A Primer for Extended C

Comp 210

March 20, 1995

Contents

1 Extended C

C was originally developed in the early 1970s by Ken Thompson and Dennis Ritchie at Bell Laboratories as a portable systems programming language. At the time, nearly all operating systems were written in machine language for a particular computer. As a result, each different kind of computer had its own peculiar operating system. It was impossible to run an operating system written for one kind of computer on another without completely rewriting it.

Thompson and Ritchie wanted to develop a new operating system called Unix (for which they eventually won the Turing Award) that could easily be run on a variety of different computers. Consequently, they needed to write the operating system in higher level machine-independent notation and rely on a compiler (adapted for each kind of machine) to translate it to machine language. C eliminates the specific machine dependencies inherent in machine language, but still expresses computations in terms of generic machine operations on machine level data.

As C has grown more popular as a language for writing applications software like spreadsheets, word processors, compilers, and so on, its low-level origins have become a serious handicap. In an attempt to overcome these low-level origins while maintaining compatibility with existing C programs, Bjarne Stroustrup developed an extension of C called C++ to support a more abstract, less machine-oriented approach to programming. C++ is particularly aggressive about supporting the object-oriented programming (OOP) style.

Unfortunately, C++ is a large, complex language. Given its conflicting origins as a low-level systems programming language and an object-oriented applications language, this complexity is not surprising. Nevertheless, C++ makes a number of improvements to C. For this reason, we teach an extension of C consisting of ISO standard C augmented with C++'s input/output library, memory management, boolean datatype, constant variables, etc. "Restricted C++" would be a more accurate name than "Extended C": this dialect is a proper subset of ISO standard C++ and hence is supported by all ISO compliant C++ compilers.

Most modern C compilers comply with the ISO C standard. However, there are some older C compilers still in use that support a slightly different language than the ISO standard.

2 Good C and C++ References

The standard ISO C reference is *The C Programming Language*, by Brian Kernighan and Dennis Ritchie. Dennis Ritchie was a co-designer of the C language at Bell Laboratories. Brian Kernighan was one of the principal implementors of the early C programming environment produced at Bell Laboratories.

A comprehensive reference for C which many prefer over Kernighan and Ritchie is *C: A Reference Manual* by Samuel Harbison and Guy L. Steele Jr. Guy Steele was also a co-designer (with Gerald Sussman) of Scheme. The reference manual was the outgrowth of a production-quality C compiler project.

Like other living languages, C++ is still evolving. The definitive references on C++ are *The C++ Programming Language* by Bjarne Stroustrup and *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup.

You may also want to read *C Traps and Pitfalls* by Andrew Koenig. It contains many useful tips on how to avoid common mistakes in C.

3 Getting Acquainted with C

While most Scheme implementations support the interactive construction and modification of programs, most C implementations do not. (There exist C implementations that do and Scheme implementations that don't, but they are less common.) C programs are usually developed using the *batch programming* model. The programmer uses an editor (such as Emacs) to create a file or files containing a complete program, compiles the file(s) into an executable using the C compiler, and then executes the program.

For example, the following text

```
// This is a simple C program.

int max(int x, int y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  max(4, 5);
}
```

is a complete C program which computes the maximum of 4 and 5. The program defines a procedure `max` that maps two integer (`int` in C) arguments to an integer result. The body of the program is the body of the the procedure called `main`; it computes the maximum of the integers 4 and 5. Every C program has a procedure called `main`, which is called to start program execution.

3.1 Running a C Program

In Emacs, opening a file with a name ending in `.c` or `.cc` puts Emacs into C or C++ editing mode, respectively. In this mode, Emacs indents lines appropriately for C programs. Suppose that you created the preceding sample program in a file named max.cc. Before you can run this program, you must *compile* it by typing the command

```
g++ max.cc
```

in the Unix (xterm) window. This command translates the C program in max.cc into a machine language program and stores it in the file called a.out in your current directory. To run the compiled program, simply type its name: a.out (followed by pressing the Return key). If you want the compiled program to be created with a more descriptive name than a.out (such as max), you can add the option -o max to the command line.

```
g++ -o max max.cc
```

You should also use the -Wall option to g++, which turns on all warnings. Our final compilation command is

```
g++ -Wall -o max max.cc
```

We recommend using a C++ compiler such as g++ (the GNU C++ compiler) instead of a conventional C compiler (such as gcc or acc), even when compiling C programs, because C++ compilers perform more precise type checking.

You may be wondering what the program max does. It computes the value 5 and terminates *without producing any output.*

3.2 Simple Input/Output

In Scheme, the programming interface repeatedly reads expressions, evaluates them, and prints their values. In addition, Scheme programs automatically check the validity of arguments passed to primitive operations. If a bad argument is passed, Scheme aborts execution and prints an error message. In C, the compiler catches some errors during the compilation process, but compiled programs do not perform any error checking during program execution. If you make a mistake that the compiler does not catch, your C program can fail in unpredictable ways.

If we want the program max to print the value computed by the expression max(4,5), we have to pass the expression to a library procedure that prints data values. C++ provides a standard library of input and output routines called iostream.h. To include this library in your program, you must place the line

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

at the beginning of your program, before any of the library routines are called. The standard print routine is <<, which is similar in behavior to the Scheme procedure printf: it outputs an arbitrary datum (its right-hand argument) to a specified stream (its left-hand argument). The built-in stream cout represents the terminal screen. For example, the following statement prints the string “Input a number: ”:

```
cout << "Input a number: ";
```

The routine >> reads in data. The built-in stream cin represents the keyboard. For example, the following statement causes the program to pause until the user types in an integer. The integer is stored in the variable n.

```
cin >> n;
```

We can use these constructs modify our program to read in two numbers and print out the greater of the two.

```
// This is a simple C program that does input and output.

#include <iostream.h>

int max(int x, int y)
{
  if (x >= y)
    return x;
  else
    return y;
}

int main()
{
  int n1, n2;

  cout << "Input a number: ";
  cin >> n1;
  cout << "Input another number: ";
  cin >> n2;
  cout << "The maximum of the two numbers is " << max(n1,n2) << endl;
}
```

Note that uses of `<<` can be strung together to send more than one piece of data to an output stream. `endl` ends a line of input; you may think of it as the newline character in this context.

