Procedural programming

C language

Basic instructions

Instructions which can be executed by a computer

(RiSC-16)

Working in this environment id HARD:

Boilerplate

The following table describes the different instruction operations.

Mnemonic	Name and Format	Opcode (binary)	Assembly Format	Action
add	Add RRR-type	000	add rA, rB, rC	Add contents of **regB** with **regC**, store result in **regA**.
addi	Add Immediate RRI-type	001	addi rA, rB, imm	Add contents of **regB** with **imm**, store result in **regA**.
nand	Nand RRR-type	010	nand rA, rB, rC	Nand contents of **regB** with **regC**, store results in **regA**.
lui	Load Upper Immediate RI-type	011	lui rA, imm	Place the 10 ten bits of the 16-bit **imm** into the 10 ten bits of **regA**, setting the bottom 6 bits of **regA** to zero.
sw	Store Word RRI-type	101	sw rA, rB, imm	Store value from **regA** into memory. Memory address is formed by adding **imm** with contents of **regB**.
lw	Load Word RRI-type	100	lw rA, rB, imm	Load value from memory into **regA**. Memory address is formed by adding **imm** with contents of **regB**.
beq	Branch If Equal RRI-type	110	beq rA, rB, imm	If the contents of **regA** and **regB** are the same, branch to the address PC+1+**imm**, where PC is the address of the beq instruction.
jalr	Jump And Link Register RRI-type	111	jalr rA, rB	Branch to the address in **regB**. Store PC+1 into **regA**, where PC is the address of the jalr instruction.

Boilerplate

Coming from newspaper printing…

In computer programming, **boilerplate code** or just **boilerplate** refers to sections of code that have to be included in many places with little or no alteration.

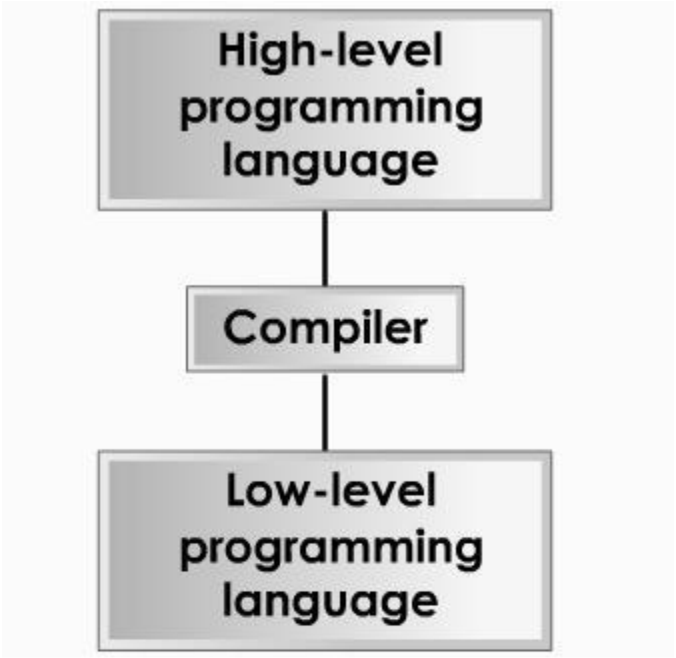
Boilerplates are

- take the most of development resources
- good sources of bugs

High level programming languages

- Better understood by human
- Preimplemented boilerplates → **API**
- With good language specification, platform independent source code can be created
- Programming paradigms

C, C++, C#, Java, Pascal, Delphi, Basic



Paradigm

In science and philosophy, a **paradigm** is a distinct set of concepts or thought patterns, including

theories, research methods, postulates, and standards

for what **constitutes legitimate contributions to a field**.

Programming paradigms

A way to classify **programming languages** based on their **features**

Imperative

the programmer instructs the machine how to change its state

- **structured** – structured flow into blocks and control statements
- **procedural** - groups instructions into procedures
- **object-oriented** - groups instructions together with the part of the state they operate on

Declarative

the programmer merely declares properties of the desired result, but not how to compute it

- **functional** - the desired result is declared as the value of a series of function applications
- **logic** - the desired result is declared as the answer to a question about a system of facts

Structured programming

Aimed at improving the **clarity**, ***quality***, and **development time** of a computer program by making extensive use of the structured control flow constructs of

- *code blocks*
- *selection*
- *repetition*
- subroutines

Procedural programming

Derived from structured programming, based upon the concept of the *procedure calls*. **Procedures**, also known as subroutines or functions, simply contain a **series of computational** steps to be carried out.

Any given procedure might be called at any point during a program's execution, including other procedures or itself.

Procedures are

- stateless (by definition)
- but can work on global variables (can maintain state)
- make only data transformation

Basic language components – C

- Variables, types, type casts
- User defined types
- Arrays, strings, constants
- Operators, precedence, overload
- Control statements
- Code modules and parameters
- Dynamic memory management, data references
- Function pointrers – method references

Variable

Variable

- data storage unit of memory
- can be referred by its name
- stores a data with predefined type
- the stored data is a subject of change

```
int counter;
counter = 1;
counter = 2;
```

Primitive types

Data types specify the amount of allocated memory for a variable, and the method of its usage.

