, Programming Research Group, University of Amsterdam, 1999.

[5] B.J. Cox and A.J. Novobilski. *Object-Oriented Programming: An Evolutionary Approach* (2nd edition). Addison-Wesly: Reading, Mass., 1986.

[6] H.M. Deitel and P.J. Deitel. *C: How to Program.* Prentice-Hall, Inc.,Upper Saddle River, NJ 07458, 1999.

[7] P. A. Darnell and P. E. Margolis. *C a Software Engineering Approach.* Springer, 1996.

[8] N.H. Gehani and W.D. Roome. *Concurrent C.* Silicon Press, Summit, NJ, 1989.

[9] S.C. Johnson and B.W. Kernighan. The Programming Language B. Computer Science Technical Report #8, AT&T Bell Laboratories, 1973.

[10] B.W. Kernighan and D. Ritchie. *The C Programming Language* (2nd edition). Prentice Hall, Englewood Cliffs, NJ, 1978.

[11] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[12] G. Nelson. *Systems Programming with Modula 3* (2nd edition). Prentice Hall, Englewood Cliffs, NJ, 1991.

[13] M. Richards and C. Whitbey-Strevens. *BCPL: The Language and its Compiler.* Cambridge University Press, Cambridge, 1979.

[14] B. Stroustrup. *The C++ Programming Language.* Addison-Wesly, Reading Mass., 1986.

[15] S. Summit. *C Programming FAQs: Frequently Asked Questions.* Addison-Wesly, Reading Mass., 1995.

[16] *C* Programming Guide.* Thinking Machines Corp., Cambridge Mass., 1990.

## Technical Reports of the Programming Research Group

**Note:** These reports can be obtained using the technical reports overview on our WWW site (`http://www.wins.uva.nl/research/prog/reports/`) or by anonymous ftp to `ftp.wins.uva.nl`, directory `pub/programming-research/reports/`.

[P0002] J.A. Bergstra, I. Bethke, and M.E. Loots. *A First Course on C Program Reading.*

[P0001] J.A. Bergstra, I. Bethke, and A. Ponse. *Basic Multi-Competence Programming.*

[P9906] J.A. Bergstra and M.E. Loots. *Software mechanics for Java multi-threading.*

[P9905] M.P.A. Sellink and C. Verhoef. *Towards Automated Modification of Legacy Assets.*

[P9904] M.P.A. Sellink and C. Verhoef. *Scaffolding for Software Renovation.*

[P9903] René Krikhaar, André Postma, M.P.A. Sellink, Marc Stroucken, and C. Verhoef. *A Two-phase Process for Software Architecture Improvement.*

[P9902] A. van Deursen, P. Klint, and C. Verhoef. *Research Issues in the Renovation of Legacy Systems.*

[P9901] M.P.A. Sellink and C. Verhoef. *Generation of Software Renovation Factories from Compilers.*

[P9811] J.A. Bergstra and M.E. Loots. *Program Algebra for Component Code.*

[P9810] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. *Restructuring of* COBOL/CICS *Legacy Systems.*

[P9809] B. Wedemeijer. *Introduction & Basic Tooling for* CASL *using* ASF+SDF.

[P9808] J.A. Bergstra and A. Ponse. *Two Recursive Generalizations of Iteration in Process Algebra.*

[P9807] M.P.A. Sellink and C. Verhoef. *An Architecture for Software Maintenance.*

[P9806] M. B. van der Zwaag. *Some Verifications in Process Algebra with Iota.*

[P9805] M.P.A. Sellink and C. Verhoef. *Development, Assessment, and Reengineering of Language Descriptions.*

[P9804] M.P.A. Sellink and C. Verhoef. *Native Patterns.*

[P9803] J.J. Brunekreef. *Annotated Algebraic Specification of the Syntax and Semantics of the Programming Language* Alma-0.