4 Names in C

C variable names are more restrictive than those of Scheme. C variable names must start with a letter and may contain only letters and digits; underscore (_) is classified as a letter. Hence,

```
a_car  XY  A  a  mid_pt  __
```

are all valid names, but

```
a-car  1XY  A?  a.1  mid-pt  **
```

are not.

Upper and lower case are different: `a` is not the same name as `A`. There is no upper bound on the length of a name.

4.1 Reserved Words in C

Scheme and C both reserve some names to identify language constructs. In Chez Scheme, the names

```
define        lambda   quote             define-structure
if            and      or                cond
local         set!     begin             case
let           letcc    letrec            let*
rec           do       eval              eval-when
unless        when     with              record-case
fluid-let     unquote  quasiquote        unquote-splicing
extend-syntax trace    trace-let         trace-lambda
syntax-case   time     critical-section  delay
```

are reserved. In C, the corresponding list is:

```
auto     break    case     char
const    continue default  do
double   else     enum     extern
float    for      goto     if
int      long     register return
short    signed   sizeof   static
struct   switch   typedef  union
unsigned void     volatile while
```

Moreover, the following additional words are reserved in C++:

```
asm    class delete   friend
inline new   operator overload
public this  virtual
```

5 Types in C

Like Scheme, C has a small collection of primitive data types. They include numeric types (see below) `char` (characters), `void` (the empty type with no values), pointer types (see section 12 on page 22), and array types (see section 9 on page 17). In addition, C includes a `struct` mechanism for constructing new types similar to the `define-structure` construct in Scheme; see section 11 (page 20) for details. The `stream` type used for input/output is a constructed type, not a primitive type.

C supports two varieties of numeric types: integral types including `int` (integers) and `unsigned int` (non-negative integers), and floating-point types including `float` (ordinary low-precision floating point numbers) and `double` (extended precision floating point numbers). Floating point arithmetic on engineering workstations has become so fast that almost all floating point arithmetic in C programs is done in extended precision. In fact, the C library of mathematical functions exclusively uses extended precision floating point numbers. In addition, all floating point constants in C (such as `0.`) have type `double`. Consequently, we *strongly* recommend using type `double` in preference to type `float`.

5.1 Type checking

You have already encountered types in Scheme. When you try to find the `cdr` of a symbol, or try to apply `+` to two lists, Scheme aborts execution and reports a *type error*. In Scheme, type errors are not detected until a program is run. Within a program, a given variable (binding occurrence of a name) does not have fixed type. During execution, a given variable can be bound to many different types of data at different times. For instance, this is legal Scheme:

```
(define x 5)                    ; x contains a number
(+ x 17)
(set! x (list 'first 'second))  ; now x contains a list
(cdr x)
```

In Scheme, each datum has a type, but variables do not have particular types; this is called *dynamic typing*.

C associates types with names: each variable can only hold values of one specific type, and a procedure can take only arguments of specified types and must produce a result of another particular type. Binding occurrences of names (both variables and functions) include type information indicating what values the name can stand for. The C compiler checks this information for consistency and reports an error *during the compilation process* if the information is not consistent. This process is called *static type-checking*.

If a variable or procedure is undeclared, C presumes that it has type `int`, but C++ issues a warning.

6 C Expressions

The basic building blocks of a C program are expressions and statements. In contrast to Scheme, which has only expressions (though some expressions change the program state and return an unspecified value), C makes a syntactic distinction between expressions and statements. For the moment, we will confine our attention to two kinds of C expressions: arithmetic expressions and boolean expressions.

6.1 Arithmetic Expressions

The syntax of arithmetic expressions in C is based on conventional mathematical notation. In particular, C uses infix notation for the standard binary operations $+, -, *, /$, and other primitive binary operations. In contrast to prefix notation (as in Scheme), infix notation is ambiguous. In the expression, `a - b + c`, which operation is performed first? In other words, is `a - b + c` equivalent to `(a - b) + c` or `a - (b + c)` ? C resolves this ambiguity by assigning *a level of precedence* to each operator. Unary minus (negation) has the highest precedence followed by multiplicative operators (`*` and `/`) and additive operators (`+` and `-`). The following figure lists the operations in decreasing order of precedence

`-` (unary negation)
`*` , `/`
`+` , `-`

When two operators of equal precedence appear consecutively in an arithmetic expression, the left operator takes precedence. The following examples illustrate how precedence governs the evaluation of arithmetic expressions:

```
  -x*y    ≡  (-x)*y
 x/y*z    ≡  (x/y)*z
a+b-c+d   ≡  ((a+b)-c)+d
a-b*c+d   ≡  (a-(b*c))+d
```

Of course, explicit use of parentheses overrides precedence.

Warning: you must either use parentheses or remember the precedence level of every operator in an expression. In expressions involving pointer variables (see section 12 on page 22), the precedence conventions become quite complicated. In this case, parentheses are essential for clarity.

By default, C performs arithmetic using machine quantities (such as 32-bit integers or 64-bit floating-point values). Such operations are extremely efficient, but are not always guaranteed to produce the correct answer. For instance, adding two large positive integers may result in a negative integer if the sum would have been too large for integers to represent on the hardware. Most operations on floating point values produce only approximate values (in any programming language)—though the approximations are often quite good. Additionally, in arithmetic operations involving both integers and floating point numbers, C converts the integers to floating point numbers, performs the specified arithmetic operation in floating point, and returns a floating point result. Such conversions generally lose some precision. C does not include built-in support for rational numbers or integers greater than `ULONG_MAX` (defined in library header file `limits.h`), which is often around $2^{64}-1$. However, widely available C libraries support bignums (large integers), rational numbers, complex numbers, and more.

Most arithmetic operators in C denote a family of functions rather than a single function; the particular function used in a given application depends on the types of the operands. Since the types of expressions are determined statically by the compiler, the compiler can always select a specific machine operation (usually implemented as a single machine instruction) to implement a given arithmetic operation. Beware that the division operator `/` on integers corresponds to the *quotient* operation in Scheme; it produces an integer result. For example,

$$\begin{aligned} \texttt{3/2} &\equiv \texttt{1} \\ \texttt{(5/2)*2} &\equiv \texttt{4} \end{aligned}$$

If either (or both) of the arguments to `/` is a floating-point number, then so is the result: `3.5/2` $\equiv$ `1.75`. The `%` operator computes the modulus: `7%5` $\equiv$ `2`.