The C language contains numeric data types which support basic arithmetical operations, and it provides possibility to create user defined types.

Primitive types: `char, int, float, double, boolean (unsigned char)`

Modifiers: `signed/unsigned, short/long`

Primitive types – example

How much space has to be allocated and how it should be threated to store age and speed data.

```
unsigned char age = 12;        // 0..255 (1 byte, as is)
float speed = 826.8 * 3.6; // exact speed in m/sec
     (4 byte, 32 bit: 0-22 mantissa, 23-30 exponent, 31 sign)
              speed = (-1)^s*m*2^(e-127)
```

Type casts

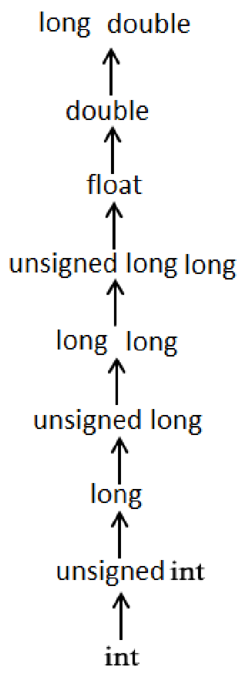
When variables of multiple types occure in an expression, type conversion is required.

This is done by the compiler in the following steps

1. Integer promotion

   Types smaller than **`int`** are casted to **`int`**

2. Types are converted to the highest occuring element of the hierarch



Type casts – automatic up

```
int  i = 17;
char c = 'c';                  // ASCII value of ’c’ is 99
float sum;                     // int and char to double
sum = i + c;                   // Value of sum : 116.000000
```

Type casts – automatic down

```
double  length = 33.3;
int boxLength;                                // Double to int

boxLength = length;                 // Value of boxLength : 33
                                                          // Automatic type
cast to less data
```

Type casts – questions

```
int d = 18 / 5;                  // Value of d: 3

int e = 'c' / 10;                // 'c' == 99, Value of e: 9

float q1 = 18 / 5;               // Value of q1: 3.0

float q2 = (float)18 / 5;        // Value of q2: 3.6
```

Custom types

Programmer can create custom types at compile time for

- structured use of logically connected data
- optimal use of storage space
- increasing source code readability

Enumeration type

- Custom type, created by the programmer at compile time
- Collection of named integer constants
- Goals:
  - Specify exact set of values of an integer variable
  - Increase code readability by using names instead of values
- Created by **`enum`** keyword
- Names have to be globally unique

Enumeration type

```
enum color { red, green, blue };
enum color favorite_color;

printf(”Please pick a color\n");
printf("1-red, 2-green, 3-blue:");
scanf("%d", &favorite_color);

switch (favorite_color)
{
      case red:
       printf(”Red selected");
       break;
  case green:
       printf("Green selected");
       break;
  case blue:
       printf("Blue selected");
       break;
  default:
       printf(”Wrong selection");
  }
```

Structs

- Custom type created by the programmer az compile time
- Collection of logically connected values of different type
- Structure of collection is set at compile time
- Structure of collection can not change at run time

Example of struct

```
typedef struct
        int price;
        int vatPercent;
} Product;
```

Name of created type

Data members of structure

Alias of created type

Create new variable:

```
Product newProduct;
```

Access data member:

```
newProduct.price = 951;
newProduct.vatPercent = 15
```

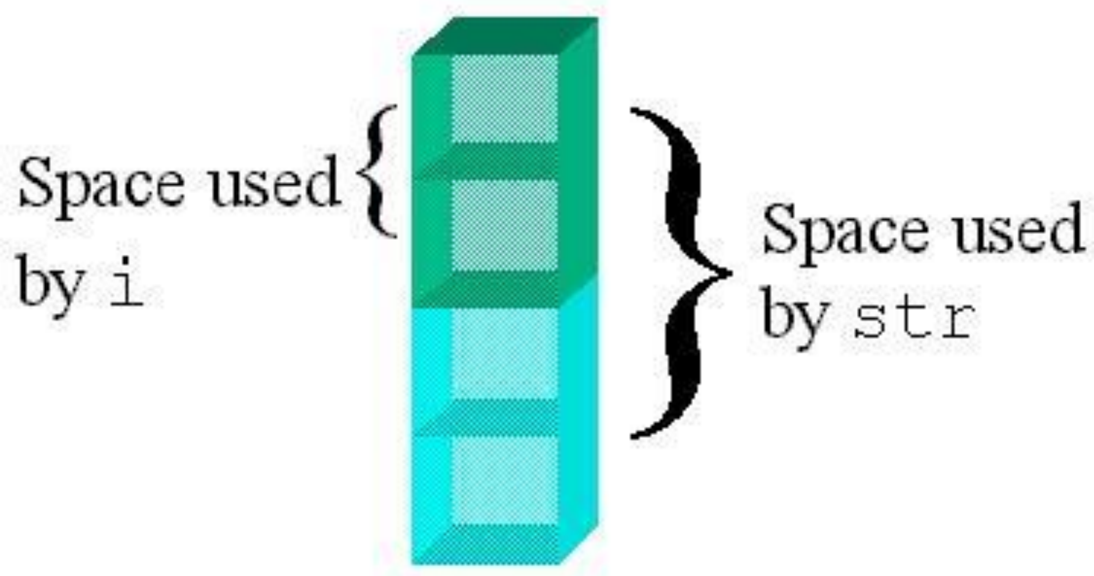
Union data storages

- Created at compile time, to store multiple type of data *exclusively*
  Only one member of struct is in use at a time
- Shares the allocated memory between possible stored types
- Allocates memory for the biggest storage type
- Minimizes the allocated memory
- Application requires high care