[P9802] W. Fokkink and C. Verhoef. *Conservative Extension in Positive/Negative Conditional Term Rewriting with Applications to Software Renovation Factories.*

[P9801] P. Klint and C. Verhoef. *Evolutionary Software Engineering: A Component-based Approach.*

[P9726] J.A. Bergstra and A. Ponse. *Grid Protocol Specifications.*

[P9725] J.A. Bergstra and A. Ponse. *Process Algebra Primitives for File Transfer.*

[P9724] J.A. Bergstra and A. Ponse. *Process Algebra with Four-Valued Logic.*

[P9723] J.A. Bergstra and A. Ponse. *Kleene's Three-Valued Logic and Process Algebra.*

[P9722] J.A. Bergstra and A. Ponse. *Bochvar-McCarthy Logic and Process Algebra.*

[P9721] M.P.A. Sellink and C. Verhoef. *Reflections on the Evolution of COBOL.*

[P9720] M. van der Graaf. *A Specification of Box to HTML in ASF+SDF.*

[P9719] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.*

[P9718] M. de Jonge. *Reuse of ASF+SDF Specifications by means of Renaming.*

[P9717] A. Sellink (Ed.). *Participants Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications.*

[P9717] E. Visser. *From Context-free Grammars with Priorities to Character Class Grammars.*

[P9716] H.M. Sneed. *Dealing with the Dual Crisis — Year 2000 and Euro — What Reverse Engineering can do to Help.*

[P9715] W. Fokkink and C. Verhoef. *An SOS Message: Conservative Extension in Higher Order Positive/Negative Conditional Term Rewriting.*

[P9714] M. van den Brand, M.P.A. Sellink, and C. Verhoef. *Control Flow Normalization for COBOL/CICS Legacy Systems.*

[P9713] B. Diertens. *Simulation and Animation of Process Algebra Specifications.*

[P9712] V. Partington. *Implementation of an Imperative Programming Language with Backtracking.*

[P9711] L. Moonen. *A Generic Architecture for Data Flow Analysis to Support Reverse Engineering.*

[P9710] B. Luttik and E. Visser. *Specification ofRewriting Strategies.*

[P9709] J.A. Bergstra and M.P.A. Sellink. *An Arithmetical Module for Rationals and Reals.*

[P9708] E. Visser. *Character Classes.*

[P9707] E. Visser. *Scannerless Generalized-LR Parsing.*

[P9706] E. Visser. *A Family of Syntax Definition Formalisms.*

[P9705] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Generation of Components for Software Renovation Factories from Context-free Grammars.*

[P9704] P.A. Olivier. *Debugging Distributed Applications Using a Coordination Architecture.*

[P9703] H.P. Korver and M.P.A. Sellink. *A Formal Axiomatization for Alphabet Reasoning with Parametrized Processes.*

[P9702] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. *Reengineering COBOL Software Implies Specification of the Underlying Dialects.*

[P9701] E. Visser. *Polymorphic Syntax Definition.*

[P9618] M.G.J. van den Brand, P. Klint, and C. verhoef. *Re-engineering needs Generic Programming Language Technology.*

[P9617] P.I. Manuel. *ANSI Cobol III in SDF + an ASF Definition of a Y2K Tool.*

[P9616] P.H. Rodenburg. *A Complete System of Four-valued Logic.*

[P9615] S.P. Luttik and P.H. Rodenburg. *Transformations of Reduction Systems.*

[P9614] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Core Technologies for System Renovation.*

[P9613] L. Moonen. *Data Flow Analysis for Reverse Engineering.*

[P9612] J.A. Hillebrand. *Transforming an ASF+SDF Specification into a ToolBus Application.*

[P9611] M.P.A. Sellink. *On the conservativity of Leibniz Equality.*

[P9610] T.B. Dinesh and S.M. Üsküdarlı. *Specifying input and output of visual languages.*

[P9609] T.B. Dinesh and S.M. Üsküdarlı. *The VAS formalism in VASE.*

[P9608] J.A. Hillebrand. *A small language for the specification of Grid Protocols.*

[P9607] J.J. Brunekreef. *A transformation tool for pure Prolog programs: the algebraic specification.*

[P9606] E. Visser. *Solving type equations in multi-level specifications (preliminary version).*

[P9605] P.R. D'Argenio and C. Verhoef. *A general conservative extension theorem in process algebras with inequalities.*

[P9602b] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives (revised version of P9602).*

[P9604] E. Visser. *Multi-level specifications.*

[P9603] M.G.J. van den Brand, P. Klint, and C. Verhoef. *Reverse engineering and system renovation: an annotated bibliography.*

[P9602] J.A. Bergstra and M.P.A. Sellink. *Sequential data algebra primitives.*

[P9601] P.A. Olivier. *Embedded system simulation: testdriving the ToolBus.*

[P9512] J.J. Brunekreef. *TransLog, an interactive tool for transformation of logic programs.*

[P9511] J.A. Bergstra, J.A. Hillebrand, and A. Ponse. *Grid protocols based on synchronous communication: specification and correctness.*

[P9510] P.H. Rodenburg. *Termination and confluence in infinitary term rewriting.*

