High Level Parallel Programming

C++ Basics

Alessandro Savino and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

"Hello world" Revised

C++ main differences from C

```
#include <iostream>

using namespace std;

int main(int argc, char ** argv) {
  int vi;
  float vf;
  if (argc != 1) {
    cerr << "Parameter Error!" << endl;
    return -1;
  }
  cout << "Insert variables: " << endl;
  cin >> vi >> vf;
  cout << "Values: " << vi << " " << vf << endl;
  return 0;
}
```

Console Output

Corresponding to the C printf

```
cout << “We have " << classNum << " classes!";

cout << "I am " << age << " years old!" << endl;
```

In C++ we do not need the type of the object (it is done automatically by the compiler)

- cout << expression
  - Sends expression to standard out
  - Multiple “<<“ can be chained together
- endl
  - Pre-defined "end of line" variable
  - Equivalent to '\n'

Console Input

Corresponding to the C scanf

```
int age;

cout << "Please enter your age: ";
cin >> age;
cout << "You are " << age << " years old!";
```

In C++ we do not need the type of the object (it is done automatically by the compiler)

- cin >> variable
  - Reads input from console & stores in variable
  - Usually preceded by a prompt message to user:
  - cin isn't great – difficult to detect invalid input
    - Also reads word at a time (not optimal for reading strings)

Basics of a Typical C++ Environment

- Input/output
  - **cin**
    - Standard input stream
    - Normally keyboard
  - **cout**
    - Standard output stream
    - Normally computer screen
  - **cerr**
    - Standard error stream
    - Display error messages

C style:
scanf (...);
fscanf (stdint, ...);

C style:
printf (...);
fprintf (stdout, ...);

C style:
fprintf (stderr, ...);

Namespace

```
using namespace std;
```

- Libraries separate their symbols (functions, variables) into namespaces
- The "using" declaration brings symbols from the library's namespace into the global scope of program
- Without the "using" declaration, we need to indicate where symbol comes from by using namespace and :: before the symbol

Example

```
using namespace std;

cout << ... << end;
cin >> ...
cerr << ...
```

Need to be placed in all cpp files (even for multi-file projects)

```
std::cout << ... << std::endl;
std::cin >> …
std::cerr <<
```

Data Types

- C++ data types
  - Integers
    - short / unsigned short
    - int / unsigned int
    - long / unsigned long
  - Reals (IEEE 754)
    - float
    - double
  - Characters
    - char / unsigned char
    - wchar_t
  - Logical values
    - bool

Over 1 byte

More than 1 bytes

Derived Data Types

- Enumeration types

```
enum colors_t {
  black, blue, green, cyan, red, purple, yellow, white
};
```

  - Each label in {} is an integer value to have labels but compress data requirements

- Arrays

```
int v[10];
```

  - Not a container, an array in C fashion

Derived Data Types

❖ Classical pointers

```
int* ptr;
int *ptr;
```

The size does not depend on type because pointers are **memory addresses**

- The main difference from C is that **NULL** (or 0) is, in fact) **nullptr**
  - A pointer should always be set to **nullptr** before and after usage
- Operators * and -> are "de-referencing" operators

Reference

- Reference

```
int i = 0;
int& r = i;
```

Variable r is an alias of variable i

- r defined as a reference of variable i and it can exist only if it exist the variable
- Access a reference is equivalent to accessing the original value
- A reference **cannot** be assigned to **nullptr** it **must** be assigned to a **variable**
- No standard implementation
  - Most of the times the compiler translate a reference into a pointer with de-referencing operations

Structs and Unions

- Structures
  - Heterogenous data linked together by logical (problem based) constraints

```
struct product {
  int weight;
  double price;
};
```

Also available in C

Standard C definition

Structs and Unions

Also available in C

- Unions
  - Single memory location to access the same bit configuration based on different types
    - Size equal to the biggest type in union

```
union mytypes_t {
  char c; int i; float f;
};
```

All the types are merged together, and the size of the union is equal to the longer of the fields

I can do operations on integer (e.g. bit field operations) and interpreting the result as a float

Classes

- As in other languages, a class is a collection of
  - Data variables
  - Methods, i.e., functions, with access identifiers

Name of the class

Objects can be **private** (to the class) or **public** (to everybody) or, eventually, protected

```
class ResultCode {

  private:
    int code;

  public:
    int get_code() { return code; }
    char* get_description();

};
```

Classes

- The methods can be defined
  - Inline
  - In proper external files (other *.cpp files)
- Try to avoid mixed solutions in the same class

There is one exception: Generic programming

```
class ResultCode {

  private:
    int code;

  public:
    int get_code() { return code; }
    char* get_description();

};
```

Mixing solutions makes the interface full of details and the implementation partial

Encapsulation

- Both data and member function can have different access properties
  - Everything is private, by default, if no access specifier is present
- Encapsulation is one **key** concept of OOPs
  - Define objects as private as long as possible
  - Define methods as private whatever needed
  - Define objects/methods are **protected** when they need to be inherited by sub-classes
    - Protected is like private but objects can be inherited by sub-classes

Example: Revisited

Interface

```
class ResultCode {
  private:
    int code;
  public:
    int get_code();
    char* get_description();
};
```

The key private can be erased (default)

Private and public objects can be interleaved, but it if cleaner to separate them

Advice:
It is better to put public before (when I read the code I immediately see public methods which the ones I can use from outside)

```
int get_code() {
  return code;
}

char* get_description () {
  ...
}
```

Implementation

Project (file) organization

- A C++ project must be organized as a
  - Set of header files, including the class definitions
    - <class_name>.h
  - Set of C++ files, including the header files and the implementations of the methods
    - <class_name>.cpp
  - Cpp files are compiled (without linker issues of duplicated symbols)
  - If a strict separation is maintained, it is possible to create a package **library**
    - Passing the library to rhitd-party
    - Keeping details hidden

Example

Interface
my_class.h

To avoid symbol duplication
Please, refer to "Algorithms and programming" for further details

Implementation
my_class.cpp

```
#ifndef MYCLASS
#define MYCLASS

class MyClass {
  char* ptr;
public:
  int doWork();
};

#endif
```

```
#include "my_class.h"

int MyClass::doWork() {
  //codice
  return 0;
}
```

Include symbols

Method (doWork) of a class (MyClass):
Name of the class (MyClass) +
Scope identifier (::)

Object memory management

- Classes are **instantiated** into objects
  - It's the only time when all data associated to the class is ever allocated in memory
  - Data is **not** shared among objects
    - For example, two instantiations of MyClass have different **ptr** pointers
      - They may even refer to the same object **but** they are anyway stored in different locations

```
class MyClass {
  char* ptr;
  public
    int doWork();
};
```

  - Code **is** shared among objects
    - For example, two instantiations of MyClass share the same code

Constructors

- A constructor is a special function that initializes the object when it is created
- Due to polymorphism, it is possible to have different constructors for the same object
- A constructor has
  - No return
  - The same name of the class
- Different constructors are differentiated in terms of the different set of parameters (i.e., a different signature) in number and/or type

Example

Everything is in-line for the sake of simplicity

Default constructor (no parameters)

New definition type:
After calling the constructor **code** will be defined and set to 0

Extra constructor (1 parameter)

After calling the constructor **code** will be defined and set to the parameter c

```
class ResultCode {

public:
  ResultCode(): code(0) {}
  ResultCode(int c) { code=c; }
  int getCode();
  char* getDescription();

private:
  int code;
};
```

Constructors

- Constructors are automatically invoked several times, i.e., whenever an object is defined
  - Explicit definition
    - my_class obj1, obj(2);
  - Parameter by value
    - fn(my_class obj_param)
  - Return value
    - return myobj;
  - In general, anytime an object (a class instance) is copied …