Example of union

```
union SomeUnion {
    int i;
    char str[4];
}
```

Allocated **4** bytes instead of **6**



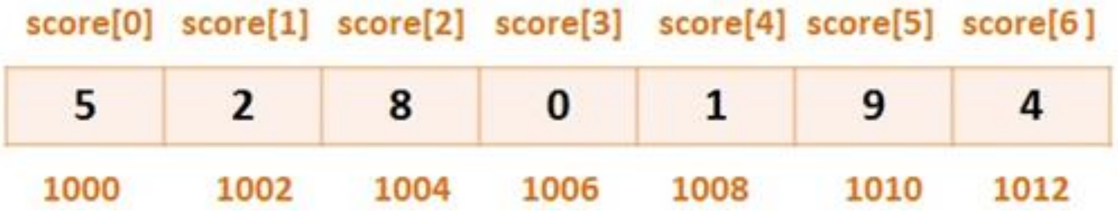
Arrays

- Collection of items of same type – type can not be changed
- Number of items set at compile time – can not be changed
- Items stored one after another in the memory
- Items can be modified independently
- Items accessed by **`indexer`** (index operator)

Example of arrays

Declaration of one dimensional array:

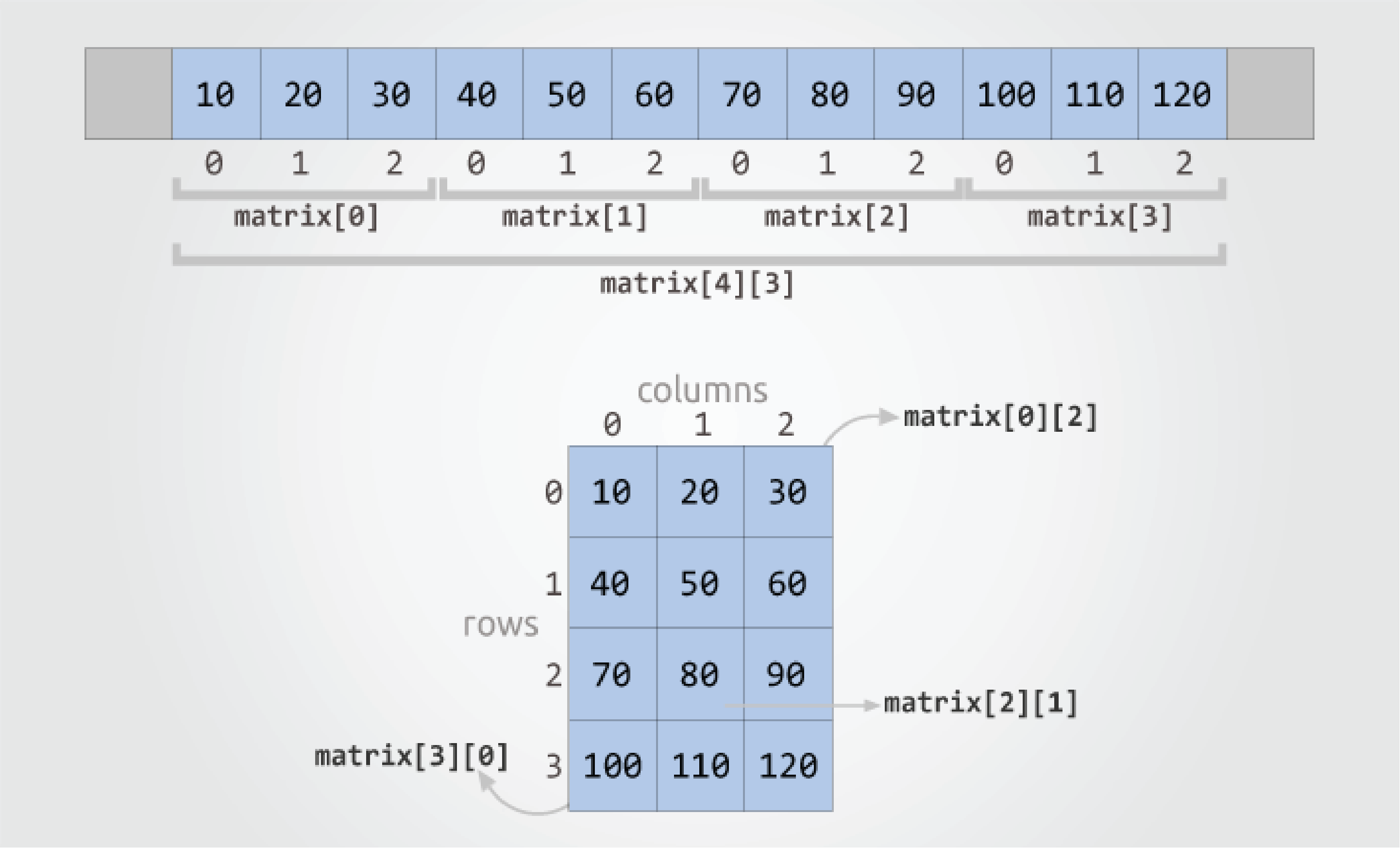
```
int score[7] = {5, 2, 8, 0, 1, 9, 4};

for(int i=0; i<7; i++) {
    score[i] += 10;
}
```