[P9509] J.A. Bergstra and Gh. Stefanescu. *Network algebra with demonic relation operators.*

[P9508] J.A. Bergstra, C.A. Middelburg, and Gh. Stefanescu. *Network algebra for synchronous and asynchronous dataflow.*

[P9507] E. Visser. *A case study in optimizing parsing schemata by disambiguation filters.*

[P9506] M.G.J. van den Brand and E. Visser. *Generation of formatters for context-free languages.*

[P9505] J.M.T. Romijn. *Automatic analysis of term rewriting systems: proving properties of term rewriting systems derived from* ASF+SDF *specifications.*

[P9504] M.G.J. van den Brand, A. van Deursen, T.B. Dinesh, J.F.Th. Kamperman, and E. Visser (editors). ASF+SDF*'95: a workshop on Generating Tools from Algebraic Specifications, May 11&12, 1995, CWI Amsterdam.*

[P9503] J.A. Bergstra and A. Ponse. *Frame-based process logics.*

[P9208c] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208b).*

[P9502] J.A. Bergstra and P. Klint. *The discrete time ToolBus.*

[P9501] J.A. Hillebrand and H.P. Korver. *A well-formedness checker for $\mu$CRL.*

[P9426] P. Klint and E. Visser. *Using filters for the disambiguation of context-free grammars.*

[P9425] B. Diertens and A. Ponse. *New features in PSF II: iteration and nesting.*

[P9424] M.A. Bezem and A. Ponse. *Two finite specifications of a queue.*

[P9423] J.J. van Wamel. *Process algebra with language matching.*

[P9422] R.N. Bol, L.H. Oei J.W.C. Koorn, and S.F.M. van Vlijmen. *Syntax and static semantics of the interlocking design and application language.*

[P9421] J.A. Bergstra and A. Ponse. *Frame algebra with synchronous communication.*

[P9420] M.G.J. van den Brand and E. Visser. *From Box to TeX: An algebraic approach to the construction of documentation tools.*

[P9419] J.C.M. Baeten, J.A. Bergstra, and Gh. Stefanescu. *Process algebra with feedback.*

[P9418] L.H. Oei. *Pruning the search tree of interlocking design and application language operational semantics.*

[P9417] B. Diertens. *New features in PSF I: interrupts, disrupts, and priorities.*

[P9416] S.M. Üsküdarlı. *Generating visual editors for formally specified languages.*

[P9415] S.F.M. van Vlijmen, P.N. Vriend, and A. van Waveren. *Control and data transfer in the distributed editor of the ASF+SDF meta-environment.*

[P9414] M.G.J. van den Brand and C. Groza. *The algebraic specification of annotated abstract syntax trees.*

[P9413] A. Ponse, C. Verhoef, and S.F.M. van Vlijmen (editors). *Workshop on Algebra of Communicating Processes May 16-17, 1994 Utrecht University.*

[P9218b] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing probabilistic processes: ACP with generative probabilities (revised version of P9218).*

[P9412] C. Groza. *An experiment in implementing process algebra specifications in a procedural language.*

[P9411] M.J. Koens and L.H. Oei. *A real time muCRL specification of a system for traffic regulation at signalized intersections.*

[P9410] J.J. van Wamel. *Inductive proofs with sets, and some applications in process algebra.*

[P9409] J.F. Groote and J.J. van Wamel. *Algebraic data types and induction in muCRL.*

[P9408] J.A. Bergstra and P. Klint. *The toolbus: a component interconnection architecture.*

[P9407] P.H. Rodenburg. *Module algebra for initial algebra semantics.*

[P9406] J.A. Bergstra, I. Bethke, and P.H. Rodenburg. *A propositional logic with 4 values: true, false, divergent and meaningless.*

[P9211b] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in real time process algebra (revised version of P9211).*

[P9405] J.C.M. Baeten and J.A. Bergstra. *Process algebra with partial choice.*

[P9314b] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with iteration and nesting (revised version of P9314).*

[P9404] J. van den Brink and W.O.D. Griffioen. *Formal semantics of discrete absolute timed interworkings.*

[P9403] J.A. Bergstra and Gh. Stefanescu. *Processes with multiple entries and exits modulo isomorphism and modulo bisimulation.*

[P9402] J.A. Bergstra and Ch. Stefanescu. *Bisimulation is two-way simulation.*

[P9401] J.C.M. Baeten and J.A. Bergstra. *Graph isomorphism models for non interleaving process algebra.*