The standard arithmetic operators are not the only functions that can be invoked in arithmetic expressions. Many more built-in operators exist, and C supports a large library of arithmetic functions in the library file `math.h`.[1] Invocations of these library procedures are written in conventional mathematical notation for function applications, namely *procedure-name*(*arg*$_1$, ... , *arg*$_n$). For example, the statement

```
d = sqrt(x*x + y*y);
```

where `d`, `x`, and `y` are variables of type `double`, sets `d` equal to $\sqrt{x^2+y^2}$ (up the accuracy of double-precision floating point arithmetic). All of the mathematical library procedures in C assume their inputs are double precision floating point numbers and produce double precision outputs.

6.1.1 Exercises

1. What results do you expect from the following program?

```
#include <iostream.h>
#include <math.h>

main()
{
 cout << "5-4+3 = " << 5-4+3 << endl;
 cout << "5-(4+3) = " << 5-(4+3) << endl;
 cout << "5/4 = " << 5/4 << endl;
 cout << "5/4. = " << 5/4. << endl;
 cout << "sqrt(2) =" << sqrt(2) << endl;
 cout << "sqrt(2.) =" << sqrt(2.) << endl;
}
```

Compile and run the program to check your predictions.

[1] To use the functions in `math.h`, the program must contain `#include <math.h>`.

2. Write a program that computes and prints $2^{15} - 1$ using the library function `exp2`. Look at the man page for `exp2` (do `man exp2`). Do you get the correct answer?

   On SunOS 4.x, the answer is "no", because the `<math.h>` header file fails to declare the type of the argument to `exp2`. The header file correctly notes that the return type is `double`, but doesn't say anything about the argument, so C presumes it has type `int`. (The reason for this problem probably has to do with desiring the header file to work in both C and C++, and so using the least common denominator. Or maybe it's just a bug.)

3. Fix the problem by inserting the declaration

   ```
   double exp2(double x);
   ```

   before procedure `main`.

6.2 Boolean Expressions

Like Scheme, C++ has a boolean datatype (called `bool`) with values `true` and `false`.[2] Like arithmetic expressions, boolean expressions are written in infix notation. The primitive boolean operators in C in order from high to low precedence are:

> `!` (not)
> (arithmetic operators go here)
> `<` (less), `<=` (less or equal), `>` (greater), `>=` (greater or equal)
> `==` (equal), `!=` (not equal)
> `&&` (and)
> `||` (or)

The not operator `!` has higher precedence than unary minus; the other boolean operators have lower precedence than the arithmetic operators, and associate from left to right. The expression `a + b < C && !e && f != g` is equivalent to `( ((a+b) < c) && (!e) ) && (f != g)`; of course, you would never write the former, but would add parentheses for clarity.

6.2.1 Exercises

1. What results do you expect from this program?

```
#include <iostream.h>
#include <math.h>

main()
{
 cout << "true = " << true << endl;
 cout << "false = " << false << endl;
 cout << "0 < 1 = " << (0 < 1) << endl;
 cout << "2 < 1 = " << (2 < 1) << endl;
 cout << "~false & ~true = " << (~false & ~true) << endl;
 cout << "~true = " << (~true) << endl;
}
```

[2] In practice, you can use any value in a boolean context; the "zero" value of any type represents false and "non-zero" values represent true. Hence, the expression 17 (considered as a boolean) is always true. Every C type includes some notion of "zero". However, it is better style to use explicit tests than to rely on this type conversion.

Compile and run the program to check your predictions.

The operators `&`, `|`, and `~` are special bit-oriented functions that you will encounter in courses that focus on low-level computing in C. Do not confuse them with the corresponding boolean operators `&&`, `||`, and `!`.

7 The Structure of C Programs

A C program consists of a sequence of `#include` statements, constant definitions, variable definitions, and procedure definitions. Comments are sequences of characters appearing to the right of `//`. With a few exceptions (such as inside string constants), extra spaces in C are ignored, just as they are in Scheme. The following template schematically describes the structure of a C program.

```
#include <iostream.h>  // include header file for stream io library
#include ...           // include other header files

const variable-definition  // define constants

// define global variables
variable-definition
...

// define program procedures
return-type procedure-name (parameter-declarations)
{
  body-of-procedure
}
...

// define main procedure
main(parameter-declarations)
{
  body-of-procedure
}
```

Procedures cannot be nested in C, but new local variables can be introduced anywhere; such variables exist until the end of the enclosing *block* (which is enclosed in curly braces `{}`), analogous to Scheme `local`.[3] Every program must contain a procedure called `main` (which may or may not take parameters), which is invoked to begin program execution.

7.1 Variable Definitions

A definition of a simple variable (a variable that is not of array or pointer type) has the form

type var `=` *expression*;

[3]Some programmers prefer to put all variable definitions at the beginnings of blocks, but others prefer to locate definitions as late as possible, just before the variable's first use.

where *type* is the name of a defined type, *var* is a name that is not reserved by C for some other purpose (such as `int` or `while`), and *expression* specifies the initial value for *var*.

The initialization clause

`=` *expression*

can be omitted, in which case the variable is left uninitialized and hence can have any value; you should set it to a sensible value before using it.[4] For example, the local definitions

```
int n = 22;
double mid;
```

define the variables `n` of type `int` and `mid` of type `double` and initialize `n` to 22.

To make program text more concise, C allows several variables of the same type to be defined together. In this case, a definition has the form

$$type\ \ var_1\ =\ expr_1,\ \ldots,\ var_n\ =\ expr_n;$$

For example, the global definition

```
double x, y = 1.0, z = -1.0;
```

defines `x`, `y`, and `z` with initial values `0.0`, `1.0`, and `-1.0`, respectively.

Warning: C does *not* support concise notation for parameter declarations; see below.

7.2 Procedure Definitions

A procedure definition in C is similar to a procedure definition in Scheme, but the syntax is quite different. The syntax of a C procedure definition is:

```
return-type procedure-name (parameter-declarations)
{
  body-of-procedure
}
```

where *body-of-procedure* is a sequence of C statements, possibly including local variable definitions, and *parameter-declarations* is a list of variable declarations (without terminating semicolons) separated by commas:

$type_1\ var,\ \ldots,\ type_n\ var_n$

Look at the sample procedure definition for `shrink` given in section 7.4 on page 12.

The simplest form of statement that can appear within a procedure is

`return` *expression*;

where *expression* is a C expression (of the specified *return-type*). To return a value from a C procedure, the body *must* execute a `return` statement. If the evaluation of the body of a procedure terminates without executing a `return` statement, then the procedure must have `void` type indicating that it does not return a value.