Example of array

Two dimensional array:

```
int matrix[4][3] = {
        {10, 20, 30},
        {40, 50, 60},
        {70, 80, 90},
        {100, 110, 120}
};
```



String type

- A built in type to store text data: ”Hello world”
- In C, string is a `″\0″` terminated `char` array (or `NULL`) egy karakter tömb (end of valid text is marked by a terminal `″\0″`)

```
char message[]=″Hello!″;
char message[7]={'H','e','l','l','o','!','\0'};
```

Literal constant

- Storage of a specified type (primitive or array)
- With referrable allocated memory area (lvalue)
- Can be string data (char array)

```
int literal = 19;
```

```
85              /* decimal int */
0213            /* octal */
0x4b            /* hexadecimal */
30u             /* unsigned int */
30l             /* long */
30ul            /* unsigned long */
"Hello world!"  /* string */
```

Constants

Typed data storage without referable memory storage (rvalue).

Creation:

- #define processor instruction

```
#define LENGTH 10
```

- const keyword

```
const int  WIDTH = 5;
```

usage

```
printf("value of area : %d", LENGTH * WIDTH);
```

Code block – scope

- Collection of logically coherent instructions
- Bounded by curly braces: `{ <instructions to execute> }`
- Variables defined in a blokk can be used only in that block
- A name space, a variable with the defined name exists only in it – scope
- Variable defined in a block hides variables defined outside ones

Scope of variable – local/global

- Declaring code block is the scope of the variable – variable is **local** in it
- In C source files variables can be declared outside of code blocks. In this case, the scope is the compilation unit – variable is **semiglobal** - global in compilation unit
- With `extern` modifier, scope of variable can be extended between compilation units. Scope of such a variable is all of the compilation units – this variable is **global**
- Variables global by default do not exist (can not exist - linking)

Operators

Operator is a symbol which tells the compiler to perform specific mathematical or logical function.

Operator types:

- Arithmetic
- Relational
- Assignment
- Bitwise
- Logical
- Misc (sizeof, ?:)

In C language operator can **not** be created for custom types

Operator precedence

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Use brackets!

Able to read!

Control statements

- Statetments to control the flow of instruction execution.
- Execution execution is independent from other instructions
- Can be nested in any level and any combination

Control statements - sequence

Series of sequentially executed statements.

Terminal of statements: ”;”

```
printf(”Hello”);
printf(”World”);
char *name = ”Tamas”;
printf(”I am %s”, name);
```

Control statements – selection

One/Two-way

- When nested in multiple levels, hard to read and follow
- Can be controlled by any boolean expression
- Conditions can contain intervals and
- `else` branch can be absent

Multiple-way

- Easy to follow with multiple choices
- Only primitive type can control
- Conditions are constant values
- Can have a `default` branch

Control statements – selection

Two-way condition:

```
if(<condition>) {
   <execute when condition is true>
}
else {
   <execute when condition is false>
}
```

Control statements – selection

Two-way condition example:

```
if(x>=0) {
   y=sqrt(x);
}
else {
   printf(”No square root for negative numbers”);
}
```

Control statements – selection

Multi-way selection:

```
switch(primitív kifejezés) {
   case <konstans kifejezés> :
      break;
   case <konstans kifejezés> :
   case <konstans kifejezés> :
      break;
   default:
}
```

Control statements – selection

Multi-way selection example:

```
switch(dice) {
    case 1:
    case 3:
    case 5:
        printf(”number is odd”);
        break;
    case 2:
    case 4:
    case 6:
        printf(”number is even”);
        break;
    default:
        printf(”Not a dice”);
}
```

Control statements – selection

How to choose:

- Execution has one or two ways
  - **`if`** with or without **`else`**
- Execution has more ways
  - Control independent , logical expressions: nested **`if`** statements
  - Control by constants: **`switch-case`**

Control statements - iteration

- Repetition of a code block is specified by a condition
- Modification of the control condition is required in the cycle body to finish the iteration

Control statements - iteration

`while`

The while statement evaluates a control expression before each execution of the loop body.

```
while(<control expression>) {
      <statements to execure, modification of control expression>
}
```

Control statements - iteration

`do-while`

The do-while statement evaluates the control expression after each execution of the loop body.

```
do {
      <statements to execure, modification of control expression
>
} while(<control statement>);
```

Control statements - iteration

`for`

The for statement evaluates three expressions and executes the loop body until the second controlling expression evaluates to false.

```
for(<init expression>;<control expression>;<modification expression>)
{
      <statements to execute>
}
```

```
for(int i=1;i<=10;i++) {
    printf(i);
}
```

while

```
int cycles;
boolean hasEmail = false;
for(; cycles>0&&!hasEmail; cycles--)
{
    printf(count);
    hasEmail = checkEmails();
}
```

Control statements - iteration

- **`while`** – runs only the control condition is true (possibly never)
- **`do-while`** – runs at least onece, than until the control contitions is true
- **`for`** – number of iteration is exactly known on run time

Function

- Separable part (module) of the executable task
- Collection of logically related instructions
- Can be referred by name
- Can have input parameter
- Have return value

Function declaration

```
<return type> <function name>([formal parameter list])
{
   <function body>
}
```

```
int square(int value) {
   return value*value;
}
```

Function parameters

Parameters declared in the formal parameter list can be referred by their name in the message body.