[P9331] J.A. Hillebrand. *An algebraic specification of a manufacturing system with hierarchical control.*

[P9330] K.M.M. de Leeuw and H. van der Meer. *A turning grille from the ancestral castle of the Dutch Stadtholders.*

[P9329] J.J. Brunekreef. *Process specification in a UNITY format.*

[P9328] B. Diertens. *A simulator for PSF in PSF.*

[P9327] M.G.J. van den Brand. *Generation of language independent prettyprinters.*

[P9326] J.C.M. Baeten and J.A. Bergstra. *Non interleaving process algebra.*

[P9306b] J.A. Bergstra, A. Ponse, and J.J. van Wamel. *Process algebra with backtracking (revised version of P9306).*

[P9325] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra with infinitesimals.*

[P9324] J.J. Brunekreef, R.L.C. Koymans J.P. Katoen, and S. Mauw. *Design and analysis of dynamic leader election protocols in broadcast networks.*

[P9323] M. Kaart and I. Polak. *Het alternating bit protocol met time out in discrete tijd.*

[P9322] N.J. Drost. *Unification in an algebra with choice and sequential composition.*

[P9321] N.J. Drost. *Unification in an algebra with choice and action prefix.*

[P9320] K.M.M. de Leeuw and H. van der Meer. *Een roostergeheimschrift van Alexander baron van Spaen.*

[P9319] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with combinators.*

[P9318] I. Bethke and A. Ponse. *A car registration authority, a concise PSF-specification.*

[P9317] S.F.M. van Vlijmen and J.J. van Wamel. *A semantic approach to Protocold using process algebra.*

[P9316] J.J. Brunekreef and A. Ponse. *An algebraic specification of a model factory, part IV.*

[P9315] M.G.J. van den Brand. *Prettyprinting without losing comments.*

[P9314] J.A. Bergstra, I. Bethke, and A. Ponse. *Process algebra with iteration.*

[P9313] S.F.M. van Vlijmen and A. van Waveren. *Algebraic specification of a system for traffic regulation at signalized intersections.*

[P9312] J.W.C. Koorn and H.C.N. Bakker. *Building an editor from existing components: an exercise in software re-use.*

[P9311] I. Polak. *Specification of a bank account in PSF.*

[P9305b] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix (revised version of P9305).*

[P9310] A.S. Klusener, S.F.M. van Vlijmen, and A. van Waveren. *Service independent building blocks-I; concepts, examples and formal specifications.*

[P9309] J.C.M. Baeten, J.A. Bergstra, and R.N. Bol. *A real time process logic.*

[P9308] J.F.A.K. van Benthem and J.A. Bergstra. *Logic of transition systems.*

[P9307] A. Stins and A. Schoneveld. *Specification of a bank account with process algebra.*

[P9306] J.A. Bergstra, A. Ponse, and J.J. van Wamel. *Process algebra with backtracking.*

[P9305] J.C.M. Baeten and J.A. Bergstra. *On sequential composition, action prefixes and process prefix.*

[P9304] S.F.M. van Vlijmen and A. van Waveren. *On generating synchronous interworkings from PSF process traces.*

[P9303] A. Ponse and J.A. Verschuren. *An algebraic specification of a model factory, part III.*

[P9302] J.A. Bergstra and J.V. Tucker. *The data type variety of stack algebras.*

[P9301] J.J. van Wamel. *A library for PSF.*

[P9208b] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra (revised version of P9208).*

[P9222] J.W.C. Koorn. *Connecting semantic tools to a syntax-directed user-interface.*

[P9221] J. Blanco. *Definability with the state operator in process algebra.*

[P9220] A. van Waveren. *Specification of remote sensing mechanisms in real space process algebra.*

[P9219] I. Bethke and P.H. Rodenburg. *Equational Constructor Induction.*

[P9218] J.C.M. Baeten, J.A. Bergstra, and S.A. Smolka. *Axiomatizing Probabilistic Processes: ACP with Generative Probabilities.*

[P9217] A. Ponse. *Process algebra and dynamic logic.*

[P9005b] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra (revised version).*

[P9216] E.E. Polak. *An efficient implementation of branching bisimulation and distinguishing formulae.*

[P9215] J.A. Bergstra and J.V. Tucker. *Equational specifications, complete term rewriting systems, and computable and semicomputable algebras.*

[P9214] J. A. Hillebrand and A. Ponse. *An algebraic specification of a model factory, part II.*