The following code defines a trivial procedure of no arguments that always returns the double-precision floating point number `0.0`.

[4] Strictly speaking, C initializes global (defined outside a procedure) variables to zero if the programmer doesn't specify an initializer; but relying on this behavior makes your programs harder to read, so you should specify an initializer even for global variables that you want to set to zero.

```
double zero()
{
  return 0.0;
}
```

7.3 Defining Constants

A line in a C program of the form

> `const` *typename varname* `=` *expression*;

defines *varname* as a constant of type *typename* with value *expression*. It behaves exactly like a variable of type *typename* initialized to the value *expression*, except that it cannot be modified. Of course, the value of *expression* must belong to the specified type.

For example, the text

```
const double PI = 3.141592653589793;
const double E  = 2.718281828459045;
```

defines `PI` and `E` as 16-digit approximations of *pi* and *e* (the natural logarithm base), respectively.

Some constants like `true` and `false` are already defined for you.

C also has another mechanism for defining constants by use of the `#define` preprocessor macro, but it has a number of disadvantages and you should never need to use it.

7.4 Static Type Checking

A C compiler rejects programs that do not respect the declared types of variables and procedures.

The following example demonstrates how C uses procedure and variable declarations to perform static type checking. A C compiler rejects this definition

```
int shrink(int val, int delta)
{
  if (val >= (INT_MIN+delta))
    return val-delta;
  else
    return "error";
}
```

because `shrink` attempts to return the string value `"error"` for some inputs, yet it is declared to return an integer. The constant `INT_MIN` is defined in the C library file `limits.h`.

8 Statements

There are two basic kinds of statements in C:

- *atomic* statements like expressions (including assignments and procedure calls) and `return` statements; and
- *composite* statements like blocks, `while` loops, `for` loops, and `if` statements.

The essential difference between these two kinds of statements is that composite statements are constructed from smaller statements while atomic statements are not.

All atomic statements end with a semicolon (`;`). Similarly, every local or global definition of a variable—but not a parameter declaration—ends with a semicolon (`;`).

8.1 Expressions as Statements

Any C expression can be converted to a statement simply by following it with a semicolon. In other words,

expression ;

is a statement. Many C expressions have side effects (change the program state), so it is common to use expressions as statements. Common examples include procedure calls and assignment expressions.

8.1.1 Assignments

Although assignments are almost always used as statements, they are technically expressions that are converted to statements by attaching a semicolon. An assignment is syntactically legal wherever an expression is legal. When an assignment is used as an expression, it does not end with a semicolon, and its value is the value of the right-hand side (which is also the value of the left-hand side, after the assignment). Most C programmers consider it bad style to embed assignments as subexpressions of larger expressions.

An assignment expression has the form

var `=` *expr*

where *var* is a variable and *expr* is an expression written in infix notation. For the purposes of this discussion, a variable is either a simple variable name (like `x`) or an expression (such as `a[i]`) that identifies a memory cell (a particular datum). If *var* is a simple variable name then the assignment above is equivalent to the Scheme expression

`(set!` *var* *expr′*`)`

where *expr′* is the expression *expr* written in Scheme *prefix* notation instead of C *infix* notation. The C notation for assignment is misleading; the `=` sign does not mean equality. A statement like

```
x = x+1;
```

looks mathematically nonsensical, but it has a simple interpretation in terms of changing state. The occurrence of `x` on the right hand side of the assignment means the value of `x` before it is changed. The equivalent Scheme code would read

```
(set! x (+ x 1))
```

Abbreviated Forms of Assignment C includes special syntax for updating the values of variables. For the binary operators $\circ \in \{+, -, *, /\}$ (and some other more obscure binary operators that we ignore in these notes), C supports the assignment expression

var $\circ$`=` *expr*

which is equivalent to the conventional assignment expression

var `=` *var* $\circ$ `(`*expr*`)`

except in the case when *var* is an expression with side-effects.[5] For instance, `x *= y` is shorthand for `x = x * y`.

In addition, the expressions

[5] For example, let *var* be the expression `a[f(i)+j]` where the evaluation of the function call `f(i)` increments the global variable `j`; then *var* `+= 1` only evaluates `f(i)` once, whereas *var* `=` *var* `+ 1` evaluates it twice.

var`++`

and

var`--`

are abbreviations for the expressions

var `+= 1`

and

var `-= 1`

respectively.

8.2 Composite Statements

Five important forms of composite statement in C are the block, the `if` statement, the `while` loop, the `for` loop, and the `do` loop.

The simplest form of composite statement is a *block* or *compound statement*; it consists of a sequence of statements (and local variable definitions) enclosed in braces:

```
{
  statement_1
  . . .
  statement_n
}
```

where any $statement_i$ can be a local variable definition (whose scope extends to the end of the block).

An example of a block is the following code

```
{
  area1 = .5 * (b - a) * (fa + fb);
  double mid = .5 * (a + b);
  double fmid = f(mid);
  area2 = .5 * ((mid - a) * (fa + fmid) + (b - mid) * (fmid + fb));
}
```

taken from a program that performs adaptive integration. The statements in a block are executed in sequence. Note that the body of a procedure is just a block.

8.2.1 if Statement

The C `if` statement has essentially the same meaning as the Scheme `if` statement, but the C `if` statement does not return a value. More precisely, the C `if` statement

```
if (expression)
    statement_1
else
    statement_2
```

is equivalent to the Scheme expression

```
(if expression'
    (begin statement1' (void))
    (begin statement2' (void)))
```

where *expression'* is the Scheme equivalent of the C expression *expression*, *statement*$_1$*'* is the Scheme equivalent of *statement*$_1$, and *statement*$_2$*'* is the Scheme equivalent of *statement*$_2$.

The `else` clause of the `if` statement is optional; it can be omitted yielding the following template:

```
if (expression)
    statement1
```

Section 8.2.2 (page 15) shows an `if` statement in the body of a `while` loop.

8.2.2 `while` Loop

The C `while` loop

```
while (expression)
    statement
```

first evaluates *expression*. If the result is true, it repeatedly evaluates *statement* followed by *expression* until *expression* is false. The C `while` loop has essentially the same meaning as a restricted form of the Scheme `local` help function. More precisely, the C `while` statement above is equivalent to the following Scheme expression:

```
(local
  [(define while
     (lambda ()
       (if expression
         (begin statement (while))
         (void))))]
  (while))
```

The function application `(void)` in the "false" branch of the `if` expression causes the computation to complete without returning a value.