Formal parameters:

- Given in compile type

Actual parameters:

- Given in run time

Formal parameter

```
int square(int value) {
    return value*value;
}

int main(int argc, char *argv[]) {
    int length = 5;
    int area = square(length);
}
```

See later...

Actual parameter

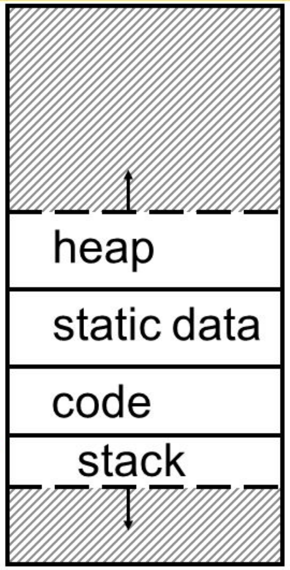
Primitive memory management

- Primitive types are stored in stack
- Declaration is in compile time
- Run time allocation is missing
- Scope + LIFO organization reduces stack fragmentation
- Unused local variables take the place until end of scope
- The stack fix, small size (single segment)
- Heap is unused

Intel 80x86 C Memory Management

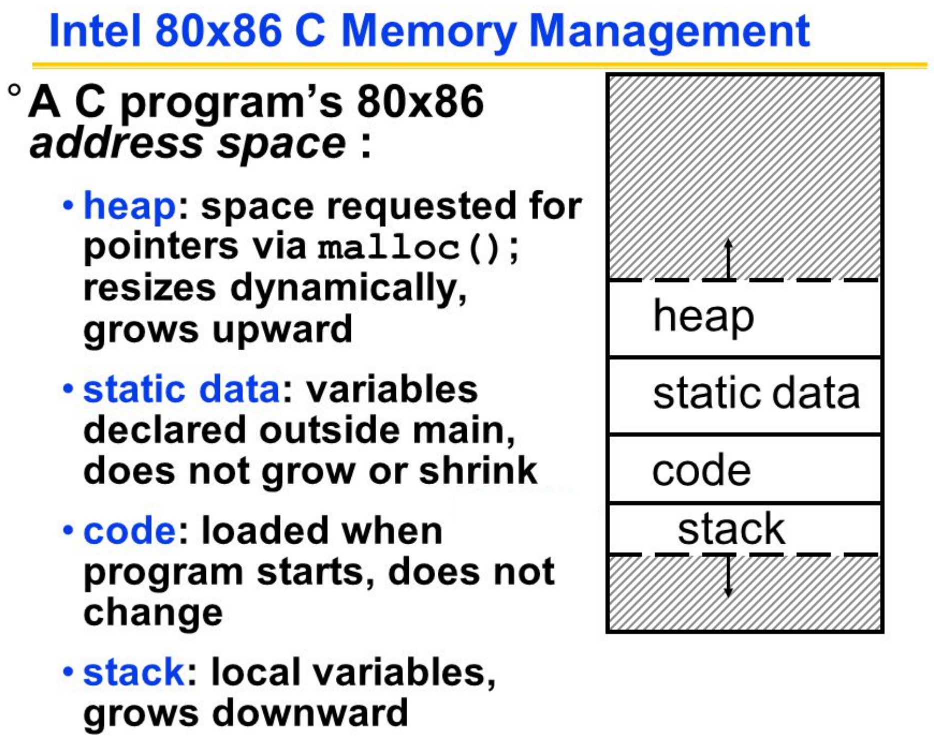
° **A C program's 80x86 *address space* :**

- **heap: space requested for pointers via `malloc()`; resizes dynamically, grows upward**
- **static data: variables declared outside main, does not grow or shrink**
- **code: loaded when program starts, does not change**
- **stack: local variables, grows downward**



Dynamic memory management

- Run time allocation
- Allocate storage in heap
- Allocated memory is releasable
- Heap is the all free memory



Dynamic memory management

Utilization of memory allocated in heap requires:

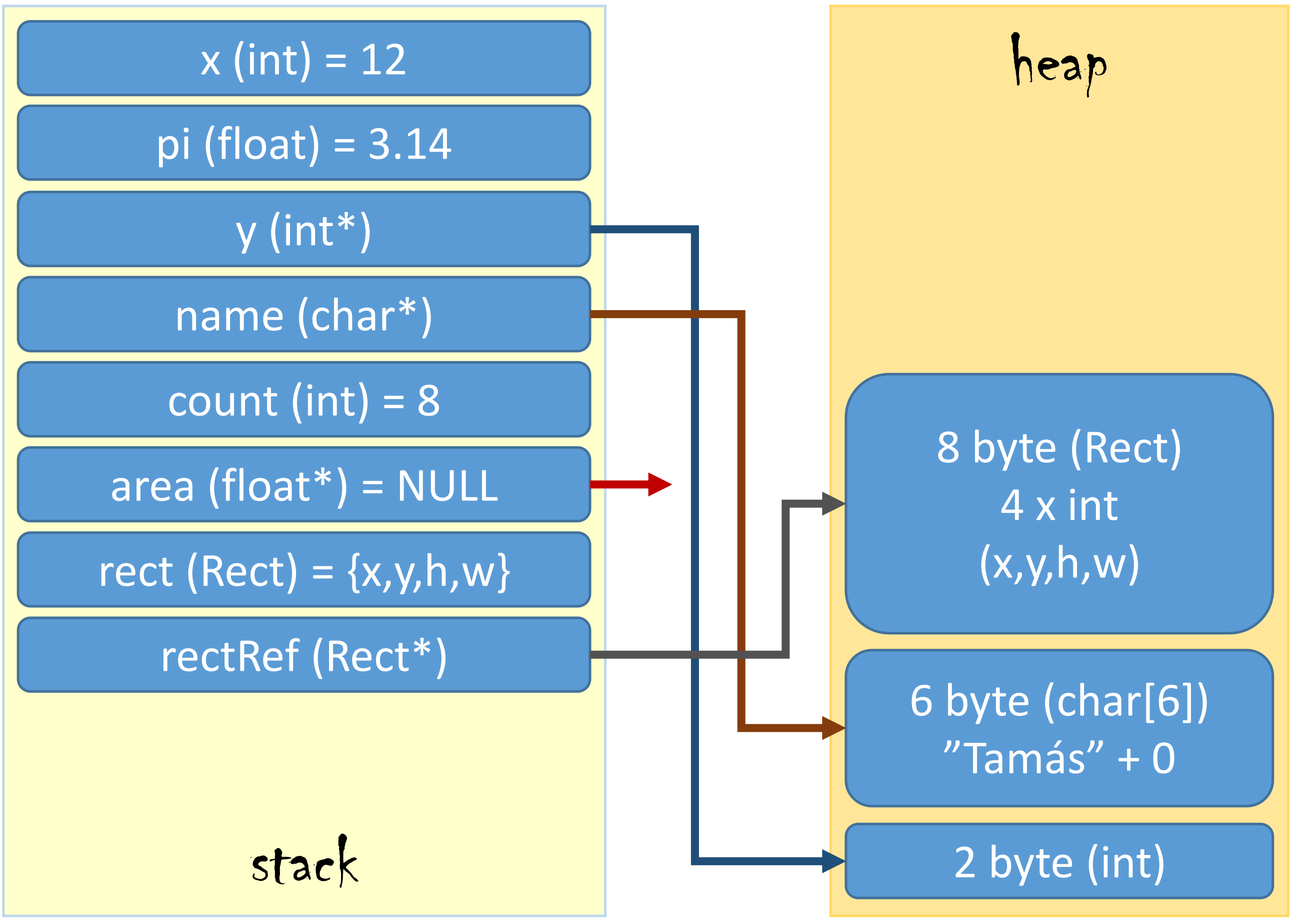
- Allocate required amount of memory
- A local variable – ***pointer type***
  - Declares the name of reference
  - Refers the exact location of the allocated area (segment:offset)
  - Specifies the size and utilization method of allocated area (size, type – meaning of bits)
  - This local variable (like others) stored in stack
- After finishing the usage, allocated space has to be released

Pointer type

Properties of pointer type variable:

- Primitive type, stored in stack
- Describes the type of referred area (typed pointer)
- Its value is the address of the pointed area
- Can point to local, static and dynamic variable (in stack or in heap)
- Its value is subject of change (the address)
- Value can be NULL (does not contain valid address)
- Accessing pointed value by ***** operator

Pointers in action



Management of pointers

- Primitive type, allocated in stack
- Its value is a memory address
- **Referred address can be changed (step forward/back, set, NULL)**
- **Movement is valid inside the allocated area**
- Then the pointer is void, no type check on usage
- Can referr valid and invalid memory address (after set or release)
- **On assignment, the address is copied, not the referred data**

Usage of pointers

```
int number = 10;
int *numPtr = &number;  // numPtr points to number
                                                  // & is the
'address of' operator
number = 20;                                 // number is 20
*numPtr = 30;                                // number is 30, set via
pointer
                                                  // * is the 'points
to' operator
numPtr = 40;                                 // WRONG!!! Invalid address is
set
```

Pointers and arrays

- Arrays are stored in heap, dinamically
- Array type variable is a typed pointer
- Ponter can be used to refer an array item
- Array item can be referenced via pointer

Pointers and arrays

Arrays can be accessed through pointers

```
int score[7] = {5, 2, 8, 0, 1, 9, 4};
int *scorePointer = score;

score[2] = 10;
*(scorePointer + 2) = 10;
```

Method parameters

- Formal parameters are value types (primitives or pointers)
- On calling, values of actual parameters are copied to formal parameters
- When a value has to be modified inside a function, its reference has to be passed as a value argument. **A pointer type formal parameter receives a reference value of an actual parameter**.