[P9213] N.J. Drost. *Unification in the algebra of sets with union and empty set.*

[P9212] J.J. van Wamel. *A study of a one bit sliding window protocol in ACP.*

[P9211] J.A. Hillebrand. *The ABP and the CABP - a comparison of performances in realtime process algebra.*

[P9210] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. *Decidability of bisimulation equivalence for processes generating context-free languages.*

[P9209] S.F.M. van Vlijmen and A. van Waveren. *An algebraic specification of a model factory.*

[P9208] J.C.M. Baeten and J.A. Bergstra. *Discrete time process algebra.*

[P9207] K. de Leeuw and H. van der Meer. *A secret message in the archive of the last great pensionary of Holland.*

[P9206] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9205] J.J. van Wamel. *An algebraic verification of the concurrent alternating bit protocol.*

[P9204] C. Verhoef. *On induction principles.*

[P9203] J.A. Verschuren. *A simulator for mCRL in ASF+SDF.*

[P9202] J.W.C. Koorn. *A generic text and structure editor.*

[P9201] P.H. Rodenburg. *Interpolation in Equational Logic.*

[P9102b] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols (revised version of P9102).*

[P9112] A. Mateescu. *Recursive systems with auxiliary variables in Basic Process Algebra.*

[P9111] J.C.M. Baeten and J.A. Bergstra. *A survey of axiom systems for process algebras.*

[P9110] J.J. van Wamel. *A formal specification of three simple protocols.*

[P9109] J.J. Brunekreef. *A formal specification of two simple protocols for Local Area Networks.*

[P9108] J.J. Brunekreef. *A formal specification of the Amoeba Transaction Protocol.*

[P9107] G.J. Veltink. *The PSF toolkit.*

[P9106] J.C.M. Baeten and J.A. Bergstra. *Asynchronous communication in real space process algebra.*

[P9105] C. Verhoef. *An operator definition principle (for process algebras).*

[P9104] J.C.M. Baeten and J.A. Bergstra. *The state operator in real time process algebra.*

[P9103] H.P. Korver. *Computing distinguishing formulas for branching bisimulation.*

[P9102] J.J. Brunekreef. *A formal specification of three Sliding Window Protocols.*

[P9101] H.P. Korver. *The current state of bisimulation tools.*

[P9009] G.J. Veltink. *From PSF to TIL.*

[P9008] J.C.M. Baeten and J.A. Bergstra. *Process algebra with signals and conditions.*

[P9007] S. Mauw and Gy. Max. *A formal specification of the Ethernet protocol.*

[P9006] G.J. Akkerman and J.C.M. Baeten. *Term rewriting analysis in process algebra.*

[P9005] J.C.M. Baeten and J.A. Bergstra. *Real space process algebra.*

[P9004] N.J. Drost. *Algbraic formulations of trace theory.*

[P9002b] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero a object (revised version of P9002).*

[P8804b] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions (revised version of P8804).*

[P8910b] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (revised version of P8910).*

[P8906b] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD (revised version of P8906).*

[P9003] C. Verhoef. *On the register operator.*

[P8916b] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra (revised version of P8916).*

[P9002] J.C.M. Baeten and J.A. Bergstra. *Process algebra with zero object and non-determinacy.*

[P9001] L.W. Kuurman. *The jungle of process semantics.*

[P8916] J.C.M. Baeten and J.A. Bergstra. *Real time process algebra.*

[P8915] J.C. Mulder. *The inevitable coffee machine.*

[P8914] H. Jacobsson and S. Mauw. *A Token ring network in PSFd.*

[P8913] H.R. Walters. *Hybrid implementations of algebraic specifications.*

[P8912] S. Mauw and G.J. Veltink. *A Tool Interface Language for PSF.*

[P8911] P.H. Rodenburg. *Interpolation in conditional equational logic.*

[P8910] J.C.M. Baeten and J.A. Bergstra. *Design of a specification language by abstract syntax engineering (preliminary version).*

[P8909] J.A. Bergstra. *Algebra of states and transitions.*

[P8908] S. Mauw and F. Wiedijk. *Specification of the transit node in PSFd.*

[P8808b] J.A. Bergstra. *A mode transfer operator in process algebra (revised version of P8808).*

[P8907] J.A. Bergstra. *Kerninformatica en toekomst.*

[P8906] J.C.M. Baeten, S. Mauw J.A. Bergstra, and G.J. Veltink. *A process specification formalism based on static COLD.*

[P8815b] J.A. Bergstra. *Process algebra for synchronous communication and observation (revised version of P8815).*