The following `while` loop is taken from a root-finding program:

```
while (fabs(posx-negx) > delta)
  {
    mid = .5 * (negx + posx);
    if (f(mid) < 0)
      negx = mid;
    else
      posx = mid;
  }
```

This example also includes a block (the body of the `while` loop) and an `if` statement (described in section 8.2.1 on page 14).

8.2.3 `do` Loop

The C `do` loop corresponds to a different restricted form of the Scheme `local` help function. In particular, the C `do` statement

```
do
  statement
while (expression);
```

repeatedly evaluates *statement* followed by *expression* until *expression* is false. It is equivalent to the following Scheme expression:

```
(local
  [(define do-while
     (lambda ()
       (begin
         statement
         (if expression
           (do-while)
           (void)))))]
  (do-while))
```

The `while` loop of section 8.2.2 (page 15) could be rewritten as follows:

```
do
  {
    mid = .5 * (negx + posx);
    if (f(mid) < 0)
      negx = mid;
    else
      posx = mid;
  }
while (fabs(posx-negx) > delta)
```

Note, however, that this `do` loop is not equivalent to the `while` loop above, because the body of this `do` loop is always executed at least once.

8.2.4 `for` Loop

The C `for` loop is an abbreviation for a restricted form of C `while` loop. More precisely, the C `for` loop

```
for (init-expr; test-expr; increment-expr)
  statement
```

is equivalent to the following `while` loop:

```
init-expr;
while (test-expr)
{
  statement
  increment-expr;
}
```

The following `for` loop is taken from a program that computes the scalar product of two arrays.

```
for (i=0; i<N; i++)
  sum = sum + (num[i] * num[i]);
```

9 Arrays

One of the most important primitive data structures in C is the *array*. A one-dimensional array is simply an indexed collection of cells, like a Scheme vector, with three minor differences.

- The values stored in a C array all have the same type, which is is specified when the array is defined.
- The C compiler does not record the length of arrays. Hence, C does not support a function analogous to `vector-length`.
- C does not do any bounds checking to determine if an index value is legal. If you ask for a non-existent element of an array, your program will behave unpredictably (and may produce a *segmentation fault*).

9.1 Defining Arrays

A C array definition has the form:

```
int table_of_data[35];
```

This particular definition creates an array called `table_of_data`, containing a maximum of 35 elements, all of which are `int`egers. Note that array definitions are identical to definitions of "scalar" variables of the corresponding type except for the square brackets enclosing a size following the variable name.

When an array is defined, the values of its elements are unspecified.[6] We can specify initial values for an array using the syntax:

```
int primes[] = {2,3,5,7,11,13,17,19,23,29,31,37,41};
```

This program text defines the variable `primes` as an array of 13 specified integers. Notice that we can omit explicitly specifying the size of the array when we provide the initial values, since the compiler can determine the size by counting the number of initial values within braces. On the other hand, if we include the size of the array in its definition, we can conveniently initialize all elements of the array to constant values. When the number of initial values is less than the size of the array, C performs default initialization on the remaining elements of the array. For example, the following definition

```
double roots[20] = {10};
```

initializes element zero of `roots` to `10`, but leaves the initial values of the remaining elements unspecified.

9.2 Accessing Array Elements

We can extract the value of an element in an array by using the element's index. The expression `primes[i]` means the element in `primes` with index `i`. The corresponding operation in Scheme is `vector-ref` (unless `primes[i]` appears on the left-hand side of an assignment, in which case the corresponding Scheme operation is `vector-set`).

As in Scheme vectors, C array indices start at `0` rather than `1`, so the last element in an array of length n has index $n-1$. Hence

```
cout << primes[0] << primes[11];
```

prints out the first and twelfth elements, `2` and `31`, of the array `primes`.

[6] When an array is defined globally, all its non-explicitly-initialized elements are automatically set to zero, but it's poor style (and confusing to the reader) to rely on this property.

9.3 Array Assignment

To change the value of an array element, C uses exactly the same syntax to identify the element to be changed as it does to access the value of the element. Only the context of the array syntax is different. If the notation `a[i]` appears on the left-hand side of an assignment statement, then the element of array `a` with index `i` is the element to be changed. For example,

```
primes[0] = 1;
```

would change the value of `primes` to

```
{1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41}
```

9.4 Strings

In C, strings are simply arrays of the primitive type `char`. Scheme also supports the primitive type character (see pp. 124–128 of *The Scheme Programming Language*, by Kent Dybvig), but Scheme treats strings and vectors as distinct forms of data. In C, `char` constants are written by enclosing the specified character in single quotation marks. For example, the character `A` is written `'A'`.

By convention, every string in C ends with a *null* character, which is written `'\0'`. In C program text, a string constant consisting of n characters enclosed in double quotation marks (") actually represents an array of length $n + 1$ because by the compiler automatically places a null character at the end of the string. C programs can tell where a string ends by looking for the null character.

The following procedure counts the number of spaces in its argument (a string):

```
int no_of_spaces(char text[])
{
  int index = 0, count = 0;

  while (text[index] != '\0')
    {
      if (text[index] == ' ')
        count++;
      index++;
    }
  return count;
}
```

Every C implementation includes a library of procedures that implement basic string operations like string length, comparison, string copying, etc. You can learn about this library by executing the Unix command

```
man 3 string
```

or by finding the man page for a function in that library, e.g.,

```
man strlen
```

9.5 Exercises

1. Try writing your own version of `strlen`, which computes the length of a string (excluding the terminating null character). This procedure already exists in the string library, but it is a useful exercise. Use the following code as a template.

```
#include <iostream.h>

char const NULLCHAR = '\0';

// input: string terminated by NULLCHAR
// output: number of characters in x excluding the null character
int my_strlen(char str[])
{
  // YOUR CODE GOES HERE
}

main()
{
  cout << my_strlen("") << " " << strlen("") << endl;
  cout << my_strlen("A") << " " << strlen("A") << endl;
  cout << my_strlen("abc") << " " << strlen("abc") << endl;
  cout << my_strlen("This is a longer test")
       << " " << strlen("This is a longer test") << endl;
  cout << my_strlen("This is an even longer test")
       << " " << strlen("This is an even longer test") << endl;
}
```

Warning: remember that the equality operator in C is `==`.