Method parameters – value/reference

```
int height=10;
int width=20;
void extend(int a, int *b) {
      a *= 2;
      *b *= 3;
}
extend(height, &width);       //height is 10, width is 60
```

Usage of pointers

```
char *string1=″Hello World!″;
char string2[20];

void stringCopy(char *dest, char *src) {
      while(dest++ = src++);
}
```

```
char src[13]={′H′,′e′,′l′,′l′,′o′,′ ′,′W′,′o′,′r′,′l′,′d′,′!′,′\0′};
char *ptr = src[0]
//    *(ptr+4) = src[4] = ′o′
```

Dynamic memory management

```
<type> *<ptr_name> = (<type>*) malloc(<total_byte_size>);
<type> *<ptr_name> = (<type>*) calloc(<unit_count>,<unit_size>);

int *intPtr = (int*)malloc(100 * sizeof(int));  //allocation
free(intPtr);
                    //release

float *floatPtr = (float*)calloc(100, sizeof(float)); //allocation
free(floatPtr);
             //release
```

Dynamically allocated memory can be handled as array

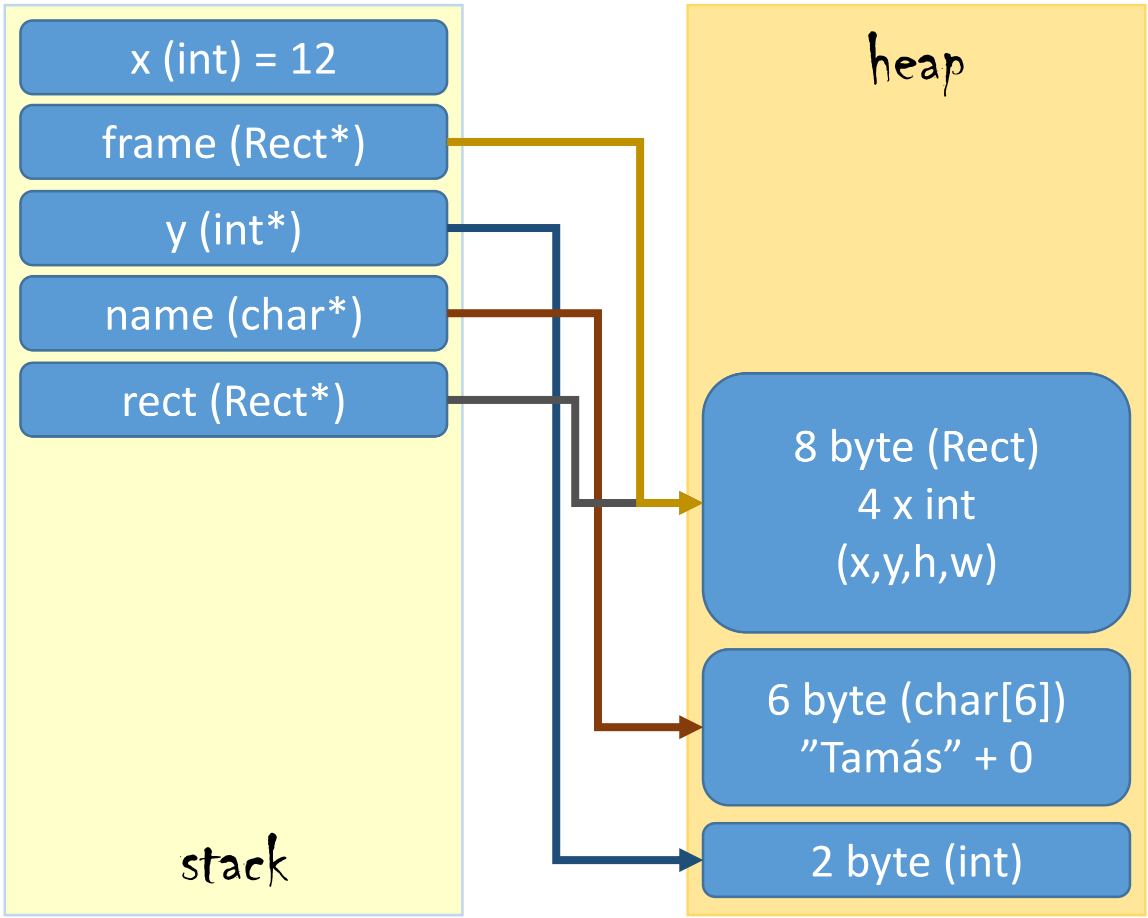
Memory allocation and release

- Reservation is done in run time, before usage
- Accessing allocated area through pointers
- After usage, release through pointer
- Release is **NOT AUTOMATIC**!
- Allocated area can be released only once
- One area can be referenced bymultiple pointers
- If there is no reference to an allocated area, it can not be accessed, nor released. The application leaks memory and can run out of resource.

Heap fragmentation

Allocated areas can be reallocated after release

If release order is not the reversed order of allocation, holes can fragmentation appears in heap.