[P8905] J.A. Bergstra and J.W. Klop. *BMACP.*

[P8904] J.A. Bergstra, S. Mauw, and F. Wiedijk. *Uniform algebraic specifications of finite sorts with equality.*

[P8903] J.A. Bergstra. *A representation of addition and deletion lists using module algebra.*

[P8902] R.P. Ogilvie. *Knowledge engineering vs. software engineering.*

[P8901] S. Mauw and G.J. Veltink. *An introduction to PSF.*

[P8826] J. Treur. *Design of modular and interactive knowledge-based systems.*

[P8825] H.W. Lenferink. *Local control of inference by means of meta-rules.*

[P8824] J.A. Bergstra and J.W. Klop. *Process theory based on bisimulation semantics.*

[P8823] J.A. Bergstra, J. Heering, and P. Klint. *Module Algebra (revised version).*

[P8822] J.L.M. Vrancken. *The algebraic specification of semicomputable data types (revised version).*

[P8821] J.C.M. Baeten and F.W. Vaandrager. *Specification and verification of a circuit in ACP (revised version).*

[P8820] J. Treur. *A logical framework for design processes.*

[P8819] H.K. Faber. *Strategische uitleg in een medisch expertsysteem.*

[P8818] H.K. Faber. *Uitleg en kennisrepresentatie in expertsystemen.*

[P8817] J. Treur. *Heuristic reasoning with incomplete knowledge.*

[P8816] J.A. Bergstra. *ACP with signals.*

[P8815] J.A. Bergstra. *Process algebra for synchronous communication and observation.*

[P8814] S. Mauw and G.J. Veltink. *A process specification formalism.*

[P8813] J. Treur. *Reasoning about partial models, actions and plans.*

[P8812] J. Treur. *On the use of reflection principles in modelling complex reasoning.*

[P8811] F. Wiedijk. *Voorlopig rapport over de specificatie-taal Perspect.*

[P8810] R.E. Swart. *The BCA Bull Course Adviser.*

[P8809] J. Treur. *Metakennis en meta-inferenties in expertsystemen.*

[P8808] J.A. Bergstra. *A mode transfer operator in process algebra.*

[P8807] J.L.M. Vrancken. *The implementation of process algebra specifications in POOL-T.*

[P8806] J. Treur. *Generic inference processes and their interactions in complex diagnostic tasks; a logical description.*

[P8805] J. Treur. *Completeness and definability in diagnostic expert systems.*

[P8804] J.A. Bergstra and J.V. Tucker. *The inescapable stack: An exercise in algebraic specification with total functions.*

[P8803] J.C.M. Baeten and F.W. Vaandrager. *Specification and verification of a circuit in ACP.*

[P8802] J.C.M. Baeten and J.A. Bergstra. *Recursive process definitions with the state operator.*

[P8801] F.R. Burggraaff and E. van der Meulen. *ASF Specification of a B-tree of order 1.*

[P8714] L.G. Bouma and H.R. Walters. *Implementing algebraic specifications.*

[P8713] W. Syski. *On a certain probabilistic approximation method for reasoning with uncertainty.*

[P8712] W. Syski and J. Treur. *Reasoning about uncertainty represented by meta-reasoning, the endorsements approach.*

[P8711] J. Treur. *Een logische analyse van diagnostische redeneerprocessen; redeneren met en over hypothesen.*

[P8710] J. Treur. *Volledigheid en definieerbaarheid in diagnostische redeneersystemen.*

[P8709] J.C.M. Baeten and J.A. Bergstra. *Global renaming operators in concrete process algebra (revised version).*

[P8708] S. Mauw. *Process algebra as a tool for the specification and verification of CIM-architectures.*

[P8707] J.H. Verhagen. *Expertsystemen bij de Nederlandse Spoorwegen.*

[P8706] J.C.M. Baeten, J.A. Bergstra, and J.L.M. Vrancken. *Processen en procesexpressies.*

[P8705] J.L.M. Vrancken. *The algebraic specification of semicomputable datatypes.*

[P8704] F. Wiedijk. *Termherschrijfsystemen in Prolog.*

[P8703] A.V. Hurkmans. *Een declaratieve en procedurele kennisrepresentatievorm voor kennissystemen, toegepast op NEXT.*

[P8702] M.R. Dasselaar. *Development of an expert system, from theory to practice.*

[P8701] R.A. Groenveld. *Verification of a sliding window protocol by means of process algebra.*