Default constructor

Constructor with 1 parameter

A new object is built by the constructor and all objects are copied in it

Assigned to an existing object outside the function

Destructors

- As there is a constructor there is also a destructor
- The destructor is a special function (only one per class) that is deputed to clear any internal (dynamic) resources handled by the object before destruction
  - Same name of the class with a ~ in front
  - No parameters
    - No polymorphism on the destructor
  - No direct calling
    - Only the compiler schedule its call
    - Automatic call, once per abandoned object

With the constructor we have no direct call, but we decide which one to call (with the parameters)

Destructor

- No need of a destructor if the class handles only static resources
  - We only need to free dynamic memory, objet descriptors, etc.
  - Same procedure used for containers

Example

```
class ResultCode {

public:
   ResultCode(): code(0) {}
   ResultCode(int c): { code=c; }
   ~ResultCode() {
     /* do somenthing */
   }
   int getCode();
   char* getDescription();

private:
  int code;

};
```

Destructor

Function Parameters

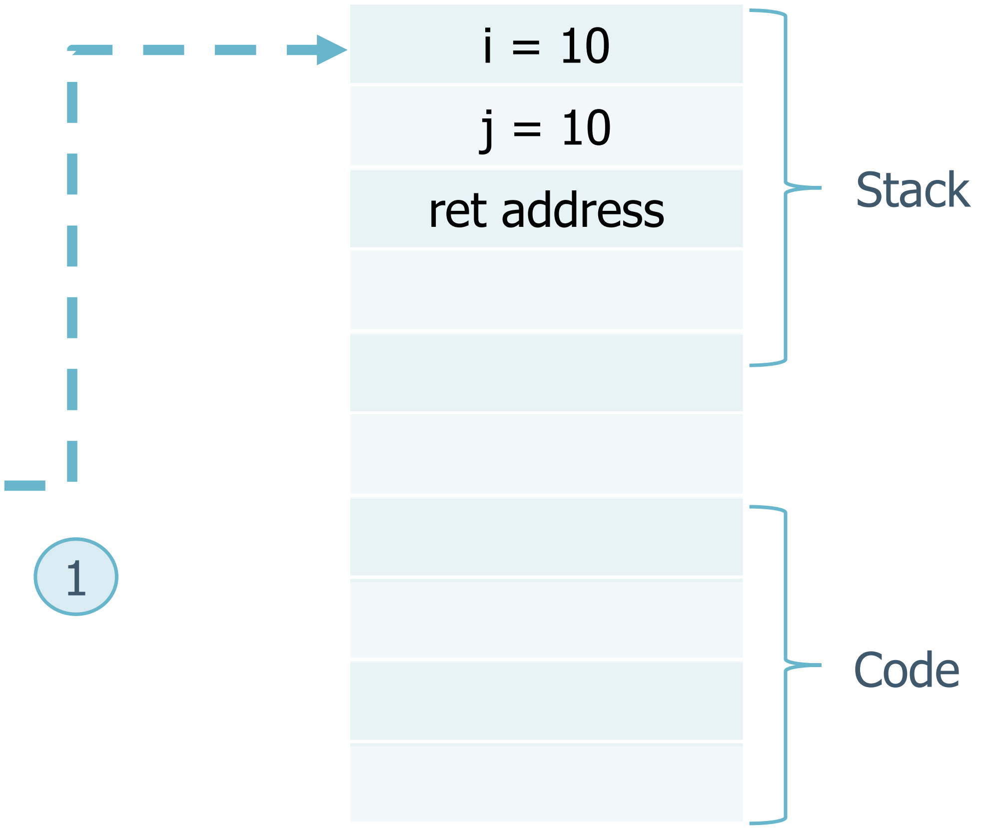
- Differently from the C language, C++ functions may have 3 different types of parameters
- Parameters, can be passed by
  - Value
  - Address (pointer)
  - Reference

Function Parameters

- By Value
  - When the function is called, it will hold a copy of the parameter
  - Changes to the local parameter will not affect the original variable

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



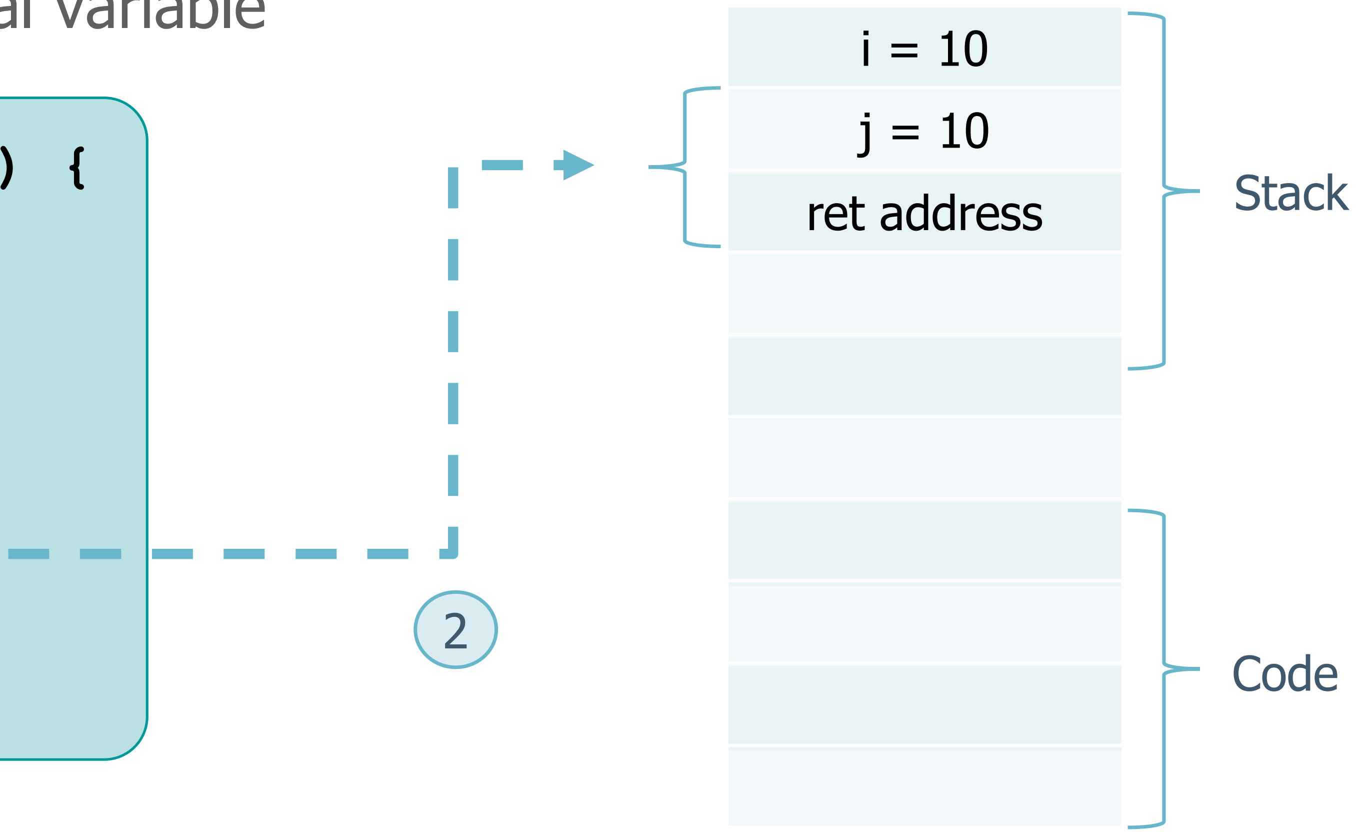
Function Parameters

➢ By Value

- When the function is called, it will hold a copy of the parameter
- Changes to the local parameter will not affect the original variable