Tip: in comparison operations involving a constant, if you put the constant on the left hand side of the comparison, then if you happen to use `=` instead of `==`, the compiler will warn you.

9.6 Multidimensional Arrays

This section can be omitted on first reading.

In general, an array definition in C has the form:

type varname`[`$size_1$`]` . . . `[`$size_n$`];`

where *type* is the name of a type (such as `int` or `double`) and $size_1$, ..., $size_n$ are constant expressions—that is, expressions that involve only constants (`const` variables and literals), so they can be evaluated at compile time. (The `new[]` operator creates arrays with bounds determined at runtime; see section 13 on page 27.) As in Scheme, the range of indices in each dimension i runs from 0 to $size_i - 1$ where $size_i$ is the corresponding array dimension. For example, the two-dimensional array defined by

```
int A[2][3];
```

has elements `A[0][0]`, `A[0][1]`, `A[0][2]`, `A[1][0]`, `A[1][1]`, and `A[1][2]`.

For an arbitrary n-dimensional array *varname*, the notation

varname`[`$expr_1$`]` ... `[`$expr_n$`]`

in a right-hand context means the value of the array element with indices $expr_1$, ..., $expr_n$. In assignment statements, the same notation is used to identify the array element to be updated. In other words, the expression

varname`[`$expr_1$`]` ... `[`$expr_n$`] =` $expr_0$

assigns the value of $expr_0$to the array element with indices $expr_1$, ..., $expr_n$. For example, the statement

```
A[0][1] = 52;
```

assigns the value 52 to the array element `A[0][1]`. The familiar notation

```
A[0,1] = 52;  // WRONG
```

from Pascal is wrong in C!

In the parameter declaration for an array passed as a parameter, every dimension except for the first one must declared by a constant expression. An array parameter declaration has the form:

type varname`[] [`$bound_2$`]` ... `[`$bound_n$`]`

where $bound_2$, ..., $bound_n$ are constant expressions. Alternately (and more flexibly), any array of the correct number of dimensions can be declared as

type `*` ... `*` *varname*

where there are as many asterisks as the array has dimensions.

For example, a function taking the argument `A` as a parameter must include a declaration in the header (technically called the *function prototype*) of the form

```
B[][3]
```

Of course the choice of `B` for a parameter name is arbitrary.

10 Expression Contexts

As we observed in the preceding subsection, the meaning of an array reference `a[i]` in C depends on the context. A *context* is simply a place in a C program where an expression can appear. In C, expression contexts are divided into two disjoint sets: *left-hand* contexts and *right-hand* contexts. A *left-hand* context is the left-hand side of an assignment. [7] All other contexts are *right-hand* contexts. Hence, in the assignment statement

```
a[i] = b[j] + c[k];
```

`a[i]` appears in a left-hand context while `i`, `j`, `k`, `b[j]`, `c[k]`, and `b[j] + c[k]` appear in right-hand contexts.

11 Structures

In Scheme, `define-structure` introduces new forms of data. In C, the analogous construct is a `struct` definition. In any context where C accepts variable definitions, C also accepts `struct` definitions. A `struct` definition has the syntax

`struct` *struct-name* `{` *field-declarations* `};`

[7]The `&` operator is discussed later in this document. In ISO C, there are two additional forms of left-hand context: the arguments to the prefix operators `&` and `sizeof`. The `&` operator returns the address of its argument—it's a way to obtain a pointer to an arbitrary object—and the `sizeof` operator returns the number of bytes taken up by its argument, which is useful when performing dynamic memory allocation via the C `malloc` library procedure.

(Note the semicolon following the close brace; in the subset of C taught here, you should see that combination only in `struct` declarations and array initializers (see section 9.1 on page 17); you don't ever need a semicolon after a close brace that ends a block.)

field-declarations is a list of *field* declarations terminated by semicolons; they look just like variable definitions:

$type_1$ $field_1$; . . . ; $type_n$ $field_n$;

Each *field* is a component variable of the structure, just as an array element is a component variable of the array containing it.

A C `struct` definition

`struct` *struct-name* `{` $type_1$ $field_1$ `;` ... `;` $type_n$ $field_n$ `; };`

roughly corresponds to the following structure definition in Scheme:

`(define-structure (`*struct-name* $field_1$ . . . $field_n$`))`

Like the Scheme definition, it does not define any new data. It simply defines a template for a composite data object consisting of the "variables" specified in the *field-declarations* and introduces selectors and mutators for accessing and updating the fields of a structure.

Following a `struct` definition, a variable v of the new `struct` type can be defined using the standard variable definition syntax:

struct-name v ;

where *struct-name* is interpreted as a type name. For example, a phone book entry could be defined as

```
struct Phone_book_entry
{
  char name[20];
  int number;
};
```

Given this `struct` definition, we could declare a phone book `white_pages` as an array with 1000 entries as follows:

```
Phone_book_entry white_pages[1000];
```

C uses special syntax to extract the fields of a `struct`. Given a `struct` value s,

s.*field-name*

is the component variable within s identified by *field-name*. In a right-hand context (anywhere but the left-hand side of an assignment statement, this expression is analogous to the Scheme expression

(*struct-name*-*field-name* s)

On the left-hand side of an assignment statement, it identifies field *field-name* of struct s as the variable to be changed by the assignment.

For example, assume `pbe` is a variable of type `Phone_book_entry` with the `name` field `"Comp"` and the `number` field `210`. Then `pbe.number` has the value `210`. When `pbe.number` is used on the left-hand side of an assignment statement, it identifies the `number` field within `pbe` as the variable to be changed. Hence, the assignment

```
pbe.number = 2643
```

changes the value of the `number` field of `pbe` (which is accessed via `pbe.number`) to `2643`.