Classification of types by access

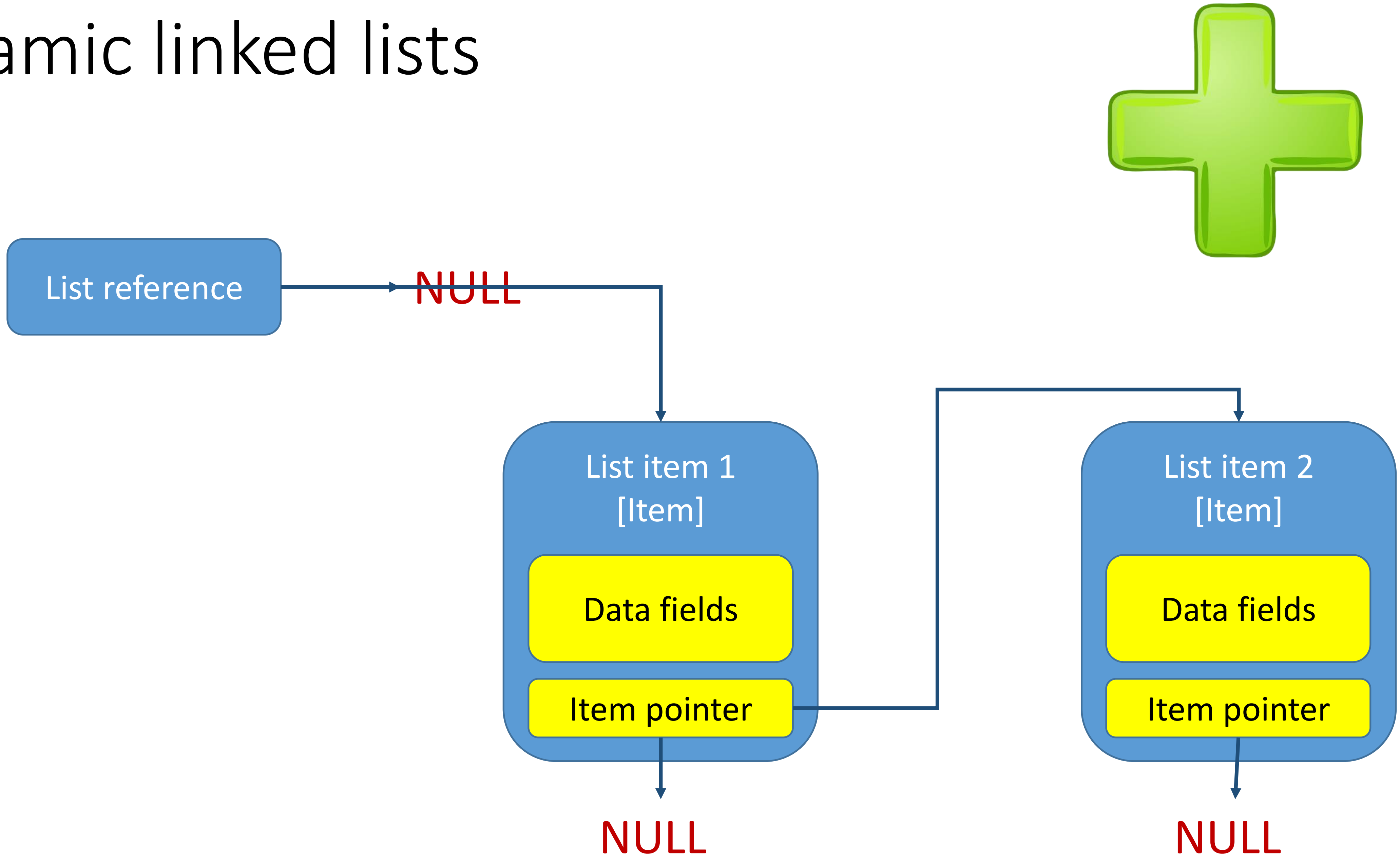
Value or pointer

- Primitive types
- Enumerations
- Typed pointers
- Void pointers

Only reference

- Arrays
- String literals (char array)
- Dynamically allocated areas (malloc/calloc)

Dynamic linked lists



Dynamic linked lists

```
typedef struct Item {
    <custom data>
    Item *nextItem
} Item;

Item *firstItem;
firstItem = NULL;
```

```
Item newItem;
newItem.nextItem = NULL;
firstItem = &newItem;
```

Function pointer

Address of first instruction of the function to execute.
Can be used like data pointers.

```
void my_int_func(int x)
{
    printf( "%d\n", x );
}
```

```
void (*foo)(int);
foo = &my_int_func;

foo( 2 );
(*foo)( 2 );
```

Subtask injection

Sort an array of a user defined class in a generic component.
With n properties, there are n! sort permutations.

```
int byName(User a, User b)
{…}

int byAge(User a, User b)
{…}
```

Subtasks

Executor

```
void sort(
    User users[], int size,
    void (*subOp)(int))
{…}
```

Injection

```
sort(users, count, &byName);

sort(users, count, &byAge);
```

Basic language components – C

- Variables, types, type casts
- User defined types
- Arrays, strings, constants
- Operators, precedence, overload
- Control statements
- Code modules and parameters
- Dynamic memory management, data references
- Function pointrers – method references