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

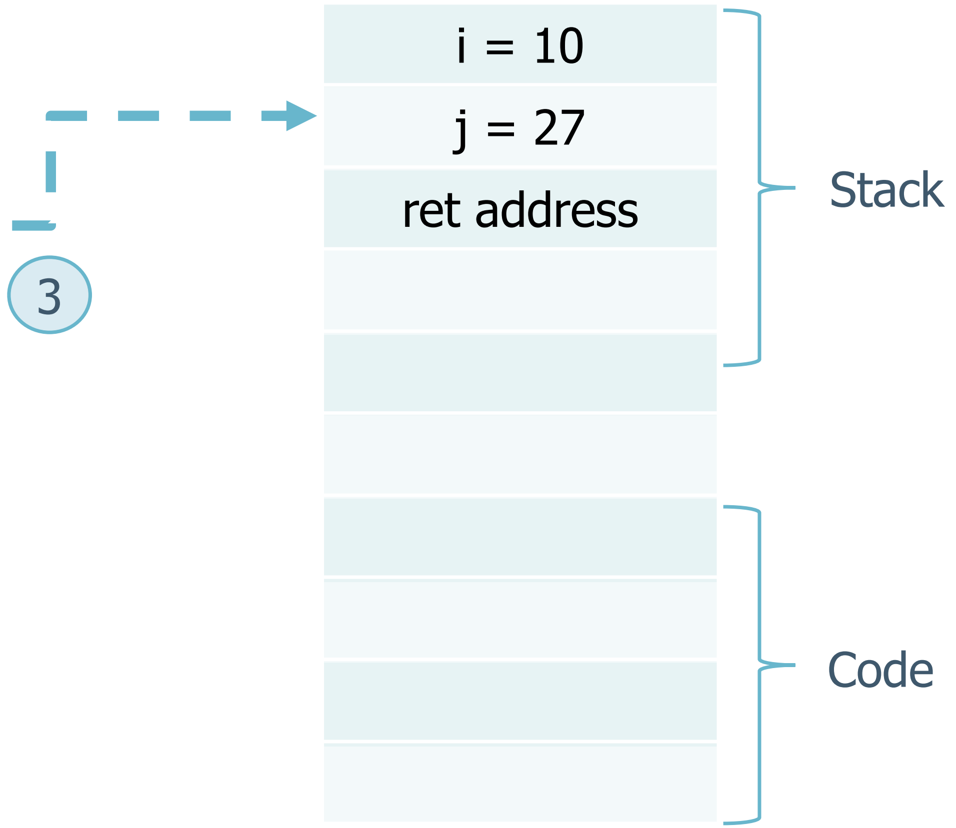


Function Parameters

- By Value
  - When the function is called, it will hold a copy of the parameter
  - Changes to the local parameter will not affect the original variable

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

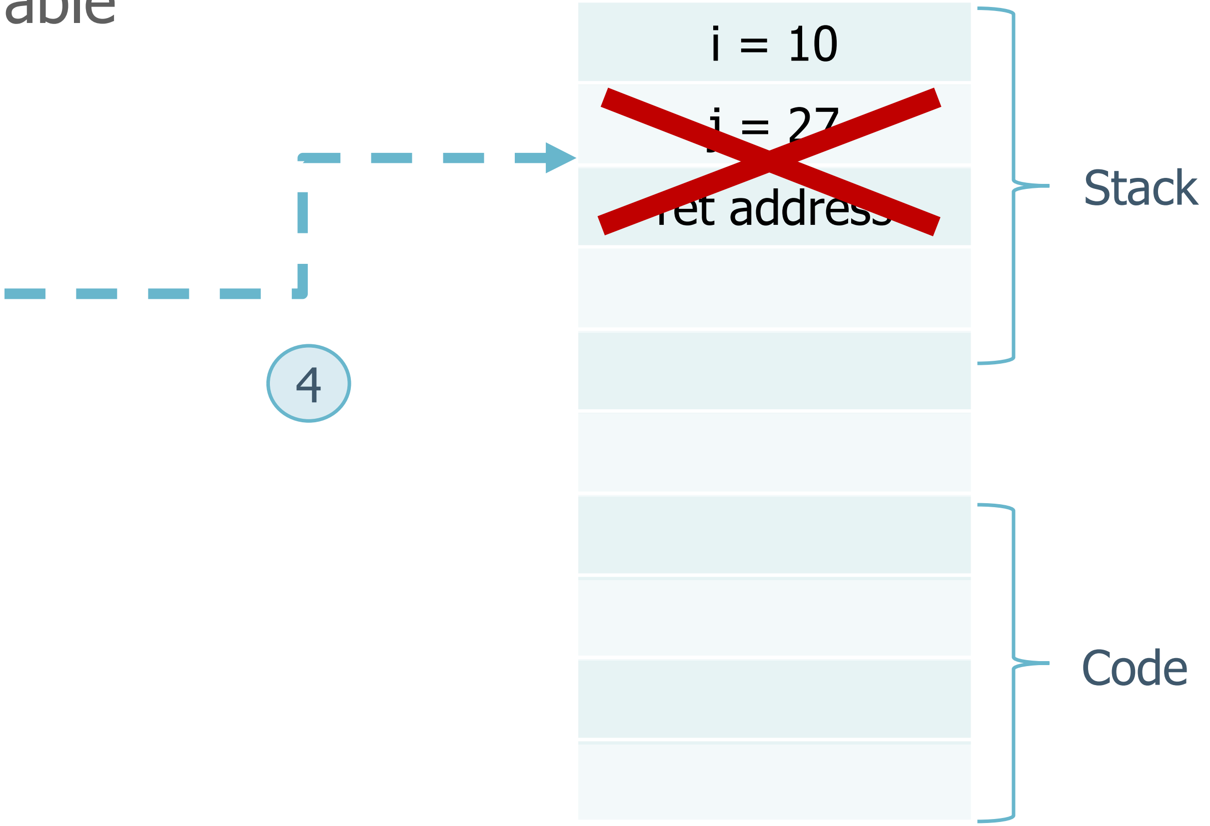


Function Parameters

- By Value
  - When the function is called, it will hold a copy of the parameter
  - Changes to the local parameter will not affect the original variable

```c
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



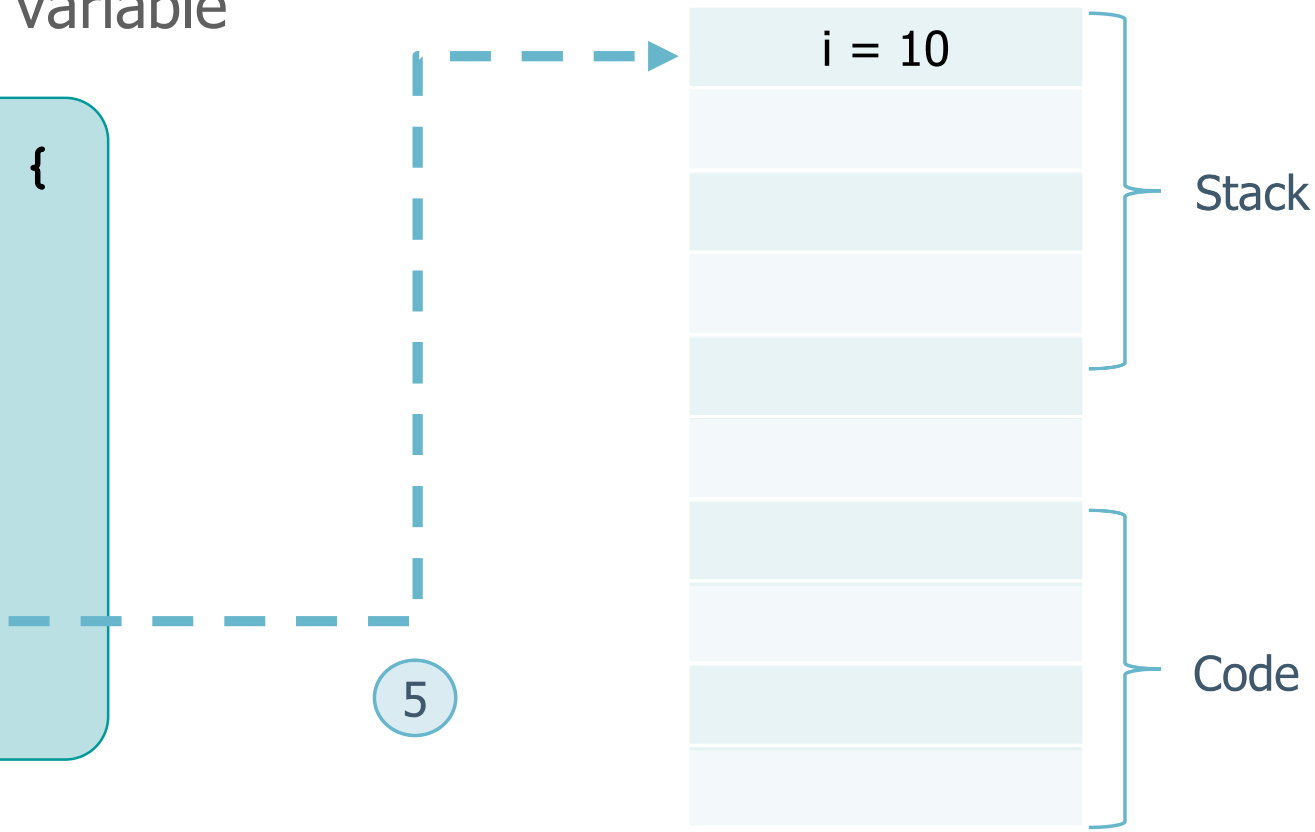
Function Parameters

By Value

- When the function is called, it will hold a copy of the parameter
- Changes to the local parameter will not affect the original variable