11.1 Exercises

1. Complete the following template for a C program to look up numbers in a phone book represented as an array of `struct Phone_book_entry`. Fill in the body of the procedure `search`, which looks for the phone book entry matching a given name. The library procedure `strcmp` declared in the header file `<string.h>` performs comparison on strings.

```
#include <iostream.h>
#include <string.h>

struct Phone_book_entry
{
  char name[20];
  int number;
};

const int Comp_pbsize = 5;

Phone_book_entry Comp_pb[Comp_pbsize] =
{
  {"Corky", 6042},
  {"Matthias", 3815},
  {"Mark", 3814},
  {"Mike", 3833}
  {"Moshe", 5977},
};

// inputs:  pb, a phone book
//          key, name to be found
// output:  the number for person with name given by key;
//          0 if no match is found
int lookup(Phone_book_entry pb[], int pbsize, char key[20])
{
  // YOUR CODE GOES HERE
}

main()
{
  cout << "Corky's number is " << lookup(Comp_pb, Comp_pbsize, "Corky") << endl;
  cout << "Mike's number is " << lookup(Comp_pb, Comp_pbsize, "Alex") << endl;
  cout << "Moshe's number is " << lookup(Comp_pb, Comp_pbsize, "Moshe") << endl;
  cout << "Joe's number is " << lookup(Comp_pb, Comp_pbsize, "Joe") << endl;
}
```

12 Pointers

Pointers are memory addresses. Inside the machine, an address is simply an integer in the range from 0 to $n-1$ where n is the number of memory cells. Since the addresses of variables are determined by the C compiler, the specific numeric value of a pointer is generally not very interesting or informative, but there are various ways available in C to interpret pointers as integers. In fact, many C compilers will let you store an integer value in a pointer variable without complaint.

In Scheme, pointers are hidden. In fact, every data value in Scheme is represented by a pointer to a chunk of memory holding the value.

In C, pointers are explicit. "Pointer to `int`" is a distinct type from `int`. Given a pointer `p` to a value in memory, C provides an explicit operation for obtaining the value; it is called *dereferencing*. The dereferencing operation is denoted by the prefix operator `*`. For example, if `p` is a pointer to an `int`, then `*p` is the value of that integer. In contrast, `p` is the *address* of the memory word containing the integer.

Beware of the fact that the `*` operator has low precedence. As a general rule, it is a good idea to use parentheses to indicate the argument of the `*` operator. For example, assume we want to print element `i` of the array pointed by `p`. The statement

```
cout << *p[i];
```

does not do what we want in this case! It extracts element `i` of the array `p` and dereferences it assuming that the element is a pointer. Instead, use

```
cout << (*p)[i];
```

which dereferences the pointer `p` and extracts the element `i` from the specified array.

Note that the rules for evaluating expressions in C are more complicated than in Scheme. All variables identify addresses memory cells, but C automatically extracts the value[8] of a variable (the contents of the corresponding chunk of memory), unless the variable appears in a *left-hand context*—the left-hand side of an assignment, including abbreviated assignments like `x++`.

12.1 The Null Pointer

Each pointer is always in one of three states: valid, invalid, or null. A valid pointer points to data of the correct type. An invalid pointer points at some arbitrary memory, but cannot be distinguished from a valid pointer; the programmer must keep track of the difference. (Defining a pointer without supplying a value creates an invalid pointer; deleting the memory to which a pointer points changes that pointer from a valid to an invalid pointer.) An invalid pointer should never be used in any way, except possibly to be given a valid value (or to be set to `NULL`). A null pointer has the value `NULL`; it does not point to any data and should not be dereferenced; it is distinct from every valid pointer. The null pointer plays a role similar to the empty list `null` (or `'()`) in Scheme. The library header file `stddef.h` defines the identifier `NULL`.

12.2 Pointers to Structures

In the special case where a pointer points to a structure, C provides special syntax for accessing the fields of the target structure. Let `p` be a pointer to a `Phone_book_entry`. To refer to the number field of the structure `(*p)`, we can simply write `p->number` instead of `(*p).number`. The operator `->` is identical to first dereferencing, then using the `.` field extraction operator. This abbreviation simplifies reading and writing such field references and can be used in both left-hand and right-hand contexts.

---

[8] As noted below, C does not support the notion of an array value. Consequently, it uses the address of (the first element of) the array as the "value" of an array variable.

12.3 Syntax of Pointer Declarations

The type of a pointer to a simple type `t` is written as `t *` or as `t*` (C ignores the spaces on either side of `*`). For instance, we define a variable `pc` of type pointer to a character thus:

```
char * pc;
```

and the program text

```
int * p = NULL;
```

declares `p` as a pointer to an object of type `int`; the initial value of `p` is `NULL`: it does not point to anything.

We recommend avoiding multiple variable declarations for non-simple types. The variable definition

```
int * x,y;
```

defines `x` as a pointer to `int` and `y` as an `int` (*not* a pointer to an `int`); neither variable is initialized.

That is all that you will need to know about pointer declarations for most of the programming that you ever do. This section and the following ones discuss the (complicated) syntax for complicated types; the relationship between pointers and arrays (which C treats as effectively the same, though your programs will be clearer if you always access arrays using array syntax and pointers using pointer syntax); address arithmetic (useful when accessing arrays via pointer syntax); and more on array subscripting via pointers. You can safely skip all of this material on first (and even second) reading.

The syntax for defining pointer variables in C is governed by a straightforward rule that works for all composite types including pointer types, array types, and function types. To generate the syntax required to define a variable x of composite (*e.g.*, pointer) type T, perform the following three steps:

1. Construct a *sample* expression that produces a value of the simple type associated with T using the variable being defined, dummy variables (necessary for composite types involving arrays or functions), the dereferencing operator `*`, array subscripting, and function application.
2. Convert this expression to a type template tt by replacing all *dummy* variables in function applications by their types and all dummy variables used as subscripts by their bounds (if known) or nothing (if the bound is unknown).[9]
3. In the program, define the variable using the syntax T tt`;`.

The sample expression is not used in the program text; it is merely scratchwork used to derive the type template. The simple types include: `int`, `char`, `double`, `float`, `void`, all `struct` types, and a few more obscure primitive types (such as `unsigned int`) that we will not discuss.

For example, to declare a variable `z` of type pointer to `int`, we write

```
int *z;
```

because the expression `*z` will produce an `int` if `z` is a pointer to `int`. In simple cases like this one, the type template is identical to the sample expression.

Similarly, to define a variable `y` of type pointer to a one-dimensional array of `Phone_book_entry`, we construct a sample expression

[9] If the variable x being defined is an array, then its bound must be declared.

```
(*y)[i]
```

where i is a dummy `int` variable. Then we convert it to the type template

```
(*y)[]
```

replacing i by nothing because the size of the array is unknown. Finally we write the following definition of `y` in the program text:

```
Phone_book_entry (*y)[];
```

The parentheses surrounding `*y` are essential. The program text

```
Phone_book_entry *(y[]);  // WRONG
```

defines `y` as an array of pointers to `Phone_book_entry`. This program text is illegal, because it does not specify how large an array of pointers to allocate. The following modification, however, is legal:

```
Phone_book_entry *(y[100]);  // RIGHT
```

it defines `y` as an array of `100` pointers to `Phone_book_entry`.