```
void f(int j) {
  j=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```



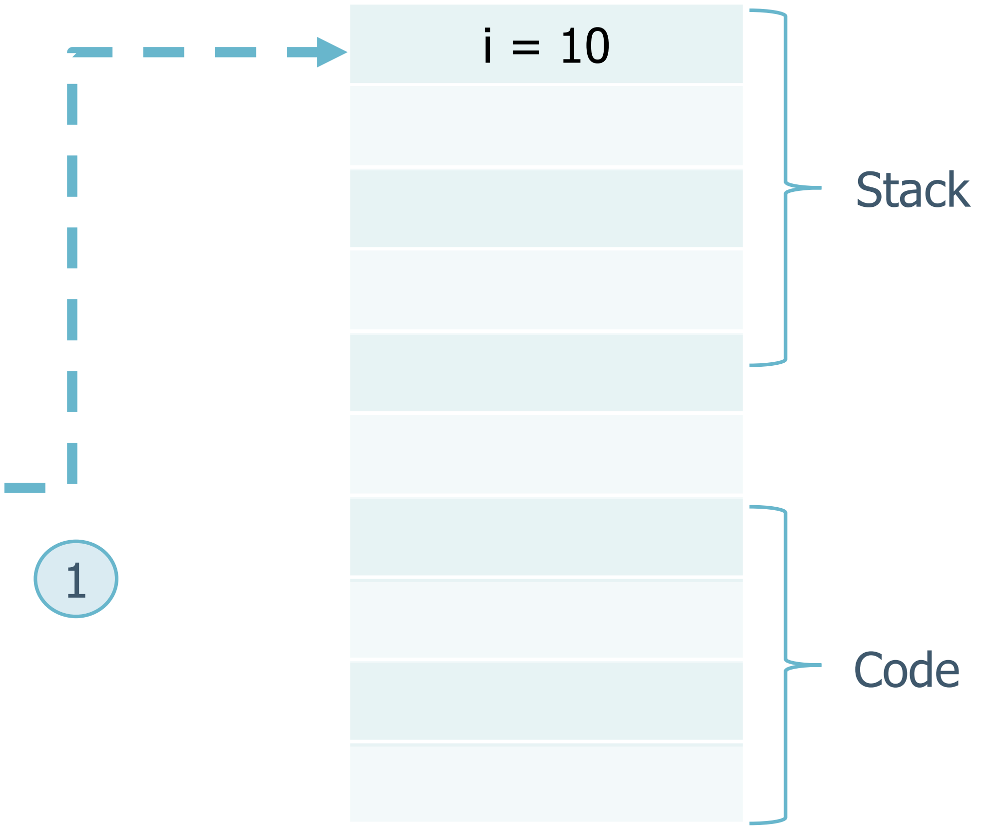
Function Parameters

- By Address (pointer)
  - An address is provided to the function
    - A copy of the address is done, not a copy of the data referenced by the address itself
      - Unfortunately, an address can be **nullptr**, thus a variable by reference can be invalid
    - By deferencing the address the function might modify the original data

Function Parameters

- By Address (pointer)
  - An address is provided to the function
    - A copy of the address is done, not a copy of the data referenced by the address itself

```
void f(int *p) {
  if (p!=nullptr)
    *p = 27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```

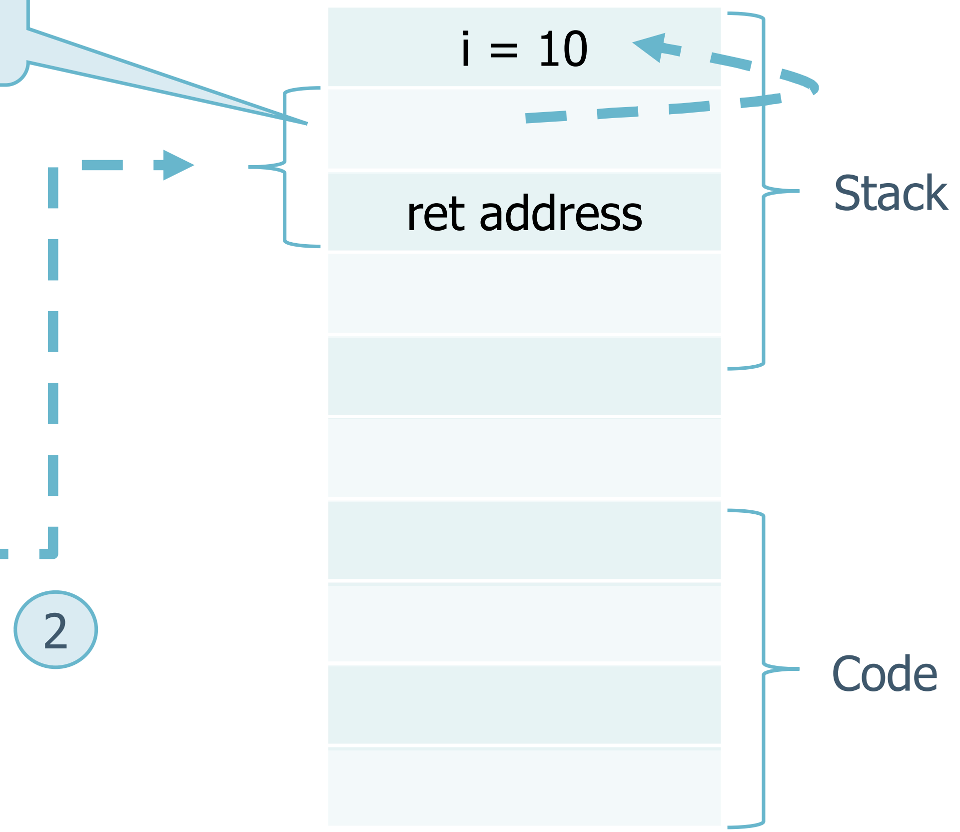


Function Parameters

- By Address (pointer)
  - An address is provided to the function
    - A copy of the address is done, not a copy of the data referenced by the address itself

The new location does not contain an integer but a pointer

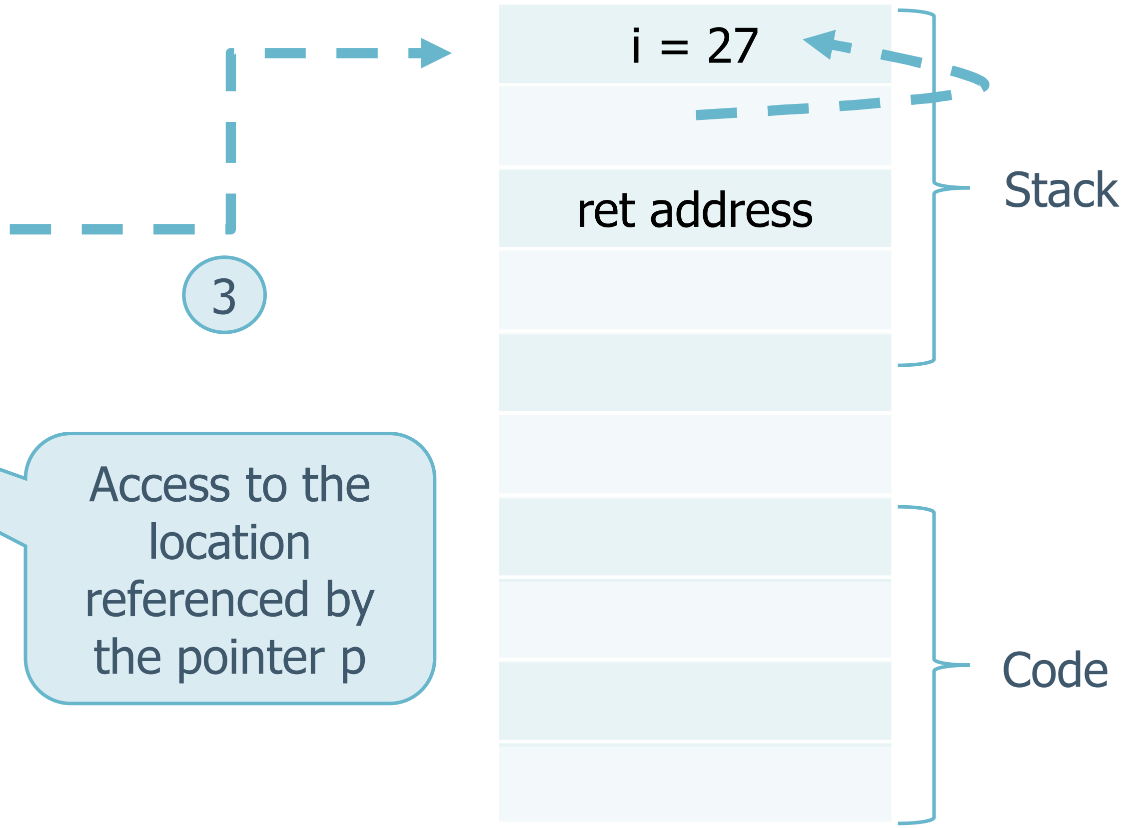
```
void f(int *p) {
  if (p!=nullptr)
    *p = 27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



Function Parameters

- By Address (pointer)
  - An address is provided to the function
    - A copy of the address is done, not a copy of the data referenced by the address itself

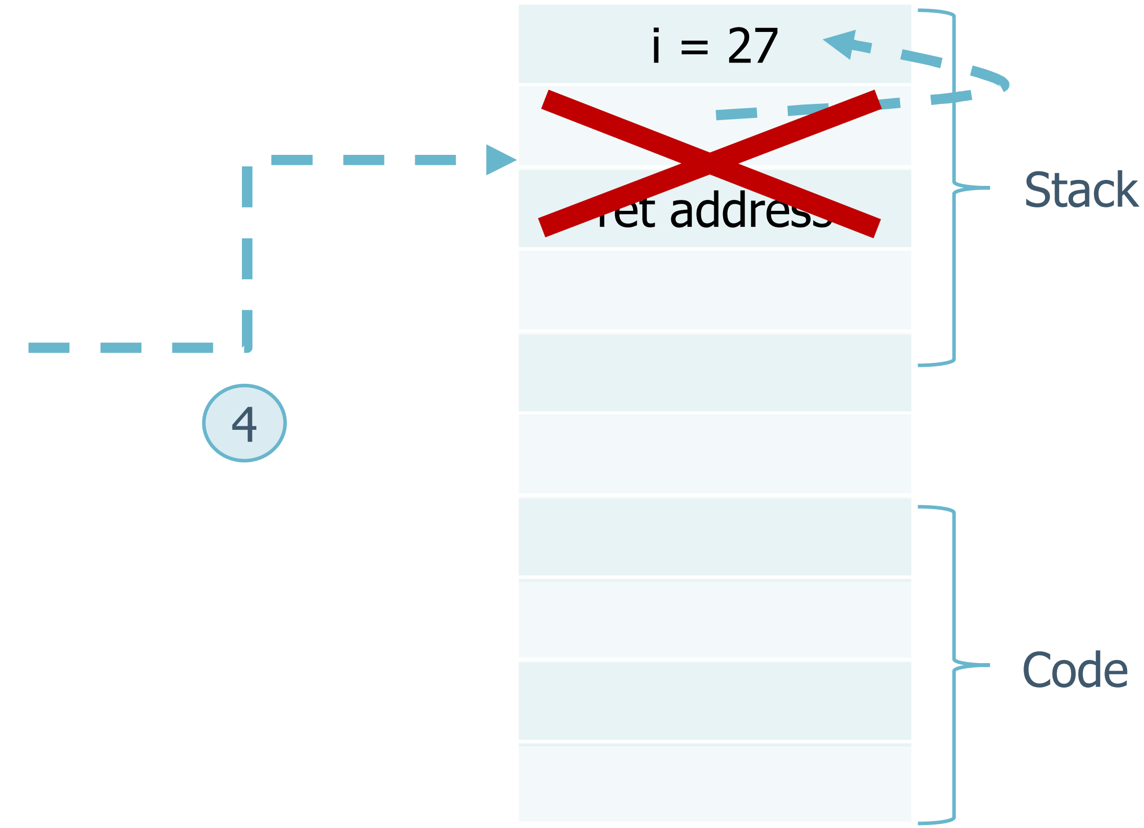
```
void f(int *p) {
  if (p!=nullptr)
    *p = 27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



Function Parameters

- By Address (pointer)
  - An address is provided to the function
    - A copy of the address is done, not a copy of the data referenced by the address itself

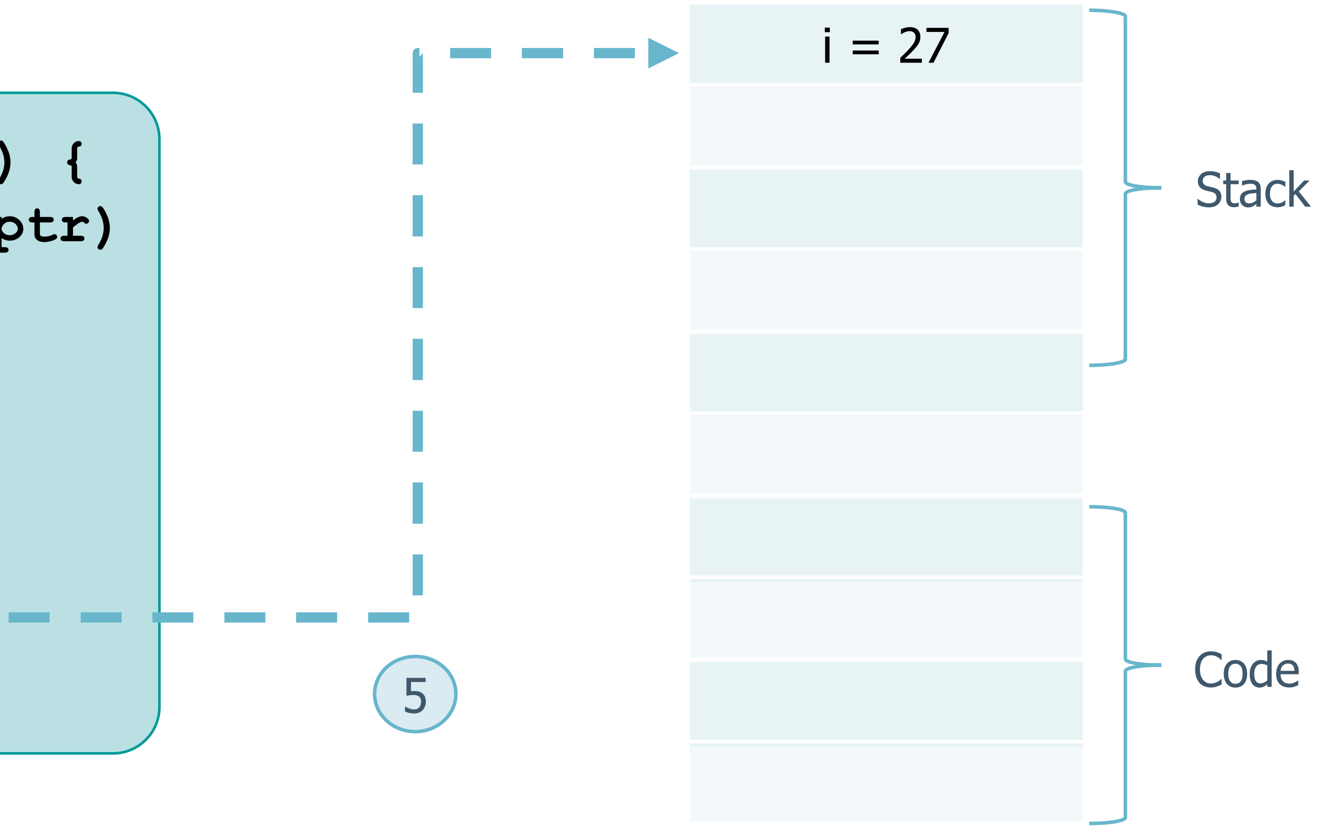
```
void f(int *p) {
  if (p!=nullptr)
    *p = 27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