As a final example, consider the definition of `a` as an array of 10 pointers to functions from `int` to `int`. A sample expression for `a` is

```
(*(a[i]))(j)
```

where `i` and `j` are dummy variables. The corresponding type template is

```
(*(a[10]))(int)
```

where 10 specifies the size of the array `a`. Hence, the definition of `a` has the form:

```
int (*(a[10]))(int);
```

Since array subscripting has precedence in program syntax over pointer dereferencing, the definition may be written more compactly as:

```
int (*a[10])(int);
```

12.4 Coercions Between Pointers and Arrays

In C, an array definition

type *varname*`[`*bound*$_1$`] . . . [`*bound*$_n$`];`

allocates space for the specified array and binds *varname* to the *base* (beginning) address of the allocated space—that is, the address of the first element. Arrays in C are peculiar, because they do not have values like other variables. Given the definitions

```
int i = 1, j = 2;
int a[4] = {4,3,2,1};
int b[4] = {0,0,0,0};
```

the statement

```
j = i;
```

assigns the value `1` to `j`. C interprets the occurrence of `i` on the right-hand side of the assignment statement as the *value* `1`. In contrast, the statement

```
b = a;
```

is illegal, because C arrays cannot be lvalues—cannot appear on the left-hand side of an assignment.

When an array name like `a` appears in a right-hand context, C interprets the name as a pointer to the *base* (the first element) of the specified array. The pointer has type t`*` where t is the element type of the array. [10]

12.5 Address Arithmetic

In C, pointers are frequently used to traverse arrays. For example, given a array `nums` and pointer variable `p`,

```
int nums[10] = {0,1,2,3,4,5,6,7,8,9};
int *p;
```

the assignment

```
p = nums;
```

binds `p` to the address of element zero of `nums`. Assume that `nums` is stored starting at location `11220` in the computer. Then `p` would be bound to the address `11220`. The statement

```
cout << p;
```

prints the value `11220`. On the other hand, the statement

```
cout << *p;
```

prints the value of `nums[0]`, which is `0`.

To "move" the pointer `p` so that it points to the next number in the array, we simply increment it. The assignment

```
p = p+1;    // equivalently, p++
```

increments the value of `p` by the size of one array element (which is 4 for an `int` on a SPARC-station). Of course, the same statement could have been written:

```
p++;
```

If we increment `p` once, the statement

```
cout << p;
```

prints the value `11224`. Similarly,

```
cout << *p;
```

prints the value of `line[1]`, which is `1`.

In general, adding an integer k to a pointer p of type t yields a pointer to a cell of type t that is k cells beyond (preceding, if k is negative) the cell identified p.

[10] Many books on C claim that an array variable really is a pointer constant bound to the base address of the memory allocated for the array. This description is accurate for our dialect of C, but the ISO C operator `sizeof` can distinguish an array variable from the corresponding pointer. In all other ISO C contexts, array variables behave like constant pointers to the element type.

12.6 Array Subscripting by Pointer Arithmetic

For a one-dimensional array

```
int A[100];
```

the expression

A[*expr*]

is equivalent to the expression

*(A + *expr*)

The name `A` is interpreted as a pointer to element zero of the array.

In a higher dimensional array like

```
int B[100][50];
```

`B` is interpreted as a pointer to row zero. C treats `B` as an array of arrays; the element type for `B` is an array of 50 integers. Hence, `B[12]` is a pointer to the base of the 13th row of `B`. It is equivalent to

```
*(B + 12)
```

Moreover,

```
*(*(B + 12)+ 13)
```

is equivalent to

```
B[12][13]
```

Subscripting is much clearer notation for addressing arrays than pointer arithmetic. Chapter 5 of the *The C Programming Language* by Kernighan and Ritchie discusses the relationship between array subscripting and pointer arithmetic in more detail.

13 Dynamic Memory Allocation

Many primitive operations in Scheme allocate memory (the most obvious one is `cons`, but others do as well). This allocation is generally hidden from the programmer because Scheme takes complete responsibility for managing memory. In C and C++, memory management is controlled by the programmer.

ISO standard C uses the library procedures `malloc` and `free`, in conjunction with the special operation `sizeof`, to allocate and free memory. The programmer specifies exactly how much memory to allocate and performs explicit type conversions.

In contrast, C++ supports higher-level *primitive* operations called `new` and `delete` for allocating and freeing memory. The expression

`new` *t*

allocates an object of type *t* and returns a pointer to the allocated object. The allocated object is not initialized.

Similarly, given a pointer `v` to an object allocated via `new`, the statement

```
delete v;
```

frees the space occupied by the object pointed to by `v`. For arrays, the expression

new t[$size$]

allocates (and returns a pointer to) an array; the memory is reclaimed by the statement

```
delete[] v;
```

which frees the array object pointed to by `v`.

It is very important to match up calls to the scalar form of `new` with calls to the scalar form of `delete` and calls to the array form of `new` with calls to `delete[]`. C++ does not check for correctness because pointers to individual objects are indistinguishable from pointers to arrays of these objects; the programmer must keep track of which is which.

For example, given the variable declaration

```
int * intPtr;
```

the code block

```
intPtr = new int;
*intPtr = 1994;
```

allocates an `int` object, returns its address, assigns that address to `intPtr`, and stores the value 1994 in the allocated `int` object. Similarly, given the variable definition

```
Phone_book_entry *(white_pages[100]);
```

the expression

```
white_pages[0] = new Phone_book_entry
```

allocates an object of type `Phone_book_entry`, returns its address, and assigns that address to the array element `white_pages[0]`.

If the type t of the object to be allocated is composite, we can generate the syntax for t using the following two step process:

1. Write the type template tt required to declare a variable `x` of type t using the familiar syntax:

   T tt;

   where T is the base type corresponding to tt.
2. Replace `x` in tt by the base type T to produce an "anonymous" type template tt'.

The anonymous type template tt' is the syntax for t accepted by the `new` operator.

For example, the expression

```
new char[10]
```

allocates an array of 10 characters. Similarly, given an `int` variable `N`, the expression

```
new Phone_book_entry[N]
```

allocates an array containing `N` elements of type `Phone_book_entry`.