Function Parameters

- By Address (pointer)
  - An address is provided to the function
    - A copy of the address is done, not a copy of the data referenced by the address itself

```
void f(int *p) {
  if (p!=nullptr)
    *p = 27;
}
int main() {
  int i=10;
  f(&i);
  return i;
}
```



Function Parameters

New parameter passing strategy

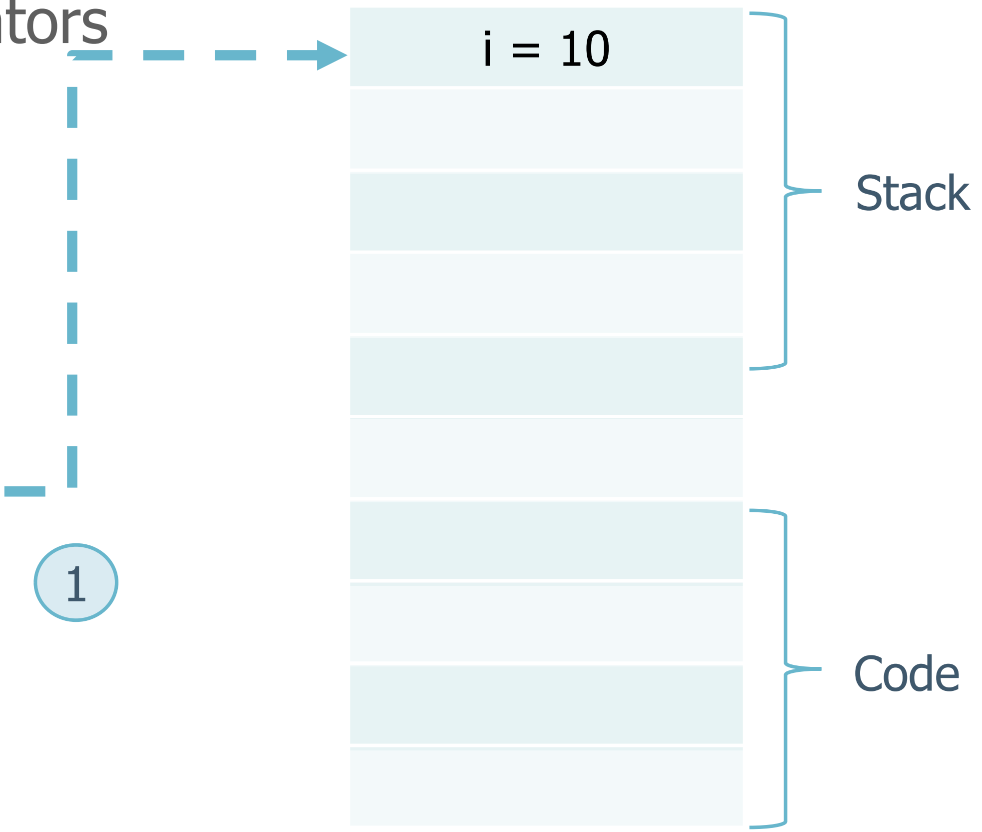
- By Reference
  - A pointer to a verified variable is passed
  - We never have **nullptr** pointers
  - The parameter is accessed directly without need of dereferencing operators

No * but &

```
void f(int& r) {
  r=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

No &



Function Parameters

New parameter passing strategy

- By Reference
  - A pointer to a verified variable is passed
  - We never have **nullptr** pointers
  - The parameter is accessed directly without need of dereferencing operators

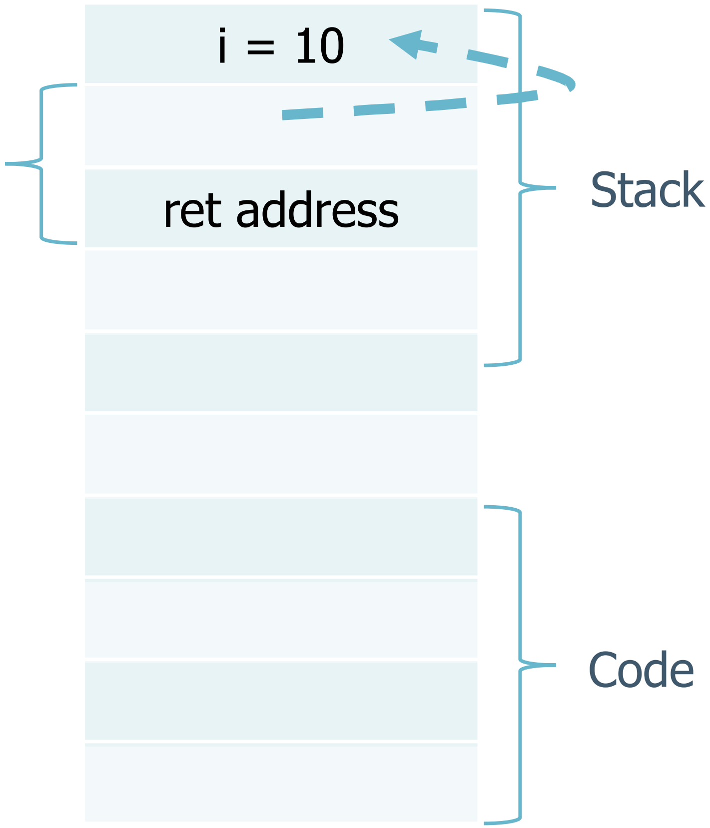
No * but &

```
void f(int& r) {
  r=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

No &

2



Function Parameters

New parameter passing strategy

- By Reference
  - A pointer to a verified variable is passed
  - We never have **nullptr** pointers
  - The parameter is accessed directly without need of dereferencing operators

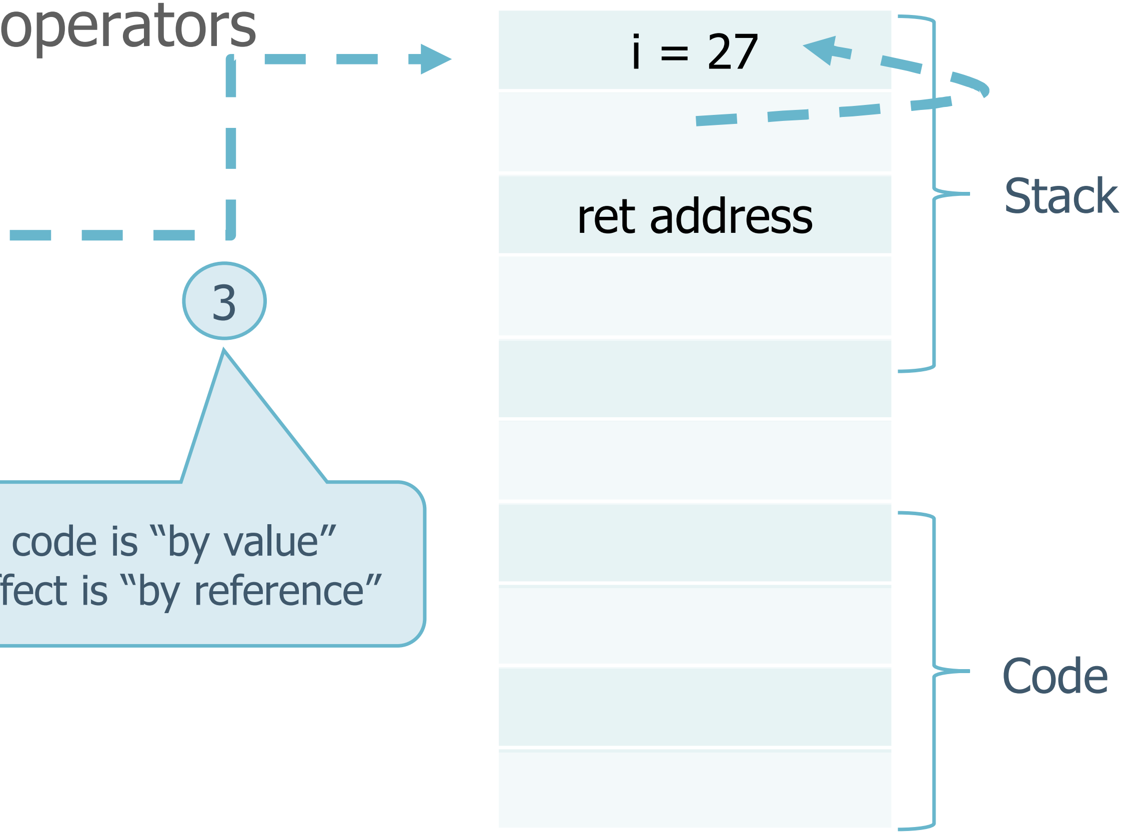
No * but &

```
void f(int& r) {
  r=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

No &

The code is "by value"
The effect is "by reference"



Function Parameters

New parameter passing strategy

➢ By Reference

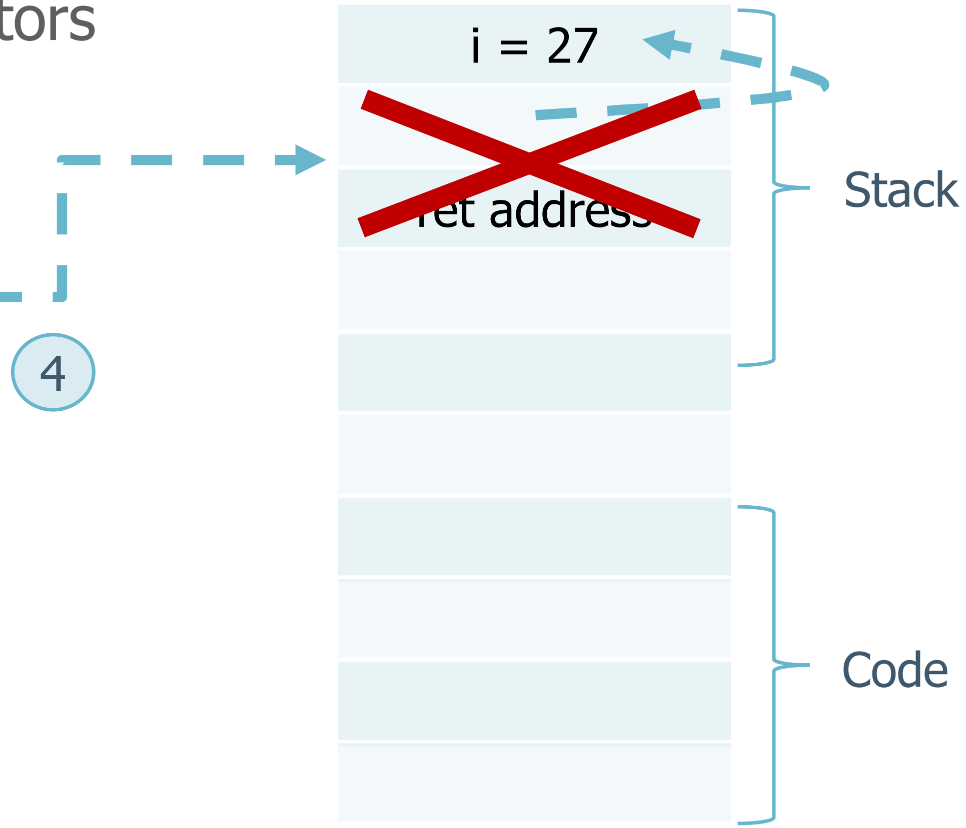
- A pointer to a verified variable is passed
- We never have **nullptr** pointers
- The parameter is accessed directly without need of dereferencing operators

No * but &

```
void f(int& r) {
  r=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

No &



Function Parameters

New parameter passing strategy

➢ By Reference

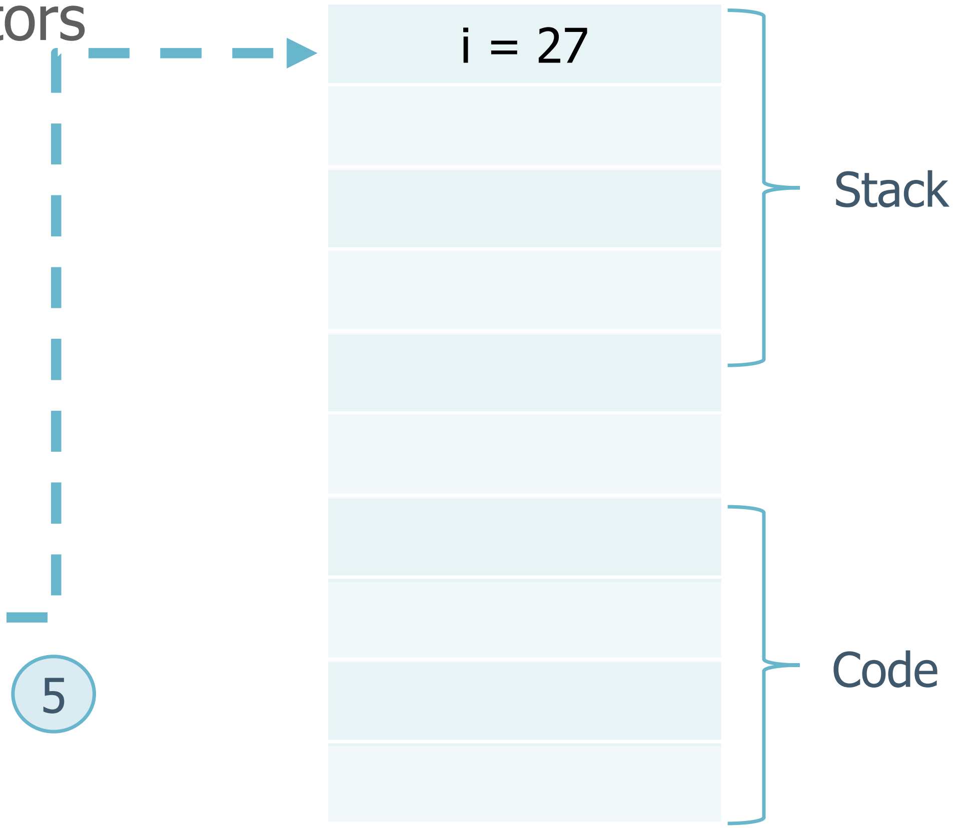
- A pointer to a verified variable is passed
- We never have **nullptr** pointers
- The parameter is accessed directly without need of dereferencing operators

No * but &

```
void f(int& r) {
  r=27;
}

int main() {
  int i=10;
  f(i);
  return i;
}
```

No &



Dynamic Memory Allocation

- Data can be allocated into the heap memory section of a program during its execution
  - This derives directly from C
  - Dynamic allocation means that the memory management of the dynamic data is up to the program to handle
  - There are several functions to allocate memory dynamically
    - malloc, calloc → new
    - realloc → does not exist !
    - free → delete

Must be done manually (allocate new structure and copy from old to new)

New

- The **new** operator allocates a memory block compatible with the type and the size defined
  - Object allocation **fires** the constructor of each allocated object
  - The are two versions of new
    - The **normal** one fires an exception (to be properly handled) so the pointer is never **nullptr**
    - The **nothrow** one does not fire any exception but returns a **nullptr** when the allocation is not possible

Closer to malloc and calloc

Example

```
using namespace std;
...

int * var = new int;

int * var_init = new int(12);


int * vect1 = new int[10];

int * vect2 = new (nothrow) int[20];



my_class *pToObj = new my_class;
```

Single variable with no initialization (malloc)

Single variable initialized to 12 (somehow calloc)

Array of size 10
vect1 cannot be nullptr

Array of size 20
vect2 **can** be nullptr

Dynamic object; pToObj is the pointer to the object
It calls the default constructore after creaation

Delete

- The **delete** operator is dual to
  - It calls the destructor
  - Release the memory block
    - For each new call a delete
    - First call new, then call delete
  - Two versions
    - **Single** one applies only to single data, such as variables or single objects
    - **Multiple** one applies to multiple blocks, such as vectors

Example

```
int * var = new int;
int * var_init = new int(12);
int * vect1 = new int[10];
int * vect2 = new (nothrow) int[20];
my_class *pToObj = new my_class;
```

Previous news

Subsequent deletes

```
using namespace std;

...

delete var;
delete var_init;

delete[] vect1;
delete[] vect2;

delete pToObj;
```

Single variables

Arrays

Calls the default destructor before release

Dangling pointers

- Dangling pointers are pointers that do not point to a valid object of the appropriate type
- Dangling pointers
  - Generate memory safety violations
  - Are typically generated when memory is released
- It is very difficult to trace-back dangling pointers
  - Dangling poitners often imply memory leaks

Dangling pointers

- To avoid dangling pointers, always set any ptr to nullptr (and check against before usage) when
  - A pointer is created but not assigned to an address
    - int *ptr; → int *ptr = nullptr;
  - After the delete operations
    - delete ptr; → delete ptr; ptr = nullptr;

Types and Qualifiers

- Any <type> in C++ (except functions and reference types) can be cv-qualified
  - Const-qualified
    - const T
  - Volatile-qualified
    - volatile T
- cv-qualifiers can appear in any order, before or after the type
  - volatile const
  - const volatile

Types and Qualifiers

- Semantics
  - Constant objects cannot be modified by the user once defined and initialize
  - Volatile objects prevent the compiler to optimize the code segments involving the object
    - The compiler must suppose that the value of the object can be changed by means undetectable by the compiler
    - Volatile should be avoided in most cases
      - Deprecated from C++20

Example

```
int i = 100;
while (i == 100) {
   ...
}
```

If i is not modified, the compiler may be tempted to change the while construct into while(true)

```
volatile int i = 100;
while (i == 100) {
   ...
}
```

To avoid the compiler optimization define the object as volatile

Types and Qualifiers

- When a type conversion is required, the C language allows to resort to cast operations
  - Syntax
    - (<type>)<var_to_convert>
  - The C language does not perform any check about the accuracy/meaning of the cast, it only checks its feasibility
- C++ expands the conversion mechanism by providing a more meaningful (and safe) syntax

Static cast

```
static_cast <new_type> (expression)
```

- The static_cast conversion is used to cast between related types
  - Converts the value of **expression** to a value of **new_type**
    - **new_type** must be at least as cv-qualified as the type of expression
  - It performs the conversion but it also checks whether the conversion is acceptable
  - Can be used to convert void pointers to pointers of another type

Example

Polimorphism

Type Cast

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

int main() {
    int a = 42, y;
    double b = 3.14, x, z;

    x = sum(a, b);
    y = sum(a, static_cast<int>(b));
    z = sum(static_cast<double>(a), b);
}
```

Error
Do not know which prototype to use

OK
Using the second prototype

OK
Using the first prototype

Pointer Cast

```
int i = 42;
void* vp = &i;
int* ip = static_cast<int*>(vp);
```

Check automatically performed

Reinterpret cast

```
reinterpret_cast <new_type> (expression)
```

- The reinterpret_cast conversion is used to convert between unrelated types
  - It converts the underlying bit sequence representing **expression** as a value of **new_type** without any logic check
    - new_type must be at least as cv-qualified as the type of expression

Reinterpret cast

- Usually it does not generate any CPU instructions
  - It is a purely compile-time directive which instruct the compiler to treat **expression** as if it had the type **new_type**

Similar to what happens within different union fields

- Only a very restricted set of conversions is allowed using reinterpret_cast
  - A pointer to an object can be converted to a pointer to std::byte, char or unsigned char
  - A pointer can be converted to an integral type (typically uintptr_t)

Reinterpret cast

- Invalid conversions usually lead to undefined behavior
- It is undefined behavior to access an object using an expression of different type
  - We are not allowed to access an object through a pointer to another type (pointer aliasing)
  - Consequently, compilers typically assume that pointers to different types cannot have the same value
- There are very few exceptions to this rule

Example

```
#include <cstdint>
#include <iostream>
using namespace std;

int f() { return 42; }

int main() {

  uintptr_t v1 = reinterpret_cast<uintptr_t>(&i);
  cout << "The value of &i is 0x" << hex << v1 << '\n';

  void(*fp1)() = reinterpret_cast<void(*)()>(f);
  fp1();
```

From pointer to integer

Static cast would be an error

From a pointer to a function to another pointer

Calling fp1 would imply an undefined behavior

Example

Correct and safe

```
int(*fp2)() = reinterpret_cast<int(*)()>(fp1);
cout << dec << fp2() << '\n';
```

type aliasing through pointer

```
int i = 7;
char* p2 = reinterpret_cast<char*>(&i);
if (p2[0] == '\x7')
  cout << "This system is little-endian\n";
else
  cout << "This system is big-endian\n";
```

Type aliasing through reference

```
reinterpret_cast<unsigned int&>(i) = 42;
cout << i << '\n';
}
```